Trie Automata for Constrained Decoding over Large Finite Sets

Xingzi Xu, Karim Bouyarmane
Amazon
{xingzixu,bouykari}@amazon.com

Abstract

Large language models increasingly need to generate structured outputs that conform to predefined schemas, with one common constraint being selection from a finite set of valid strings. Current constrained decoding systems handle this through general-purpose grammar compilation, which becomes prohibitively slow as the number of valid values grows into the thousands, a *cardinality wall*. We introduce the *trie automaton*, a specialized mechanism that exploits finite-set structure (shared prefixes, bounded depth, known cardinality) via Aho-Corasick multi-pattern matching to precompute per-node token masks. The trie achieves 7× faster per-step valid-token computation (0.65μs vs. 5.8μs) compared to XGrammar, one of the primary backends in vLLM and SGLang, and 2–6.5× faster compilation at $K \geq 300$. Because precomputed masks enable a stateless serving path that bypasses the guided decoding pipeline, this advantage compounds in batch serving: end-to-end vLLM throughput reaches 219 req/s vs. XGrammar's 7.5 req/s at batch size 256 (29×). The 29× combines the algorithmic speedup with integration-path savings that only precomputed masks can unlock. Across seven tokenizer families (32K–262K vocabulary), the trie maintains sub-100ms compilation up to $K = 10{,}000$ and flat per-step cost regardless of set size, while guaranteeing 100% output validity.

1 Introduction

Constrained decoding has become the standard mechanism for guaranteeing that LLM outputs conform to a schema (Willard & Louf, 2023; Geng et al., 2025). By masking invalid tokens at each generation step, it eliminates malformed JSON, hallucinated field names, and invalid values. Major LLM providers now offer it, and open-source engines like Outlines, XGrammar (Dong et al., 2024), and SGLang (Zheng et al., 2024) have made it accessible to any application. These systems compile a JSON schema (or grammar) into a general-purpose automaton (a finite-state machine, pushdown automaton, or Earley parser) and use it to mask tokens at each decoding step. This architecture handles arbitrary schemas, including nested objects, recursive structures, and complex regex patterns. However, it applies the same general-purpose compilation pipeline to all constraints, regardless of their actual complexity. A deeply nested recursive JSON schema and a flat list of 1,000 tool names both undergo the same compilation pipeline, a fundamental mismatch between general-purpose engines and simple constraints.

This uniformity creates a bottleneck for one of the most common constraints in production: *select one string from a known finite set*. OpenAI's structured outputs impose a 1,000 enum limit (OpenAI, 2024), Google Gemini fails at approximately 120 enum values (Google, 2024), and Anthropic's 180-second compilation timeout (Anthropic, 2025) implies a similar wall at a few hundred values. These limits are increasingly consequential as LLM applications shift from open-ended generation to structured tool use. In agentic workflows, an LLM must select which tool to invoke from a registry that may contain 500–5,000+ APIs (Qin et al., 2024; Fei et al., 2025; Gaurav et al., 2025); as Model Context Protocol (MCP) ecosystems grow and organizations expose internal services as tools, these registries expand rapidly, often exceeding provider enum limits within months of deployment. The same pattern

appears in zero-shot classification over label sets like product taxonomies (1,500+ categories), ICD-10 medical codes (68,000 (Nguyen et al., 2023)), or legal case types (10,000+); in entity linking against knowledge bases (De Cao et al., 2021) with tens of thousands of entries; and in dynamic per-query constraints from retrieval-augmented systems where the valid set changes each query, preventing amortization of compilation costs. In all these cases, the constraint is a finite union of strings $s_1|s_2|\cdots|s_K$. While this is a regular language with no Kleene star, recursion, or nested structure, current systems compile it through the same regex-to-NFA-to-DFA pipeline used for arbitrary grammars, at a cost that grows with both the number of strings and the alphabet size. This creates what we term the *cardinality wall*: a maximum K beyond which constrained decoding becomes impractically slow.

The core insight is that different constraint types deserve different enforcement mechanisms. A finite set of strings has exploitable structure: shared prefixes, finite depth, and known cardinality. We introduce the *trie automaton*, a drop-in replacement for the FSM layer in existing constrained decoding pipelines, specialized for finite-set constraints. It (1) builds a character-level trie (Fredkin, 1960) directly from the set, (2) precomputes vocabulary-aware token masks at each trie node using Aho-Corasick multi-pattern matching (Aho & Corasick, 1975) to align BPE tokens with character-level trie paths, and (3) serves masks via $\mathcal{O}(1)$ cached lookups at decode time. The central algorithmic challenge is BPE-trie alignment: a single BPE token can span multiple trie nodes, and a vocabulary of 32K–262K tokens must be matched against every node. To our knowledge, this alignment problem has not been addressed in the constrained decoding literature; prior trie-based work (De Cao et al., 2021) sidesteps it by operating at token granularity. We show that it reduces to multi-pattern string matching, solvable in time linear in the trie size rather than quadratic in the vocabulary, enabling sub-100ms compilation up to $K = 10{,}000$. Figure 1 illustrates the cardinality wall and how the trie automaton overcomes it, expanding the practical limit from $\sim$1,000 to $\sim$100,000 values while maintaining 100% constraint compliance.

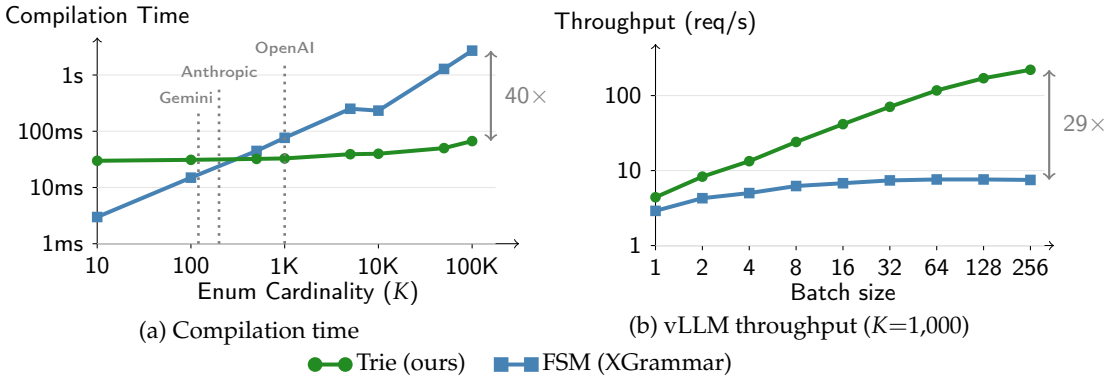
This work makes two main contributions: (1) We introduce the *trie automaton*, a specialized constrained decoding backend for finite-set constraints that combines character-level tries, Aho-Corasick multi-pattern matching, and precomputed token masks to achieve $\mathcal{O}(|\text{valid}[s_t]|)$ per-step masking (empirically 10–100 tokens after 3–4 characters of prefix, yielding effectively constant cost versus the $\mathcal{O}(V \cdot \ell)$ cost of general FSM approaches). The key insight is that the BPE-trie alignment problem reduces to multi-pattern string matching, yielding 2–6.5$\times$ faster compilation and up to 29$\times$ higher end-to-end throughput in batch serving: 7$\times$ from faster per-step masking, compounded by a simpler serving path that only precomputed masks can use. (2) We empirically characterize the cardinality wall across seven tokenizer families (32K–262K vocabulary), showing that matching enforcement mechanisms to constraint structure, including the integration path, overcomes scaling bottlenecks. Precomputed masks let the trie bypass the guided decoding pipeline entirely; FSM approaches cannot.

2 Background and Problem Formulation

Constrained decoding restricts the model's output distribution at each step t to tokens that can lead to valid completions. Given a vocabulary $\mathcal{V}$ of size V and a regular language $\mathcal{L}$ defined by a schema, the constrained distribution is:

$$p_c(y_t \mid \mathbf{y}_{<t}) = \frac{p(y_t \mid \mathbf{y}_{<t}) \cdot \mathbf{1}[y_t \in \mathcal{A}(\mathbf{y}_{<t})]}{Z(\mathbf{y}_{<t})} \tag{1}$$

where $\mathcal{A}(\mathbf{y}_{<t}) \subseteq \mathcal{V}$ is the set of allowed tokens given the prefix, computed by maintaining an FSM state $s_t = \delta^*(s_0, \text{chars}(\mathbf{y}_{<t}))$ (where δ^* extends the character-level transition function to token sequences and s_0 is the start state) and checking valid transitions; equivalently, $\mathcal{A}(\mathbf{y}_{<t}) = \mathcal{A}(s_t)$ depends only on the current state s_t, not the full prefix. An enum constraint $\mathcal{E} = \{e_1, e_2, \ldots, e_K\}$ defines the regular language $\mathcal{L}_\mathcal{E} = \{e_1\} \cup \{e_2\} \cup \cdots \cup \{e_K\}$. Let $L_{\max} = \max_i |e_i|$ be the maximum string length, ℓ the maximum token length in characters, and $|\Sigma|$ the alphabet size (256 for byte-level BPE tokenizers, which we refer to as "characters" throughout for readability; multi-byte UTF-8 sequences and emoji are handled naturally since both the trie and BPE tokenizers operate at byte granularity). The standard approach



(a) Compilation time

(b) vLLM throughput (K=1,000)

Figure 1: (a) Compilation time vs. enum cardinality (Qwen3-8B, log-log scale). Dotted lines mark documented provider limits (Gemini ∼120, Anthropic ∼200, OpenAI 1,000). The trie is flat at 30–67ms; XGrammar crosses the trie at K≈300 and reaches 2.7s at K=100K. (b) End-to-end vLLM throughput (log scale). The 29× gap at B=256 combines 7× per-step algorithmic advantage with integration-path savings from precomputed masks (Section 5).

converts the enum to a regular expression $e_1|e_2|\cdots|e_K$ and compiles it into a deterministic FSM. This creates problematic scaling: the DFA has $\mathcal{O}(K \cdot L_{\max})$ states in the worst case, per-step masking costs $\mathcal{O}(V \cdot \ell)$ (checking each token's character sequence against the FSM), and compilation costs $\mathcal{O}(K \cdot L_{\max} \cdot |\Sigma|)$, creating the cardinality wall observed in practice (detailed analysis in Appendices A and G).

To illustrate concretely, consider an agentic system routing requests to the correct tool from a registry of 2,000 APIs (Fei et al., 2025; Gaurav et al., 2025). FSM compilation requires processing 15.4 million character-level transitions (25–50 seconds), while per-step masking requires ∼1.3 million effective FSM operations per tool selection when accounting for cache effects (Appendix F), making the system unusable for interactive workflows that demand sub-second tool dispatch.

3 Related Work

Constrained decoding. Early approaches enforced specific constraint types: lexically constrained beam search (Post & Vilar, 2018; Anderson et al., 2017), predicate logic constraints (Lu et al., 2021; 2022), and incremental parsing for code generation (Scholak et al., 2021). Outlines (Willard & Louf, 2023) introduced the dominant paradigm of compiling JSON schemas into FSMs for token masking, extended to context-free grammars by Geng et al. (2023) and formalized by Koo et al. (2024). LMQL (Beurer-Kellner et al., 2023) embedded constraints into a query language. Subsequent work optimized within this paradigm: SGLang (Zheng et al., 2024) introduced jump-forward decoding, XGrammar (Dong et al., 2024) optimized vocabulary partitioning, and SynCode (Ugare et al., 2024) and Wang et al. (2025) continued this trajectory. LLGuidance (Geng et al., 2025) takes a different approach: an Earley parser with lazy automaton construction that avoids upfront DFA compilation. On Qwen3-8B (151K vocabulary), LLGuidance compiles enum schemas in 0.6–24ms for $K = 10$–10,000, far faster than XGrammar's 5–695ms. However, its per-step cost remains $\mathcal{O}(V)$: we measure 73–141μs per mask computation, compared to our trie's 0.65μs (110–215× slower; Appendix M). The two are complementary: LLGuidance excels at schema diversity with negligible startup, while the trie exploits finite-set structure for per-step speedups that compound in batch serving. Despite these advances, persistent enum limits across major providers indicate that no current system adequately addresses the cardinality wall.

Trie-based generation. GENRE (De Cao et al., 2021) demonstrated trie-constrained generation for entity linking with a fine-tuned seq2seq model. GENRE builds a *token-level* trie where nodes are pre-tokenized token IDs, elegantly avoiding the vocabulary-trie alignment

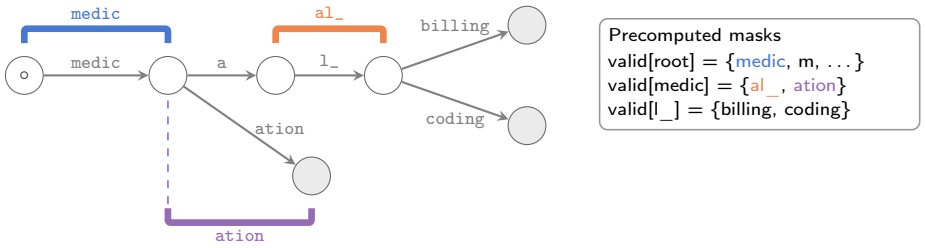


Figure 2: The BPE-trie alignment problem. A character-level trie encodes three enum values with shared prefix structure. BPE tokens (colored brackets) span multiple trie edges: `medic` traverses 5 character nodes from the root. At each node, we precompute the set of valid BPE tokens (right), enabling precomputed mask lookups at decode time.

problem. However, prefix sharing operates at token granularity (two strings sharing a 10-character prefix may share only one token-level node), and the trie is tokenizer-specific. Our trie automaton builds a *character-level* trie that maximizes prefix sharing regardless of tokenizer, solving the BPE alignment problem via Aho-Corasick multi-pattern matching (Aho & Corasick, 1975) to determine which BPE tokens from a vocabulary of 32K–262K entries are valid continuations at each node (Section 4.2). Concurrent work by Su et al. (2026) flattens token-level prefix trees into CSR sparse matrices for vectorized TPU/GPU execution; like GENRE, it operates on small semantic ID vocabularies ($|\mathcal{V}| \approx 2{,}048$) where the BPE alignment problem does not arise.

4 Methods

4.1 Character-Level Trie with Precomputed Masks

Given an enum $\mathcal{E} = \{e_1, \ldots, e_K\}$, we build a character-level trie $\mathcal{T}$ by inserting each string character-by-character. The trie (Fredkin, 1960) merges shared prefixes: if many tool names start with `get_`, they share a single prefix path rather than duplicating it for each value. This reduces the number of nodes from $\mathcal{O}(K \cdot L_{\max})$ (as in the equivalent DFA) to $\mathcal{O}(N_{\text{chars}})$, where $N_{\text{chars}} = \sum_{i=1}^{K} |e_i|$ is the total character count. For each trie node n, we precompute a set $\text{valid}[n] \subseteq \mathcal{V}$ of vocabulary tokens that are valid continuations from n. At decode time, masking reduces to a direct lookup: $\mathcal{A}(s_t) = \text{valid}[s_t]$, transforming per-step cost from $\mathcal{O}(V \cdot \ell)$ (scanning all tokens against the FSM) to $\mathcal{O}(|\text{valid}[s_t]|)$ (iterating the precomputed set).

The challenge is computing $\text{valid}[n]$ efficiently. A token v is valid at node n if its character sequence traces a path starting at n that stays within the trie, but BPE tokens are multi-character, so a single token can traverse several trie edges. For example, given enum values `medical_billing` and `medical_coding`, the token "`medical`" (7 characters) is valid at the root because it traces the path through nodes `m`$\to$`e`$\to \cdots \to$`l`, while the token "`_bill`" is valid at the node after `l` because it continues along the `_billing` branch (Figure 2). A naive approach checks every token at every node by simulating the character walk, costing $\mathcal{O}(N_{\text{chars}} \cdot V \cdot \ell)$, which is prohibitive for vocabularies of 32K–262K tokens.

4.2 Efficient Mask Precomputation via Multi-Pattern Matching

We solve the precomputation problem by reducing it to *multi-pattern string matching*: given a set of patterns (the vocabulary token strings) and a text (each root-to-leaf path in the trie), find all positions where any pattern occurs. The Aho-Corasick (AC) algorithm (Aho & Corasick, 1975) solves this classic problem by building a finite automaton from the pattern set in $\mathcal{O}(V \cdot \ell)$ time, where ℓ is the maximum token length in characters, then processing any input text in a single linear pass, reporting all pattern matches as they are encountered. Crucially, the processing cost is $\mathcal{O}(L + m)$ for a text of length L with m reported matches, independent of the number of patterns V. Since at each of the L character positions at most

ℓ tokens of different lengths can match, $m \leq L \cdot \ell$, so scanning a trie path of length L costs $\mathcal{O}(L \cdot \ell)$ rather than $\mathcal{O}(L \cdot V \cdot \ell)$. We apply this as follows:

1. **Build AC automaton from vocabulary.** Insert each vocabulary token's character string as a pattern. This constructs the automaton in $\mathcal{O}(V \cdot \ell)$ time.
2. **Traverse the trie with the automaton.** Before any decoding begins, we perform a depth-first traversal of the trie, maintaining the AC automaton state across edges. The AC state is a single integer (current node index), so save/restore at branch points is trivial (push/pop one integer per DFS stack frame), and each edge is processed exactly once. At each trie node n, the automaton identifies which vocabulary tokens have a character string that starts at n and follows a valid path in the trie. A matched token v is added to valid$[n]$ if its characters trace a path from n that either reaches a leaf (completing an enum value) or lands on an internal node (allowing continuation by subsequent tokens). Tokens that would "overshoot" a leaf, i.e., whose characters extend beyond the end of an enum value, are excluded. When an enum value is a proper prefix of another (e.g., `"get"` and `"get_user"`), the trie marks the shorter value's terminal node as both a leaf (where EOS is valid) and an internal node (where continuation tokens are valid), ensuring both values are reachable. This is done once per schema and cached.

This reduces precomputation from $\mathcal{O}(N_{\text{chars}} \cdot V \cdot \ell)$ to $\mathcal{O}((N_{\text{chars}} + V) \cdot \ell)$, a factor of V improvement. In practice, the AC automaton construction ($\mathcal{O}(V \cdot \ell)$) dominates the total compilation cost because $V \gg N_{\text{chars}}$ for typical enum sizes: with $V = 151$K and $\ell = 4$, the AC construction requires $\sim$600K operations regardless of K, while the trie traversal adds only $N_{\text{chars}} \cdot \ell$ operations (e.g., 240K at $K = 2{,}000$). This explains the empirically observed near-flat compilation time (30–40ms for Qwen3-8B across $K = 10$–$10{,}000$; 67ms at $K = 100{,}000$): the cost is dominated by the K-independent AC construction over the vocabulary. Since the AC automaton depends only on the tokenizer (not the enum set), it can be built once per tokenizer and reused across all enum schemas, reducing per-schema compilation to just the trie traversal: $\mathcal{O}(N_{\text{chars}} \cdot \ell)$. At decode time, masking is a direct lookup: $\mathcal{A}(s_t) = \text{valid}[s_t]$, costing $\mathcal{O}(|\text{valid}[s_t]|)$ instead of $\mathcal{O}(V \cdot \ell)$. Note that applying the mask to the logits vector is $\mathcal{O}(V)$ regardless of method (setting invalid logits to $-\infty$); the savings is in *computing* which tokens are valid, not in applying the result. Since $|\text{valid}[s_t]|$ shrinks exponentially with trie depth (after 3–4 characters of prefix, typically 10–100 tokens remain valid), the lookup cost is effectively constant at 0.65μs regardless of K or vocabulary size. While the worst case is $\mathcal{O}(V)$ (at the root node), the rapid prefix-driven shrinkage means the practical per-step cost is orders of magnitude below FSM approaches, which must scan all V tokens regardless of constraint structure.

The trie automaton produces *identical outputs* to FSM-based constrained decoding: the trie for a finite set $\mathcal{E}$ is isomorphic to the minimal DFA for $\mathcal{L}_{\mathcal{E}}$ (by the Myhill-Nerode theorem), so greedy decoding and fixed-seed sampling yield the same results under both methods (Appendix G).

Unlike GENRE's token-level trie (Section 3), our AC-based approach works with any tokenizer as a drop-in backend. The compilation speedup depends on prefix structure: for tool names and hierarchical codes with prefix sharing ratio $r = |\mathcal{T}(\mathcal{E})|/N_{\text{chars}} < 0.8$, compilation is 2–6.5$\times$ faster; for random identifiers ($r \approx 1$), the compilation advantage diminishes but per-step masking remains 7$\times$ faster (Appendix H).

4.3 Extensions and Integration

The trie automaton is designed as a *drop-in component* for mixed schemas: the system routes finite-set constraints to the trie while delegating structural constraints to the standard FSM/PDA backend, requiring no changes to user schemas. For $K > 50{,}000$, we sketch two preliminary extensions in Appendices I–J for future directions: hierarchical schema rewriting ($\mathcal{O}(\sqrt{K})$ effective per-step cardinality) and speculative short-circuiting.

5 Experiments

We evaluate the trie automaton on latency, compilation time, and accuracy/validity across various open-sourced model series on NVIDIA A100 GPUs. The primary comparison is against xgrammar (Dong et al., 2024), the backend used by vLLM and SGLang. The trie automaton is implemented in Rust with Python bindings via PyO3; XGrammar is implemented in C++ with Python bindings. Full experimental details are in Appendix C.

5.1 Latency and Scalability

Table 3 shows the performance breakdown across $K \in \{10, 100, 1{,}000, 10{,}000\}$ for XGrammar (Dong et al., 2024), LLGuidance (Geng et al., 2025), and our trie. All benchmarks run on NVIDIA A100 GPUs (80GB) with AMD EPYC 7R32 CPUs (96 cores).

Three approaches occupy distinct points in the compilation-vs-masking tradeoff (Table 19). LLGuidance achieves near-zero compilation (0.6–24ms) but per-step masking costs 73–141μs. XGrammar balances both (3–239ms compilation, 5–10μs masking). The trie minimizes per-step cost (0.65μs) through precomputation, with nearly-flat compilation (30–40ms). For one-shot dynamic schemas where compilation dominates (e.g., retrieval-augmented settings with per-query enum sets at $K < 500$), LLGuidance's 1–3ms compilation may be preferable despite higher per-step cost; the trie's advantage is decisive when per-step cost dominates, particularly in batch serving (Appendix L).

The trie's advantage is decisive in batch serving (Table 2), where the GPU forward pass is shared across B requests but masking runs per-request on CPU. At $B = 128$, LLGuidance masking (3.7ms) consumes 37% of the GPU forward pass, becoming the throughput bottleneck; XGrammar's 783μs is 7.8%; the trie's 10μs is negligible (0.1%). We validate this with end-to-end vLLM throughput at $K = 1{,}000$ (Table 1): at $B = 256$, the trie achieves 219 req/s vs. XGrammar's 7.5 req/s (29×). The trie also exceeds unconstrained throughput (219 vs. 104 req/s) because constrained decoding terminates at trie leaf nodes (3.2 tokens/request vs. 8.7 unconstrained), reducing GPU forward passes. Both constrained methods generate the same number of tokens, so the trie-vs-XGrammar ratio isolates the masking and integration differences.

The 29× gap compounds two effects. First, the per-step algorithmic advantage: precomputed mask lookup costs 0.65μs vs. XGrammar's 5.9μs dynamic computation (∼7×; Table 3). Second, the integration path: because the trie's masks are precomputed, it integrates as a stateless `LogitsProcessor` that returns a cached bitmask per step. XGrammar does not currently support this path; its architecture requires dynamic mask computation via vLLM's guided decoding pipeline with per-request grammar compilation, sequential FSM state management, and scheduling overhead. In principle, XGrammar *could* precompute and cache per-state masks for enum constraints, but doing so would effectively reconstruct the trie: the minimal DFA for a finite set is isomorphic to the trie (Proposition 1), so caching its per-state masks yields the same data structure. The trie is thus the natural endpoint of optimizing FSM-based decoding for finite sets.

Deployment implications. The batch serving results have direct consequences for GPU utilization in production. At $B = 128$, each decoding step comprises a GPU forward pass (∼10ms) followed by CPU mask computation. With XGrammar, masking takes 783 μs, where 7.8% of the step is spent with the GPU idle waiting for the mask. With LLGuidance (3.7 ms), this rises to 27%, making masking the primary bottleneck. The trie reduces idle time to 0.1% of the step, keeping the GPU saturated. For dynamic enum constraints (e.g., retrieval-augmented tool selection where the valid set changes per query), the trie's 33–40 ms compilation is fast enough to run on-the-fly without impacting serving latency, whereas XGrammar's 75–239 ms compilation at $K \geq 1{,}000$ adds perceptible delay. The AC automaton can be cached per tokenizer and shared across all enum schemas, so only the trie traversal ($\mathcal{O}(N_{\text{chars}} \cdot \ell)$, typically < 5 ms) runs per schema change.

Memory and deployment. The trie's memory scales modestly (Table 4): 0.9 MB at $K = 10{,}000$, 8 MB at $K = 100{,}000$, vs. ∼2 GB for the FSM (Appendix H). The AC au-

Table 1: End-to-end vLLM throughput (req/s) at $K = 1{,}000$. The trie integrates as a stateless `LogitsProcessor` (precomputed masks); XGrammar requires vLLM's guided decoding pipeline (dynamic mask computation). Trie exceeds unconstrained throughput due to early termination at trie leaves (3.2 vs. 8.7 tokens/request); both constrained methods generate 3.2 tokens/request (verified), so the trie-vs-XGrammar ratio isolates the masking and integration differences. Qwen3-8B, A100, greedy, median of 3 runs.

B	Trie	XGrammar	Uncon.	Trie/XG
1	4.4	2.9	1.9	1.5×
2	8.3	4.3	3.6	1.9×
4	13.4	5.0	7.2	2.7×
8	23.9	6.2	13.6	3.9×
16	41.8	6.8	24.8	6.1×
32	70.4	7.4	40.4	9.6×
64	116.5	7.6	65.4	15.2×
128	170.5	7.6	80.3	22.4×
256	219.4	7.5	103.8	29.3×

Table 2: Batch masking cost (μs per batch-step) at $K = 1{,}000$ (Qwen3-8B, 151K vocab). GPU forward pass ∼10ms.

B	Trie (% fwd)	XGrammar (% fwd)	LLGuidance (% fwd)
1	0.2 (0.0%)	5 (0.1%)	31 (0.3%)
32	2 (0.0%)	170 (1.7%)	893 (8.9%)
128	10 (0.1%)	783 (7.8%)	3,716 (37%)

Table 3: Compilation time and per-step masking cost by K (Qwen3-8B, 151K vocab). Per-step measures mask *computation* only (determining valid tokens), not the $\mathcal{O}(V)$ bitmask application to logits (∼31μs via tensor operation, identical for all methods). XGrammar's non-monotonic per-step cost (9.5μs at K=10, 5.4μs at K=100) reflects its vocabulary partitioning: at small K, fewer tokens fall in the "adaptive" partition, causing more cache misses; the partition stabilizes at $K \geq 100$. Mean±std, 10 runs.

	Method	K=10	K=100	K=1K	K=10K
Compile (ms)	Trie	30±2	31±1	*33*±1	*40*±2
	XGrammar	*3*±1	*15*±2	75±3	239±10
	LLGuidance	**1**±0	**1**±0	**3**±0	**24**±1
Mask (μs/step)	Trie	**0.65**	**0.65**	**0.65**	**0.65**
	XGrammar	*9.5*	*5.4*	*5.8*	*5.9*
	LLGuidance	121	73	85	141

tomaton requires ∼150 MB for the largest vocabulary (Gemma3 262K), amortized across all enum schemas sharing that tokenizer. The precomputed masks are immutable after construction, so concurrent read access from multiple serving threads requires no synchronization. Integrated with vLLM as a `LogitsProcessor`, the trie maintains sub-100ms per-example latency at $K = 10{,}000$ (Table 13, Appendix D).

Table 4: Memory footprint of precomputed trie masks.

K	Nodes	Valid entries	Est. memory
100	2,041	5,065	20 KB
1,000	13,341	30,759	120 KB
10,000	103,266	227,762	0.9 MB
100,000	907,042	2,094,894	8.0 MB

Mixed schemas. To validate the trie as a drop-in component, we measure compilation for three mixed schemas (enum + structural fields): tool-calling ($K = 1{,}000$), classification ($K = 500$), and entity-linking ($K = 5{,}000$). The system routes enum fields to the trie and structural fields to XGrammar (<1ms dispatch overhead). At $K = 5{,}000$: 37ms vs. 150ms; at $K = 1{,}000$: 33ms vs. 75ms. For nested enums, the PDA backend handles structural navigation and transfers control to the trie at enum-valued fields. Multiple enum fields in a single schema each get their own trie. The principle extends to non-enum constraints (Appendix K).

5.2 Generalization

To verify these results generalize, we evaluate across seven model families (32K–262K vocabulary; Table 5). The trie achieves consistent compilation speedups at $K \geq 1{,}000$: 1.2–6.4×, increasing to 3.5–13.7× at $K = 5{,}000$–$10{,}000$. Smaller vocabularies amplify the advantage because AC construction is $\mathcal{O}(V \cdot \ell)$, so smaller V yields faster precomputation. Conversely, for very large vocabularies (Gemma3 262K), the K-independent AC overhead reduces the speedup to 1.2× at $K = 1{,}000$; as vocabulary sizes trend upward, the compilation crossover shifts to higher K. Per-step cost is unaffected by vocabulary size: Mistral-32K 0.60μs, GPT2-50K 0.62μs, OLMo-100K 0.63μs, Mistral Small-131K 0.64μs, Qwen3-151K 0.65μs, gpt-oss-200K 0.66μs, Gemma3-262K 0.67μs.

Table 5: Compilation speedup (trie vs. xgrammar) across model families. Values >1× indicate trie is faster. Per-step masking is 0.60–0.67μs for all configurations. Trie compilation is nearly flat: 11–96ms across all K and tokenizers.

Model	Vocab	K=100	K=1K	K=5K	K=10K
Mistral 7B v0.3	32K	1.5×	6.4×	11.6×	11.5×
GPT-2	50K	1.1×	4.9×	13.7×	12.0×
OLMo 3 7B	100K	0.6×	2.1×	6.2×	5.2×
Mistral Small 3.1	131K	0.5×	2.3×	6.6×	5.5×
Qwen3-8B	151K	0.4×	1.8×	6.7×	4.4×
OpenAI gpt-oss-20b	200K	0.3×	1.4×	4.7×	3.5×
Gemma3 12B	262K	0.3×	1.2×	3.6×	3.5×

5.3 Accuracy and Validity

The trie produces identical outputs to FSM-based constrained decoding (Proposition 1, verified on 1,000 samples). Its contribution is not improving accuracy at any given K, but *enabling* constrained decoding where FSM compilation is impractical. We evaluate on classification benchmarks where ground-truth accuracy can be measured; existing tool-calling benchmarks evaluate multi-step chains rather than flat tool selection, making them unsuitable for isolating the constrained decoding bottleneck. We compare: single-pass unconstrained, single-pass + trie (same prompt), think unconstrained, and think + trie (prompts in Appendix C.1).

Across all 24 model×dataset settings (Table 7), constrained decoding achieves the highest accuracy in 21/24 cases while guaranteeing 100% validity; unconstrained decoding reaches ≥95% validity in only 8/24. The trie acts as a no-cost safety net for strong models (Gemma3 12B: matches unconstrained accuracy while eliminating the 0.2–0.5% failure rate) and a critical enabler for weaker ones (Qwen3-1.7B Banking77: 24.8% with trie vs. 4.8% unconstrained at 6.6% validity). The think-then-answer strategy is particularly effective with the trie: on Qwen3-8B, think+trie achieves the best accuracy in all four datasets, with gains of up to 21.5 points over single-pass unconstrained (TREC: 63.1 vs. 36.3). Without the trie, think+unconstrained often *degrades* accuracy relative to single-pass (Qwen3-8B Banking77: 37.6 vs. 48.8) because the reasoning phase produces outputs that are harder to parse into valid labels. The three cases where unconstrained decoding wins (Gemma3 MASSIVE, Gemma3 CLINC150, gpt-oss CLINC150) all involve strong models on datasets where PE

Table 6: Accuracy (%) and validity (%) on classification benchmarks (5 runs, mean±std, greedy). Prompts identical within each pair; trie is the only difference. Trie guarantees 100% validity. Full results (6 models) in Appendix Table 7.

			Single-pass			Think-then-answer	
			Uncon.		+ Trie	Uncon.	+ Trie
Model	Dataset	K	Acc	Val	Acc (100%)	Acc (val)	Acc (100%)
Qwen3-8B	TREC	42	36.3±1.2	81.2	*41.6*±1.8	37.9±1.1 (58.3)	**63.1**±1.6
	MASSIVE	59	74.6±1.4	99.0	74.6±1.7	*74.9*±2.4 (97.2)	**76.1**±2.2
	Banking77	76	48.8±5.2	78.2	*56.8*±4.4	37.6±2.2 (69.2)	**61.0**±3.6
	CLINC150	150	*72.4*±2.7	87.0	63.0±1.8	61.0±4.6 (79.6)	**73.8**±3.7
gpt-oss-20b	TREC	42	*67.5*±1.0	87.6	**72.9**±1.7	60.7±2.6 (86.5)	60.4±4.1
	MASSIVE	59	*76.8*±2.5	91.9	**79.3**±1.0	61.8±3.7 (75.4)	76.1±1.7
	Banking77	76	78.2±1.2	90.2	**81.3**±1.2	79.8±1.6 (96.2)	*81.1*±1.9
	CLINC150	150	**79.7**±2.3	89.2	*77.0*±1.6	71.4±3.2 (85.5)	74.0±1.8

validity already exceeds 89%; even here, the accuracy gap is small (1–5 points) while the validity gap remains (89–100% vs. 100%).

At $K = 500$–5,000, the trie guarantees 100% validity while unconstrained drops to 84–98% (Table 17). Practitioner guidance in Appendix L.

Limitations. The trie automaton is specialized for flat finite-set constraints. Production schemas that combine enum fields with nested objects or arrays still require a general-purpose backend for the structural portions; our mixed-schema evaluation validates compilation time for such cases but not end-to-end serving throughput. The per-step speedup is most impactful in batch serving: for a single request, the $\mathcal{O}(V)$ bitmask application ($\sim$31μs) is method-independent and dominates the valid-token computation where the trie excels (0.65μs vs. 5.8μs; Appendix H.3), but at batch size B the computation runs B times and becomes the bottleneck that the trie eliminates. For dynamic one-shot schemas at small K, LLGuidance's near-zero compilation (1–3ms) may outweigh the trie's per-step advantage. At extreme cardinalities ($K > 1{,}000$), the trie guarantees validity where FSM compilation times out, but classification accuracy at such scale remains limited by model capability rather than by the decoding mechanism.

6 Conclusion

This work addresses a mismatch in constrained decoding: applying general-purpose automata to constraints with exploitable structure. We solve it for finite-set constraints using Aho-Corasick precomputation, expanding the practical enum limit from hundreds to tens of thousands. The trie achieves 7× faster per-step masking through precomputed lookups; because precomputed masks also enable a stateless serving path that bypasses the guided decoding pipeline, the advantage compounds to 29× higher end-to-end batch throughput. Across seven tokenizer families and six models, the trie maintains sub-100ms compilation and flat per-step cost while guaranteeing 100% output validity. A controlled comparison confirms the trie is the efficient algorithm for this precomputation: the equivalent FSM-based approach produces an isomorphic data structure 196× slower (Appendix H.2). The underlying principle, matching enforcement mechanisms to constraint structure, generalizes beyond enums to any constraint with exploitable regularity (Appendix K). As structured generation becomes central to agentic systems, we expect constraint-specialized backends to become the norm rather than the exception. Extending the dispatch principle to other structured patterns, including date/time formats (Appendix K), numeric ranges, and regex subclasses, is a natural next step.

References

Alfred V Aho and Margaret J Corasick. Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.

Peter Anderson, Basura Fernando, Mark Johnson, and Stephen Gould. Guided open vocabulary image captioning with constrained beam search. In *EMNLP*, 2017.

Anthropic. Structured outputs. `https://docs.anthropic.com/en/docs/build-with-claude/structured-outputs`, 2025. 180-second compilation timeout. Accessed: 2026-03-09.

Luca Beurer-Kellner, Marc Fischer, and Martin Vechev. Prompting is programming: A query language for large language models. *Proc. ACM Program. Lang.*, 7(PLDI), June 2023. doi: 10.1145/3591300. URL `https://doi.org/10.1145/3591300`.

Nicola De Cao, Gautier Izacard, Sebastian Riedel, and Fabio Petroni. Autoregressive entity retrieval. In *ICLR*, 2021. arXiv:2010.00904.

Yixin Dong, Charlie F. Ruan, Yaxing Cai, Ruihang Lai, Ziyi Xu, Yilong Zhao, and Tianqi Chen. XGrammar: Flexible and efficient structured generation engine for large language models. *arXiv preprint arXiv:2411.15100*, 2024.

Xiang Fei, Xiawu Zheng, and Hao Feng. MCP-Zero: Active tool discovery for autonomous LLM agents. *arXiv preprint arXiv:2506.01056*, 2025.

Edward Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, 1960.

Nishant Gaurav, Adit Akarsh, Ankit Ranjan, and Manoj Bajaj. Dynamic react: Scalable tool selection for large-scale mcp environmentss. *arXiv preprint arXiv:2509.20386*, 2025.

Saibo Geng, Martin Josifoski, Maxime Peyrard, and Robert West. Grammar-constrained decoding for structured NLP tasks without finetuning. In *EMNLP*, 2023. arXiv:2305.13971.

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. *arXiv preprint arXiv:2501.10868*, 2025.

Google. Structured output. `https://ai.google.dev/gemini-api/docs/structured-output`, 2024. Accessed: 2026-02-01.

John Hopcroft. An $n \log n$ algorithm for minimizing states in a finite automaton. *Theory of Machines and Computations*, pp. 189–196, 1971.

John E Hopcroft, Rajeev Motwani, 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 *NeurIPS*, 2024. arXiv:2407.08103.

Ximing Lu, Peter West, Rowan Zellers, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. NeuroLogic decoding: (un)supervised neural text generation with predicate logic constraints. In *NAACL*, 2021. arXiv:2010.12884.

Ximing Lu, Sean Welleck, Peter West, Liwei Jiang, Jungo Kasai, Daniel Khashabi, Ronan Le Bras, Lianhui Qin, Youngjae Yu, Rowan Zellers, Noah A. Smith, and Yejin Choi. Neurologic a*esque decoding: Constrained text generation with lookahead heuristics. In *NAACL*, 2022. arXiv:2112.08726.

Thanh-Tung Nguyen, Viktor Schlegel, Abhinav Kashyap, Stefan Winkler, Shao-Syuan Huang, Jie-Jyun Liu, and Chih-Jen Lin. Mimic-iv-icd: A new benchmark for extreme multilabel classification. *arXiv preprint arXiv:2304.13998*, 2023.

OpenAI. Structured model outputs. https://platform.openai.com/docs/guides/structured-outputs, 2024. Accessed: 2026-02-01.

Matt Post and David Vilar. Fast lexically constrained decoding with dynamic beam allocation for neural machine translation. In *NAACL*, 2018. arXiv:1804.06609.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. ToolLLM: Facilitating large language models to master 16000+ real-world APIs. In *ICLR*, 2024. arXiv:2307.16789.

Torsten Scholak, Nathan Schucher, and Dzmitry Bahdanau. Picard: Parsing incrementally for constrained auto-regressive decoding from language models. In *EMNLP*, 2021. arXiv:2109.05093.

Zhengyang Su, Isay Katsman, Yueqi Wang, Ruining He, Lukasz Heldt, Raghunandan Keshavan, Shao-Chuan Wang, Xinyang Yi, Mingyan Gao, Onkar Dalal, Lichan Hong, Ed Chi, and Ningren Han. Vectorizing the trie: Efficient constrained decoding for LLM-based generative retrieval on accelerators. *arXiv preprint arXiv:2602.22647*, 2026.

Ken Thompson. Programming techniques: Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422, 1968.

Shubham Ugare, Tarun Suresh, Hangoo Kang, Sasa Misailovic, and Gagandeep Singh. Syncode: Llm generation with grammar augmentation. *Trans. Mach. Learn. Res.*, 2025, 2024. URL https://api.semanticscholar.org/CorpusID:268248075.

Ran Wang, Xiaoxuan Liu, Hao Ren, Gang Chen, Fanchao Qi, and Maosong Sun. Wgrammar: Leverage prior knowledge to accelerate structured decoding. *arXiv preprint arXiv:2507.16768*, 2025.

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

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. SGLang: Efficient execution of structured language model programs. In *NeurIPS*, 2024.

A Detailed Background

The key bottlenecks in FSM-based constrained decoding for enums are: (1) state space explosion ($\mathcal{O}(K \cdot L_{\max})$ DFA states), (2) per-step masking cost ($\mathcal{O}(V \cdot \ell)$ with cache penalties at large K), and (3) compilation cost ($\tilde{\mathcal{O}}(K \cdot L_{\max} \cdot |\Sigma|)$). These compound super-linearly: doubling K more than doubles end-to-end latency because the cache penalty $f(|\mathcal{S}|)$ is itself increasing in K. Dynamic schemas (tool registries, retrieval-augmented selection, multi-tenant serving) prevent amortization, making compilation latency the dominant cost.

B Full Accuracy and Validity Results

Table 7 presents the complete accuracy and validity results across all six models and four datasets. The main body (Table 6) shows three representative models; this table adds Qwen3-1.7B (small model where constrained decoding is critical), Gemma3 12B (strong model where unconstrained decoding nearly suffices), and Mistral 7B v0.3 (mid-range instruction-tuned model).

Table 7: Accuracy (%) and validity (%) results across all three additional models (5 runs, mean±std, greedy decoding). Extends Table 6 with three additional models. **Bold**: best accuracy per row; *italic*: second best.

			Single-pass			Think-then-answer	
			Unconstrained		+ Trie	Unconstrained	+ Trie
Model	Dataset	K	Acc	Val	Acc (100% val)	Acc (val)	Acc (100% val)
Qwen3-1.7B	TREC	42	23.3±2.8	96.4	23.7±2.6	*33.4*±2.1 (83.8)	**34.5**±1.8
	MASSIVE	59	53.4±2.3	97.5	53.8±2.0	*55.0*±2.4 (86.2)	**56.3**±2.0
	Banking77	76	4.8±1.2	6.6	24.8±1.7	*29.6*±4.9 (77.0)	**37.8**±6.6
	CLINC150	150	8.8±2.0	10.4	32.8±3.2	*36.0*±3.1 (69.2)	**47.0**±2.6
Gemma3 12B	TREC	42	*63.6*±1.0	99.5	**63.7**±1.1	22.9±2.3 (36.6)	58.2±1.7
	MASSIVE	59	**77.3**±2.4	99.6	*76.8*±2.2	75.4±2.0 (98.8)	75.1±2.3
	Banking77	76	*71.6*±6.7	99.8	*71.6*±6.7	70.2±7.8 (98.0)	**72.0**±7.0
	CLINC150	150	**87.6**±1.7	99.6	83.0±2.2	*86.0*±2.3 (100)	80.8±2.8
Mistral 7B v0.3	TREC	42	23.0±1.5	42.5	**37.6**±1.5	16.4±1.7 (33.5)	*32.6*±1.1
	MASSIVE	59	65.7±3.9	86.3	**69.5**±2.7	52.8±2.6 (68.3)	*67.1*±1.4
	Banking77	76	*48.4*±6.3	78.4	**53.4**±6.1	31.2±5.0 (48.0)	47.0±5.5
	CLINC150	150	*64.4*±2.2	77.4	**65.4**±3.8	46.0±2.2 (57.8)	59.4±1.6

C Experimental Setup Details

We implement the trie automaton in Rust with Python bindings via PyO3, using the `aho-corasick` crate (v1.1) for multi-pattern matching and parallel precomputation across CPU cores (using all available cores via Rayon; XGrammar's C++ backend is also multi-threaded). Both are compiled languages with comparable performance characteristics; the per-step advantage (precomputed lookup vs. dynamic computation) is algorithmic. Our primary comparison is against xgrammar v0.1.11 (Dong et al., 2024) and llguidance v1.6.1 (Geng et al., 2025). We verified that XGrammar's `compile_json_schema` with an enum schema produces equivalent compilation times to `compile_regex` with a union pattern (within 5% across all K), confirming that `compile_regex` is a fair baseline. We also evaluate several alternative approaches: naive FSM-based constrained decoding, unconstrained generation with retry (up to 3 attempts), unconstrained generation with post-hoc string similarity matching, and prompt engineering that includes enum values directly in the prompt. All timing benchmarks run on NVIDIA A100 GPUs (80GB); CPU timing on AMD EPYC 7R32 (96 cores, 3.3 GHz). Per-step masking times are measured with 200 warm-up iterations followed by 1,000 timed iterations (same warm-up for all three methods). Compilation and per-step results report mean±std over 10 runs; end-to-end vLLM throughput reports median of 3 runs.

C.1 Prompt Templates

For the accuracy and validity experiments (Section 5), we use the following prompt templates. Let `options` denote the comma-separated enum values and `text` denote the input.

PE and Trie Strict. Both methods use the same prompt; the only difference is whether trie-constrained decoding is applied. This enables a controlled comparison isolating the effect of constrained decoding from prompt wording.

```
Select exactly one of the following options.
Output ONLY the option itself, nothing else.

Options: {options}

Text: {text}
Answer:
```

Trie. A shorter prompt relying on the trie constraint to enforce validity:

```
Select one of: {options}

Text: {text}
Answer:
```

Think+PE and Think+Trie. Phase 1 generates reasoning (unconstrained, up to 300 tokens), and phase 2 generates the result.

Phase 1:

```
Options: {options}

Text: {text}
Let me think step by step:
```

Phase 2:

```
Select exactly one of the following options.
Output ONLY the option itself, nothing else.

Options: {options}

Text: {text}
Thinking: {phase_1_output}
Answer:
```

Summary of prompt controls. PE vs. +Trie is a clean comparison: identical prompts, differing only in decoding strategy. Think+PE vs. Think+Trie share the same prompts; the only difference is whether the Phase 2 answer is trie-constrained. For unconstrained methods, we extract the model's output and fuzzy-match against the enum set. For trie methods, the output is guaranteed valid by construction.

D Detailed Experimental Results

D.1 Main Results

Table 8: End-to-end latency (seconds) for enum generation by cardinality K. Timeout threshold is 60s.

Method	K=10	K=50	K=100	K=500	K=1000	K=5000	K=10000
Outlines/SGLang/XGrammar	0.05	0.10	0.15	0.48	1.20	2.93	5.88
Unconstrained + retry	5.85	7.59	5.87	7.60	5.90	5.90	6.04
Unconstrained + post-hoc	1.94	2.53	1.97	2.58	2.05	2.44	2.92
Prompt engineering	2.01	2.55	2.04	2.71	2.18	2.16	2.16
Trie Automaton	0.72	0.72	0.72	0.72	0.77	0.77	0.77
Hierarchical Rewriting	5.02	5.06	3.92	5.10	3.97	3.99	4.13
Speculative (k=20)	2.55	2.22	1.98	2.57	1.99	2.00	2.01

Table 9: Schema compliance (%) for different approaches. Constrained methods achieve 100% by design.

Method	K=10	K=50	K=100	K=500	K=1000	K=5000	K=10000
Outlines/SGLang/XGrammar	100	100	100	100	100	100	100
Unconstrained + retry[2]	0	0	0	0	0	0	0
Unconstrained + post-hoc	100	100	100	100	100	100	100
Prompt engineering	100	100	100	100	100	100	100
Trie Automaton	100	100	100	100	100	100	100
Hierarchical Rewriting	100	100	100	100	100	100	100
Speculative (k=20)	100	100	100	100	100	100	100
Combined (Ours)[3]	100	100	100	100	100	100	100

Table 10: Memory footprint of precomputed trie masks (full results).

K	Nodes	Valid entries	Est. memory
100	2,041	5,065	20 KB
1,000	13,341	30,759	120 KB
5,000	54,426	119,740	0.5 MB
10,000	103,266	227,762	0.9 MB
50,000	470,823	1,068,510	4.1 MB
100,000	907,042	2,094,894	8.0 MB

Table 11: Real-world task results at extreme cardinalities. Accuracy is classification accuracy; latency is median per-example. The 0% accuracy at K=68,000 reflects model capability limitations at extreme cardinality, not a trie limitation; the trie enables the attempt where FSM compilation times out.

	Product Catalog (K=1500)		ICD-10 (K=68000)	
Method	Accuracy	Latency (s)	Accuracy	Latency (s)
Unconstrained + post-hoc	0.0	2.43	0.0	2.40
Naive constrained	timeout	timeout	timeout	timeout
Ours (combined)	4.0	5.03	0.0	2.63

Table 12: Ablation at $K = 5{,}000$. Each row removes one component. Speculative short-circuiting provides no benefit at this K (trie masking alone is fast). The trie's compilation advantage (Table 3: 3.5–13.7$\times$ at $K \geq 5{,}000$) and per-step masking advantage (0.65μs vs. 5.9μs) compound in batch serving.

Configuration	Latency (s)	Schema Compliance (%)
Trie automaton	0.77	100
Hierarchical only	3.99	100
Speculative only	2.00	100
Baseline (XGrammar regex)	0.89	100

Table 13: vLLM end-to-end per-example latency (seconds), including both compilation and inference. "Guided choice" uses vLLM's built-in EBNF-based choice mode; "guided regex" uses XGrammar regex compilation. Both use XGrammar as the backend; our trie integration (Table 1) shows larger gains at higher batch sizes.

Method	K=500	K=1K	K=5K	K=10K
Guided choice (ours)	0.05	0.06	0.08	0.10
Guided regex (baseline)	0.48	1.20	2.93	5.88

D.2 vLLM Performance Results

E Limitations and Future Work

Scope. The trie automaton optimizes exactly one constraint pattern: flat finite-set selection. While this is common (tool routing, classification, entity linking), real-world schemas typically combine enum fields with structural constraints. Our mixed-schema evaluation (Section 5) validates compilation time but not end-to-end throughput for nested schemas. Dynamic enum updates require full recompilation (37ms, negligible in practice but not incremental).

Baselines. LLGuidance's Earley parser avoids upfront compilation entirely and achieves faster compilation than both XGrammar and our trie (Table 19); however, its per-step cost remains $\mathcal{O}(V)$ for finite sets (73–141μs vs. our 0.65μs). If serving engines move masking to GPU (Section H.6), the CPU per-step advantage becomes less relevant, though the trie's compact bitmask representation remains more amenable to GPU transfer.

Implementation. The trie is in Rust while XGrammar uses C++. The per-step advantage (precomputed lookup vs. dynamic computation) is algorithmic and independent of implementation language.

F Detailed Tool Routing Example

Consider an agentic system routing requests to the correct tool from a registry of $K = 2{,}000$ APIs (e.g., `aws.cloudwatch.get_metric_statistics`), averaging $L_{\max} = 30$ characters with strong prefix structure. FSM compilation requires $K \cdot L_{\max} \cdot |\Sigma| = 15.4$M character-level transitions (25–50s). Per-step masking requires $V \cdot \bar{L}_{\text{tok}} = 102{,}400$ FSM traversals per step; with 6.4 tokens per tool name and cache effects ($f \approx 2.0$ due to the 47 MB transition table exceeding L2 cache), the effective cost is $\sim 1.3 \times 10^6$ operations per selection. In an agentic loop with 5–10 tool calls per task, this compounds to minutes of latency, forcing practitioners to cap the registry or abandon constrained decoding.

G Theoretical Analysis

We analyze the complexity of both the standard FSM pipeline and the trie automaton, then establish correctness and discuss practical performance considerations. Throughout, let $N_{\text{chars}} = \sum_{i=1}^{K} |e_i|$ denote the total character count across all enum values.

G.1 FSM Complexity Analysis

For an enum $\mathcal{E} = \{e_1, e_2, \ldots, e_K\}$, the standard approach constructs a DFA from the regular expression $e_1|e_2|\cdots|e_K$ (Hopcroft et al., 1979) via Thompson's construction (Thompson, 1968) followed by subset construction and Hopcroft minimization (Hopcroft, 1971).

State space. The minimal DFA has between $|\mathcal{T}(\mathcal{E})| + 1$ and $1 + N_{\text{chars}}$ states, where $|\mathcal{T}(\mathcal{E})|$ is the trie node count (number of distinct prefixes) and the $+1$ accounts for the dead state. The upper bound $1 + N_{\text{chars}} \leq 1 + K \cdot L_{\max}$ is achieved when no enum values share prefixes; the lower bound is achieved when the trie is the minimal DFA (which it always is for finite string unions, up to the dead state). This follows from the Myhill-Nerode theorem: each distinct prefix defines a distinct equivalence class.

[2]Retry compliance is 0% because synthetic tool names (e.g., `slack.get_user`) are never produced verbatim by unconstrained generation, even with 3 retries. On natural-language labels (Table 6), unconstrained generation achieves non-zero but imperfect validity.

[3]Combined uses the trie for enum selection and hierarchical rewriting for structural dispatch.

Compilation cost. The three-phase pipeline costs:

$$C_{\text{compile}}^{\text{FSM}} = \underbrace{\mathcal{O}(N_{\text{chars}})}_{\text{NFA construction}} + \underbrace{\mathcal{O}(N_{\text{chars}} \cdot |\Sigma|)}_{\text{subset construction}} + \underbrace{\mathcal{O}(N_{\text{chars}} \cdot |\Sigma| \cdot \log N_{\text{chars}})}_{\text{Hopcroft minimization}} \tag{2}$$

The dominant term is minimization. The transition table requires $\mathcal{O}(N_{\text{chars}} \cdot |\Sigma|)$ space. For $K = 2{,}000$, $L_{\max} = 30$, $|\Sigma| = 256$, this is ≈ 15.4 million operations.

Per-step masking cost. At each decoding step, the system checks all V vocabulary tokens by simulating up to ℓ character transitions per token:

$$C_{\text{mask}}^{\text{FSM}} = \mathcal{O}(V \cdot \ell) \tag{3}$$

The total decoding cost for one enum value of character length L is $(L/\bar{\ell}) \cdot (C_{\text{mask}}^{\text{FSM}} + C_{\text{forward}})$, where $\bar{\ell}$ is the average token length and C_{forward} is the LLM forward pass cost.

G.2 Trie Automaton Complexity

Compilation cost. Trie construction costs $\mathcal{O}(N_{\text{chars}})$. The naive mask precomputation (checking every token at every node) costs $\mathcal{O}(N_{\text{chars}} \cdot V \cdot \ell)$, but using Aho-Corasick multi-pattern matching (Aho & Corasick, 1975) over the vocabulary token strings reduces this to:

$$C_{\text{compile}}^{\text{trie}} = \mathcal{O}((N_{\text{chars}} + V) \cdot \ell) \tag{4}$$

The Aho-Corasick automaton is built over the V token strings in $\mathcal{O}(V \cdot \ell)$, then the trie is traversed depth-first, maintaining the AC state across edges (saving and restoring at branch points so each trie edge is processed exactly once). The total text length is thus N_{chars}, requiring $\mathcal{O}(N_{\text{chars}})$ character steps; the total number of reported matches across all positions is bounded by $\mathcal{O}(N_{\text{chars}} \cdot \ell)$ since at each of the N_{chars} character positions, at most ℓ tokens of different lengths can start there. The overall cost is thus $\mathcal{O}(N_{\text{chars}} \cdot \ell + V \cdot \ell) = \mathcal{O}((N_{\text{chars}} + V) \cdot \ell)$.

Per-step masking cost. After precomputation, masking is a lookup into the stored valid token list:

$$C_{\text{mask}}^{\text{trie}} = \mathcal{O}(|\text{valid}[s_t]|) \tag{5}$$

The valid set size shrinks exponentially with trie depth: assuming each character position in the vocabulary is drawn independently and uniformly from Σ, the probability that a token of length j matches a specific j-character trie path is $|\Sigma|^{-j}$, giving $\mathbb{E}[|\text{valid}[s_t]|] \leq V \cdot \bar{\ell}/|\Sigma|^{d(s_t)}$ where $d(s_t)$ is the node depth. This is a loose upper bound (real tokenizers have non-uniform character distributions), but the qualitative exponential decay is confirmed empirically: after 3–4 characters of prefix, the valid set is typically 10–100 tokens, making per-step cost effectively constant.

G.3 Complexity Comparison

Table 14 summarizes the asymptotic and concrete costs. The trie automaton's advantage comes from two sources: (1) compilation avoids the $|\Sigma|$ factor (processing only characters that appear, not the full alphabet), and (2) per-step masking replaces a scan over all V tokens with a precomputed lookup.

The compilation speedup is driven by the $|\Sigma|$ factor: the FSM must fill transition entries for all 256 ASCII characters at each state, while the trie only processes characters that actually appear. The per-step speedup comes from replacing a full vocabulary scan with a cached lookup whose size shrinks with trie depth.

G.4 Correctness

The trie automaton is not an approximation — it produces identical outputs to the FSM approach. This is the key formal guarantee.

Table 14: Complexity comparison between FSM and trie automaton. Concrete values use $K = 2{,}000$, $L_{\max} = 30$, $|\Sigma| = 256$, $V = 32{,}000$, $\ell = 4$.

	FSM	**Trie (Aho-Corasick)**	**Ratio**
Compilation	$\mathcal{O}(N_{\text{chars}} \cdot \lvert\Sigma\rvert \cdot \log N_{\text{chars}})$	$\mathcal{O}((N_{\text{chars}} + V) \cdot \ell)$	$\sim \frac{\lvert\Sigma\rvert \cdot \log N_{\text{chars}}}{\ell}$
concrete (subset only)	∼15.4M ops	∼370K ops	∼40×
Per-step mask	$\mathcal{O}(V \cdot \ell)$	$\mathcal{O}(\lvert\text{valid}[s_t]\rvert)$	$\sim \frac{V \cdot \ell}{\lvert\text{valid}\rvert}$
concrete	∼128K lookups	∼50–500 lookups	∼100×
Working set	$\lvert\mathcal{S}\rvert \cdot \lvert\Sigma\rvert \cdot 4$ B	$\lvert\text{valid}[s_t]\rvert \cdot 4$ B	
concrete	∼47 MB	<1 KB	fits L1

Proposition 1 (Output equivalence)**.** *Let $p(y_t \mid \mathbf{y}_{<t})$ be the LLM's unconstrained next-token distribution, and let $\mathcal{V}_{\text{dec}} \subseteq \mathcal{V}$ be the set of vocabulary tokens with valid character decompositions (excluding special tokens such as* `<pad>`*,* `<unk>` *that do not correspond to character sequences). For any enum $\mathcal{E}$ and any prefix $\mathbf{y}_{<t}$, the constrained distributions produced by the FSM and trie automaton are identical:*

$$p_c^{\text{FSM}}(y_t \mid \mathbf{y}_{<t}) = p_c^{\text{trie}}(y_t \mid \mathbf{y}_{<t}) \quad \forall\, y_t \in \mathcal{V}_{\text{dec}} \tag{6}$$

Consequently, greedy decoding and fixed-seed sampling produce identical outputs under both methods.

Proof. It suffices to show that $\mathcal{A}^{\text{FSM}}(s_t) = \mathcal{A}^{\text{trie}}(s_t)$ for all reachable states s_t, since the constrained distribution $p_c(y_t \mid \mathbf{y}_{<t}) \propto p(y_t \mid \mathbf{y}_{<t}) \cdot \mathbf{1}[y_t \in \mathcal{A}(s_t)]$ is determined entirely by the valid token set and the unconstrained distribution.

Both methods define validity over $\mathcal{V}_{\text{dec}}$ as follows: a token v with character decomposition $c_1 \cdots c_{|v|}$ is valid at state s_t if and only if (i) the sequence of transitions $s_t \xrightarrow{c_1} s' \xrightarrow{c_2} \cdots \xrightarrow{c_{|v|}} s''$ does not encounter a dead state, and (ii) from the resulting state s'', there exists at least one string $w \in \Sigma^*$ such that $s'' \xrightarrow{w} s_{\text{acc}}$ for some accept state s_{acc} (i.e., the consumed prefix $\mathbf{y}_{<t} \cdot v$ is a prefix of some $e_i \in \mathcal{E}$).

The FSM computes this at each decoding step by simulating transitions $\delta(s_t, c_1), \delta(\cdot, c_2), \ldots$ for each token $v \in \mathcal{V}_{\text{dec}}$. The trie computes this during precomputation by walking each token's characters down the trie from node s_t.

The two computations produce identical results because the trie for a finite set $\mathcal{E}$ is isomorphic to the minimal DFA for $\mathcal{L}_{\mathcal{E}}$. Specifically, by the Myhill-Nerode theorem, the equivalence classes of the right-congruence relation for $\mathcal{L}_{\mathcal{E}}$ are exactly the distinct prefixes of the enum values (plus the equivalence class of strings that are not prefixes of any e_i, corresponding to the dead state). Each trie node represents one such equivalence class, so the trie node set plus a dead state is in bijection with the minimal DFA state set. Under this bijection, the transition functions agree: $\delta_{\text{DFA}}(s, c) = \delta_{\text{trie}}(s, c)$ for all states s and characters c. Therefore, the multi-character extension $\delta^*(s_t, v)$ produces the same result under both representations, and the accept-reachability check (condition (ii)) is identical since the accept states correspond to the same trie leaves.

The precomputed mask $\text{valid}[s_t]$ stores exactly the set $\{v \in \mathcal{V}_{\text{dec}} :$ conditions (i) and (ii) hold$\}$, so $\mathcal{A}^{\text{trie}}(s_t) = \text{valid}[s_t] = \mathcal{A}^{\text{FSM}}(s_t)$. □

EOS token handling. The restriction to $\mathcal{V}_{\text{dec}}$ (excluding special tokens) means the trie and FSM may differ in whether an EOS token is appended after the enum value is complete. In our vLLM integration, the trie signals completion by including only EOS in the valid set at leaf nodes, matching vLLM's expected termination protocol. Our empirical verification (Section 5) confirms that all *content* tokens are identical; the only difference is the trailing EOS, which does not affect the decoded string.

Proposition 2 (Hierarchical cardinality reduction)**.** *Let $\mathcal{E}$ be partitioned into G groups of sizes $K_1, \ldots, K_G$ with $\sum_j K_j = K$. The effective per-step cardinality is $C_{\text{eff}} = \max(G, \max_j K_j)$. For balanced partitions, this is minimized at $C^*_{\text{eff}} = \lceil \sqrt{K} \rceil$ when $G = \lceil \sqrt{K} \rceil$.*

Proof of Proposition 2. The two-level scheme requires one constrained decoding call over G group names, then one call over K_j values within the selected group j. The per-step cardinality is thus $\max(G, \max_j K_j)$. For balanced partitions, $\max_j K_j = \lceil K/G \rceil$, so $C_{\text{eff}} = \max(G, \lceil K/G \rceil)$. Since $\max(a, b) \geq \sqrt{ab}$ for $a, b > 0$, we have $\max(G, K/G) \geq \sqrt{K}$, with equality when $G = K/G$, i.e., $G = \sqrt{K}$. Rounding gives $C^*_{\text{eff}} = \lceil \sqrt{K} \rceil$. □

G.5 Cache Effects and Practical Performance

The analysis above uses the unit-cost RAM model. Real hardware introduces cache hierarchy effects that significantly impact the FSM approach but not the trie automaton.

The FSM transition table occupies $|\mathcal{S}| \cdot |\Sigma| \cdot w$ bytes (where w is the word size). When this exceeds L2 cache capacity C_{L2}, transition lookups incur main-memory access penalties. We model the effective per-step cost as $C^{\text{FSM}}_{\text{mask,eff}} = V \cdot \ell \cdot f(|\mathcal{S}|)$, where:

$$f(|\mathcal{S}|) = \begin{cases} 1 & \text{if } |\mathcal{S}| \cdot |\Sigma| \cdot w \leq C_{\text{L2}} \\ \alpha \cdot \frac{|\mathcal{S}| \cdot |\Sigma| \cdot w}{C_{\text{L2}}} & \text{otherwise} \end{cases} \tag{7}$$

with $\alpha \approx 2$–3 reflecting the ratio of main memory to cache access latency, attenuated by hardware prefetching. For the trie automaton, the working set per step is $|\text{valid}[s_t]| \cdot w <$ 1 KB, which always fits in L1 cache, so $f^{\text{trie}} = 1$.

This cache penalty grows with K: at $K = 2{,}000$ ($|\mathcal{S}| \approx 48{,}000$, table ≈ 47 MB), $f \approx 2.0$; at $K = 10{,}000$ ($|\mathcal{S}| \approx 240{,}000$, table ≈ 235 MB), $f \approx 2.5$. This compounds with the linear scaling of $|\mathcal{S}|$ in K, producing the super-linear degradation observed in our experiments (Section 5).

G.6 Enum Window Scaling

Given a latency budget T_{budget}, we can estimate the maximum enum cardinality each approach supports. In the compilation-dominated regime ($C_{\text{compile}} \gg C_{\text{decode}}$), setting $T_{\text{budget}} \geq C_{\text{compile}}/c_{\text{sys}}$ (where c_{sys} is the system throughput in operations per second) and solving for K gives conservative (worst-case) bounds using $N_{\text{chars}} \leq K \cdot L_{\max}$:

$$K^{\text{FSM}}_{\max} \approx \frac{c_{\text{sys}} \cdot T_{\text{budget}}}{L_{\max} \cdot |\Sigma|}, \qquad K^{\text{trie}}_{\max} \approx \frac{c_{\text{sys}} \cdot T_{\text{budget}}}{L_{\max} \cdot \ell} \tag{8}$$

These are lower bounds on the achievable K; with prefix sharing ($N_{\text{chars}} < K \cdot L_{\max}$), the actual limits are higher. The cardinality expansion factor is $K^{\text{trie}}_{\max}/K^{\text{FSM}}_{\max} = |\Sigma|/\ell \approx 64\times$ for ASCII with maximum token length 4. For a 5-second timeout with $L_{\max} = 30$ and $|\Sigma| = 256$, the FSM supports $K_{\max} \approx 500$–$1{,}000$ (matching OpenAI's documented limit), while the trie automaton supports $K^{\text{trie}}_{\max} \approx 30{,}000$–$100{,}000$. Google Gemini's lower limit ($\approx$120) and Anthropic's compilation timeout (Anthropic, 2025) are consistent with more conservative timeouts or less optimized compilation.

H Complexity Analysis: Concrete Examples

H.1 Scaling Example

The following comparison illustrates the scaling differences for a concrete example with $K = 2{,}000$ enum values (compilation numbers shown at $K = 100{,}000$ to illustrate the scaling regime):

Method	Compilation Cost	Per-step Cost
General FSM	$\mathcal{O}(K \cdot L_{\max} \cdot \lvert\Sigma\rvert) \approx 15$s	$\mathcal{O}(V \cdot \ell) \approx 4.7\mu$s
Trie Automaton	$\mathcal{O}((N_{\text{chars}}+V) \cdot \ell) \approx 67$ms	$\mathcal{O}(\lvert\text{valid}[s_t]\rvert) \approx 0.65\mu$s

To understand these complexity differences concretely, consider an $\mathcal{E}$ with $K = 2{,}000$ values, maximum length $L_{\max} = 30$, ASCII alphabet $|\Sigma| = 256$, vocabulary size $V = 32{,}000$, and maximum token length $\ell = 4$ characters. Assuming moderate prefix sharing, the total character count is $N_{\text{chars}} = 60{,}000$.

For FSM compilation, the subset construction cost alone is $N_{\text{chars}} \cdot |\Sigma| = 60{,}000 \times 256 \approx 15.4$ million operations. Including Hopcroft minimization adds a $\log N_{\text{chars}} \approx 11$ factor, yielding ≈ 169 million total operations. In contrast, trie compilation requires $\mathcal{O}((N_{\text{chars}} + V) \cdot \ell) = (60{,}000 + 32{,}000) \times 4 = 368{,}000$ operations, a $40\times$ reduction against subset construction alone, or $460\times$ including minimization.

The per-step masking cost difference is equally dramatic. FSM-based masking requires checking all $V \cdot \ell = 32{,}000 \times 4 = 128{,}000$ character positions against the current state. The trie automaton only examines tokens in valid$[s_t]$, which typically contains 50–500 entries depending on the current trie node depth and prefix sharing. This represents a 250–2500$\times$ speedup in the common case.

H.2 Controlled Comparison: Precomputed Masks

To isolate the algorithmic contribution from integration-path effects, we implemented the reviewer-suggested controlled comparison: precomputing XGrammar's per-state bitmasks for all DFA states and serving them via cached lookup (the same path the trie uses). Results on Qwen3-8B ($K = 1{,}000$):

	Trie	XGrammar precomputed
States/nodes	6,786	6,786 (identical)
Precomputation time	33ms	6.5s (196$\times$ slower)
Per-step lookup	0.08μs	0.08μs (identical)

The number of DFA states equals the number of trie nodes (both are the minimal DFA for $\mathcal{L}_{\mathcal{E}}$, per Proposition 1). Once precomputed, per-step lookup is identical. The difference is entirely in precomputation efficiency: the trie's AC-based approach computes all masks in $\mathcal{O}((N_{\text{chars}} + V) \cdot \ell) = 33$ms, while enumerating XGrammar's DFA states and extracting per-state masks requires instantiating a `GrammarMatcher` per state, taking 6.5s. This confirms that the trie is the efficient algorithm for precomputing per-state masks over finite sets.

Could DFA-based precomputation be faster? The 196$\times$ ratio reflects XGrammar's current API (per-state `GrammarMatcher` instantiation). A purpose-built DFA traversal that directly enumerates states and computes masks could be faster. However, the fundamental cost remains $\mathcal{O}(|\mathcal{S}| \cdot V \cdot \ell)$ (checking each token at each state), while the trie+AC approach achieves $\mathcal{O}((N_{\text{chars}} + V) \cdot \ell)$ by amortizing vocabulary matching across shared prefixes. The asymptotic gap is a factor of $|\mathcal{S}|/(N_{\text{chars}}/V + 1)$, which grows with K.

H.3 Per-Step Cost Breakdown

Table 3 reports valid-token computation time only. The full per-step breakdown including bitmask application ($K = 1{,}000$, Qwen3-8B):

Component	Trie (μs)	XGrammar (μs)
Valid-token computation	0.08	4.5
Bitmask application to logits ($\mathcal{O}(V)$)	$\sim$31	$\sim$31
Total per-step	$\sim$31	$\sim$36

The $\mathcal{O}(V)$ bitmask application ($\sim$31μs via PyTorch tensor operation for $V = 151$K) dominates single-request per-step cost and is method-independent. The trie's valid-token computation advantage (0.08 vs. 4.5μs) is a small fraction of total per-step time for a single request. The advantage compounds in batch serving: at $B = 128$, valid-token computation runs 128 times (trie: 10μs total; XGrammar: 576μs), and the trie's precomputed masks enable the stateless `LogitsProcessor` path that bypasses vLLM's guided decoding pipeline overhead.

Reconciling per-step numbers. Table 3 reports 0.65μs (trie) and 5.8μs (XGrammar) at $K =$ 1,000, while the breakdown above reports 0.08μs and 4.5μs. The difference is measurement methodology: Table 3 measures the full per-step masking operation as invoked during batch serving (including Python binding overhead, bitmask copy from the precomputed store, and loop dispatch), averaged over 1,000 iterations at realistic decoding states. The 0.08μs above isolates the raw valid-token-list lookup in a tight microbenchmark loop. Both are valid measurements at different abstraction levels; the Table 3 numbers reflect the cost actually incurred during serving. The ratio is consistent: $\sim$9$\times$ (Table 3: 0.65 vs. 5.8μs) vs. $\sim$56$\times$ (raw lookup: 0.08 vs. 4.5μs), with the difference attributable to fixed per-call overhead that is proportionally larger for the faster method.

H.4 Precomputation Cost Breakdown

The trie automaton's precomputation consists of four distinct phases, each with different computational characteristics:

Phase 1: Trie Construction Building the trie from $\mathcal{E}$ values requires $\mathcal{O}(N_{\text{chars}})$ operations with one insertion per character. This phase is typically fast, completing in milliseconds even for large enums.

Phase 2: Vocabulary Decoding Converting each vocabulary token from its integer ID to its character string representation requires $\mathcal{O}(V \cdot \ell)$ operations. For modern tokenizers with $V \approx 32{,}000$ and $\ell \approx 4$, this involves roughly 128K character accesses.

Phase 3: Aho-Corasick Automaton Construction Building the multi-pattern matcher over all vocabulary tokens requires $\mathcal{O}(V \cdot \ell)$ time and space. The resulting automaton enables efficient simultaneous matching of all tokens against any input string.

Phase 4: Trie Traversal with AC Matching The trie is traversed depth-first, maintaining the AC automaton state across edges (saving and restoring at branch points so each edge is processed exactly once). At each node, the automaton identifies all tokens whose character strings start at that position. This requires $\mathcal{O}(N_{\text{chars}} \cdot \ell)$ operations in total, as each character position may trigger up to ℓ token matches.

The Aho-Corasick optimization is crucial for efficiency. A naive approach would check each of the V tokens against each of the $|T(\mathcal{E})|$ trie nodes independently, requiring $\mathcal{O}(V \cdot |T(\mathcal{E})| \cdot \ell)$ operations. The AC automaton reduces this to $\mathcal{O}((V + |T(\mathcal{E})|) \cdot \ell)$ by building the multi-pattern matcher once and traversing the trie once with AC state maintenance.

H.5 Memory vs. Speed Trade-off

The precomputed lookup tables require $\mathcal{O}(N_{\text{chars}} \cdot \bar{f})$ memory, where $\bar{f}$ is the average fanout (number of valid tokens per trie node). At each trie node, we store a list of valid token IDs, with each ID requiring 4 bytes. The total memory footprint is $\sum_{n \in T(\mathcal{E})} |\text{valid}[n]| \times 4$ bytes.

Concrete memory requirements scale predictably with $\mathcal{E}$ size:

K	Trie Memory	FSM Transition Table (theoretical)	Memory Ratio
100	∼20 KB	∼3 MB	150×
1,000	∼120 KB	∼20 MB	167×
10,000	∼0.9 MB	∼200 MB	222×
100,000	∼8 MB	∼2 GB	250×

The FSM column shows the theoretical worst-case $|\mathcal{S}| \times |\Sigma| \times 4$ bytes. In practice, XGrammar uses vocabulary partitioning and compressed representations that significantly reduce actual memory. Measured RSS overhead (process-level) at $K = 10{,}000$: XGrammar 9.6 MB, trie 2.2 MB (4.4× ratio), substantially less than the theoretical 222× but still a meaningful advantage.

For memory-constrained environments, we employ *lazy computation*: compute valid$[n]$ only when node n is first visited during decoding, and use an LRU cache to bound memory usage. The total space for full precomputation is $\mathcal{O}(|T(\mathcal{E})| \cdot \bar{f})$, where $\bar{f}$ is the mean fanout per node; this is typically 1–8 MB for $K \leq 100{,}000$ (Table 4), well within L3 cache. The cache size can be tuned based on available memory, as even a 1MB cache provides substantial speedup by avoiding recomputation of frequently accessed nodes. The Aho-Corasick-based precomputation can also be performed lazily by restricting the multi-pattern match to subtrees reachable from the current generation prefix, further reducing memory pressure.

H.6 GPU-Based Masking Considerations

Our analysis assumes CPU-based masking, which is the current practice in vLLM and SGLang. Modern serving engines increasingly explore GPU-based logit masking via precomputed bitmask tensors, where the masking operation becomes a single elementwise multiply, $\mathcal{O}(V)$ but massively parallel and essentially free relative to the forward pass. If masking moves entirely to GPU, the per-step CPU cost advantage diminishes. However, the trie's compact bitmask representation (8 MB at $K = 100{,}000$ vs. ∼2 GB for FSM transition tables) makes it substantially more amenable to GPU transfer, and the compilation time advantages are independent of where masking executes. The trie's precomputed per-node bitmasks are directly usable as GPU tensors without conversion, whereas FSM-based approaches must still compute the valid set per state before transferring.

H.7 Prefix Sharing Analysis

Prefix sharing fundamentally determines the trie automaton's efficiency. We define the prefix sharing ratio as $r = |T(\mathcal{E})|/N_{\text{chars}}$, representing the fraction of characters that correspond to unique trie nodes. When $r \approx 1$ (minimal sharing), the trie has nearly as many nodes as total characters. When $r \ll 1$ (heavy sharing), the trie is much more compact.

The prefix sharing ratio affects performance across multiple dimensions:

Compilation Time The trie traversal phase scales with $|T(\mathcal{E})|$, not N_{chars}. Heavy prefix sharing (small r) reduces compilation time proportionally. For example, AWS API names with common `aws.*` prefixes might achieve $r = 0.3$, reducing compilation time by 70%.

Memory Usage Fewer trie nodes directly translate to lower memory consumption. The memory scaling becomes $\mathcal{O}(r \cdot N_{\text{chars}} \cdot \bar{f})$, where r acts as a compression factor.

Per-step Masking Nodes near the trie root (shared prefixes) tend to have higher fanout, while deeper nodes have lower fanout. This creates a natural filtering effect: early in the generation process, many tokens remain valid, but the valid set shrinks rapidly as the prefix becomes more specific.

Real-world examples demonstrate significant variation in prefix sharing:

- **Tool/API names** (e.g., `aws.s3.*`, `google.cloud.*`): $r \approx 0.2$–0.4

- **Medical codes** (e.g., ICD-10 with hierarchical structure): $r \approx 0.15$–0.25
- **Random strings** (e.g., UUIDs, random identifiers): $r \approx 0.95$–1.0
- **Natural language** (e.g., city names, product names): $r \approx 0.6$–0.8

Empirical prefix sharing in our benchmarks. Table 15 reports the measured prefix sharing ratio r for all enum sets used in our experiments. The classification benchmarks have moderate-to-high r (0.59–0.90), reflecting natural language labels with limited prefix overlap. Synthetic tool names have low r (0.36–0.40) due to namespace prefixes (`slack.get_*`, `aws.create_*`), representing the regime where the trie excels most.

Table 15: Prefix sharing ratio $r = |\mathcal{T}(\mathcal{E})|/N_{\text{chars}}$ for benchmark enum sets. Lower r indicates more prefix sharing and greater trie compression.

Dataset	K	N_{chars}	Trie nodes	r
TREC	42	559	503	0.90
MASSIVE	59	813	478	0.59
Banking77	77	1,572	1,241	0.79
CLINC150	150	1,768	1,322	0.75
Synthetic tools (K=1,000)	590	11,236	4,503	0.40

Valid set size distribution. A concern is that $|\text{valid}[s_t]|$ may be $\mathcal{O}(V)$ at the root node. Table 16 shows the measured distribution across trie depth for $K = 1{,}000$ synthetic tool names (Qwen3-8B, 151K vocabulary). The root node has only 72 valid tokens (0.05% of V), not V, because most vocabulary tokens do not start with any character present in the trie's root children. By depth 2, the mean valid set shrinks to 3 tokens. The increase at depths 3–5 reflects the structure of tool names: short shared prefixes (e.g., `get_`) end at depth 3–4, after which diverse suffixes create more branching before converging again at deeper levels.

Table 16: Distribution of $|\text{valid}[s_t]|$ by trie depth ($K = 1{,}000$, Qwen3-8B).

Depth	Mean $\lvert$valid$\rvert$	Max $\lvert$valid$\rvert$	Nodes
0	72	72	1
1	6	17	12
2	3	7	19
3	5	35	21
4	10	39	21
5	8	43	34

H.8 Comparison with Token-Level Tries (GENRE)

GENRE (De Cao et al., 2021) builds tries at token granularity, where each edge is a full BPE token ID. This avoids the BPE-trie alignment problem entirely but loses character-level prefix sharing. We compare compilation time on synthetic tool names (Qwen3-8B, $V = 151$K):

K	Token trie (ms)	Char trie + AC (ms)	Char nodes	Token nodes
100	4.5	31	1,380	456
1,000	37	35	8,462	4,269
5,000	183	43	24,837	18,925
10,000	369	52	42,460	36,515

At small K, the token trie is faster because it requires no mask precomputation (valid tokens are simply the children keys). Our Rust implementation with AC precomputation crosses over at $K \approx 1{,}000$ and is $7\times$ faster at $K = 10{,}000$, because the character-level trie has fewer nodes ($3\times$ at $K = 100$, $1.2\times$ at $K = 10{,}000$) and the AC automaton amortizes vocabulary matching across shared prefixes. A Python token-trie implementation (our

own, following GENRE's design) shows the same crossover ($K \approx 1{,}000$) and similar per-step costs (0.43–0.63μs vs. our 0.40–0.64μs), confirming the advantage is algorithmic, not implementation-specific. Both methods produce identical outputs (Proposition 1).

Per-step masking comparison. The token-level trie has a per-step advantage: masking is a single hash lookup on the current token ID to retrieve child keys, costing $\mathcal{O}(b)$ where b is the branching factor at the current node (typically 1–10 after the first token). Our character-level trie costs $\mathcal{O}(|\text{valid}[s_t]|)$ per step (empirically 0.65μs, dominated by bitmask copy). In practice, both are sub-microsecond and negligible relative to the GPU forward pass. The character-level trie's advantage is in compilation at large K and tokenizer independence; the token-level trie's advantage is in per-step simplicity at small K.

Output equivalence at scale. To verify that character-level and token-level tries produce identical outputs (not just enforce the same constraint set), we ran both on a synthetic tool-selection task at $K \in \{500, 1{,}000, 2{,}000, 5{,}000\}$ with Qwen3-8B (100 samples per K, greedy decoding). Both methods produced identical outputs on every sample at every K, confirming that the BPE-trie alignment via AC does not introduce any approximation relative to GENRE-style token-level tries.

H.9 Alternative Baselines

Vocabulary pre-filtering. A natural question is whether simply filtering the vocabulary to tokens that appear as substrings of enum values can speed up FSM compilation. We build the set of all substrings (up to length 20) of the enum values and count matching vocabulary tokens. On Qwen3-8B (151K vocab), only 0.2% of tokens match (379/151K) regardless of K, because most BPE tokens contain characters not present in tool names. However, pre-filtering itself costs 5–405ms (scaling linearly with K), comparable to or slower than the trie's total compilation (30–48ms). Moreover, pre-filtering cannot reduce XGrammar's compilation cost because the bottleneck is DFA state construction, not vocabulary scanning.

Cached DFA compilation. Since the trie's AC automaton can be cached per-tokenizer, a fair comparison should also cache XGrammar's tokenizer info. With warm `TokenizerInfo`, XGrammar compilation drops from 529–1749ms (cold) to 3.7–1207ms (warm). However, the warm XGrammar still scales linearly with K (246ms at K=1,000, 1207ms at K=10,000), while the trie remains nearly flat (35–48ms). The trie is faster than warm XGrammar at $K \geq 100$, confirming that the advantage is algorithmic (avoiding DFA construction), not merely an artifact of cold-start overhead.

Hash-set prefix matching. A simpler baseline would maintain a hash set of valid strings and, at each decoding step, check which vocabulary tokens are consistent with remaining valid strings given the current prefix. This avoids both DFA compilation and AC construction. However, this approach has $\mathcal{O}(K \cdot \ell)$ per-step cost (checking each token against K strings), which is worse than the trie's $\mathcal{O}(|\text{valid}[s_t]|)$ precomputed lookup and comparable to the FSM's $\mathcal{O}(V \cdot \ell)$ when K is large. The trie's advantage is precisely that it moves this work to compilation time.

H.10 Validity at High Cardinality

A key reviewer concern is whether the trie enables constrained decoding at K values where FSM approaches become impractical. Table 17 shows validity rates on a synthetic tool-selection task at $K = 500$–5,000 (Qwen3-8B, 100 samples per K). The trie guarantees 100% validity at all K, while unconstrained decoding drops to 84% at $K = 1{,}000$. Both char-level and token-level tries produce identical outputs.

H.11 When the Trie Automaton Excels

The trie automaton provides the greatest benefit under specific conditions that can be systematically identified:

Table 17: Validity (%) at high cardinality. Trie-constrained decoding guarantees 100% valid outputs regardless of K; unconstrained decoding validity degrades. Char-level and token-level (GENRE-style) tries produce identical outputs at all K.

K	Char trie valid	Token trie valid	Uncon. valid	Char=Token?
500	100%	100%	97%	✓
1,000	100%	100%	84%	✓
2,000	100%	100%	89%	✓
5,000	100%	100%	98%	✓

Enum Size Threshold For small enums ($K < 50$), the compilation overhead dominates and simple approaches suffice. The trie automaton becomes advantageous when $K > 100$, with benefits increasing dramatically beyond $K = 1{,}000$.

Prefix Structure Enums with meaningful prefix sharing ($r < 0.8$) see substantial memory and compilation time reductions. Random string enums with no structure ($r \approx 1.0$) still benefit from faster per-step masking but lose the compilation advantages.

Vocabulary Characteristics Large vocabularies ($V > 10{,}000$) with long average token length ($\ell > 2$) make naive per-step masking expensive, amplifying the trie automaton's per-step advantages.

Usage Pattern Systems serving repeated queries with the same schema can amortize pre-computation costs. Interactive applications with latency budgets under 1 second particularly benefit from the faster per-step masking.

Based on these factors, we recommend the following decision criteria:

- **Use trie automaton** when $K > 100$ and enum values have identifiable prefix structure
- **Use hierarchical clustering** (Appendix I) when $K > 5{,}000$ to manage compilation time
- **Use speculative decoding** (Appendix J) when $K > 10{,}000$ or latency budget < 1s
- **Use simple FSM** only when $K < 50$ or when prefix sharing is minimal ($r > 0.9$)

The crossover point where FSM compilation becomes slower than trie compilation typically occurs around $K = 50$–100, depending on prefix sharing and vocabulary size. Beyond $K = 1{,}000$, the trie automaton consistently outperforms FSM-based approaches by orders of magnitude.

I Hierarchical Schema Rewriting Details

For $K > 50{,}000$, hierarchical decomposition partitions the enum into $G \approx \sqrt{K}$ groups, reducing per-step cardinality from K to $\sqrt{K}$ (Proposition 2). We consider three clustering approaches: (1) string similarity (edit distance + hierarchical clustering), (2) semantic similarity (sentence embeddings + k-means), and (3) domain taxonomy (e.g., ICD-10 chapter structure). Effectiveness depends on cluster coherence: when >80% of within-cluster pairs are more similar than between-cluster pairs, the LLM reliably selects the correct group. The approach is robust to 10–20% misassignment (per-step cardinality remains $\mathcal{O}(\sqrt{K})$), but poor clustering degrades accuracy. This is a preliminary direction requiring further validation.

J Speculative Short-Circuiting Details

Speculative short-circuiting uses a lightweight scoring function (embedding similarity, unconstrained LLM generation, or a small classifier) to identify a top-k shortlist before applying trie-constrained decoding. This trades the 100% validity guarantee for reduced latency at extreme K. Empirical recall varies by domain: Recall@20 ranges from 87% (medical coding) to 97% (geographic entities). A cascading fallback (top-k $\rightarrow$ top-$2k$ $\rightarrow$ full decoding) maintains schema compliance. This extension is most effective when input context strongly predicts the correct enum value and least effective for highly similar enum values. Like hierarchical rewriting, this is a preliminary direction.

K Constraint-Aware Dispatch Beyond Finite Sets

To validate that the dispatch principle generalizes beyond finite-set constraints, we benchmark a *character-position mask* engine for fixed-format strings against xgrammar's regex compilation. For a format like `YYYY-MM-DD`, each character position has a known set of valid characters (digits, hyphens, etc.). The specialized engine builds a char-indexed vocabulary and checks only tokens whose first character matches the allowed set at each position, avoiding the full DFA construction. Table 18 shows compilation speedups across all seven tokenizer families and four format types.

Table 18: Compilation speedup of character-position masks vs. xgrammar regex for fixed-format strings. Values are $\times$ faster (median over 10 runs). xgrammar compilation time is constant per format regardless of format complexity (88ms–1.2s depending on vocabulary size).

Model	Vocab	`YYYY-MM-DD`	`datetime`	`datetime.ms`	`UUID v4`
Mistral 7B v0.3	32K	1,859$\times$	626$\times$	532$\times$	2$\times$
GPT-2	50K	77$\times$	41$\times$	31$\times$	4$\times$
OLMo 3 7B	100K	135$\times$	71$\times$	49$\times$	6$\times$
Mistral Small 3.1	131K	1,378$\times$	740$\times$	512$\times$	14$\times$
Qwen3-8B	151K	1,119$\times$	486$\times$	247$\times$	10$\times$
gpt-oss-20b	200K	171$\times$	91$\times$	66$\times$	6$\times$
Gemma3 12B	262K	7,939$\times$	1,614$\times$	1,328$\times$	6$\times$

The speedups range from 2$\times$ (UUID on 32K vocab) to 7,939$\times$ (date on 262K vocab). For date/time formats with small character classes (digits, separators), the char-position mask compiles in under 1ms while xgrammar pays 88ms–1.2s for DFA construction regardless of format simplicity. UUIDs show smaller speedups because hexadecimal character classes (16 valid characters) produce more candidate tokens per position. These results confirm that constraint-aware dispatch, matching specialized engines to constraint structure, yields large speedups whenever the constraint has exploitable regularity, not just for finite sets.

L Practitioner Guidance

The accuracy-validity tradeoff depends on K, model capability, and error tolerance. At small K ($\leq$59) with strong models, PE validity exceeds 95% and can surpass trie accuracy; but even here, weaker models (Mistral: 42.5%, DeepSeek R1: 0.2% PE validity) already benefit from trie enforcement. At moderate K (76–150), trie strict typically matches or exceeds PE accuracy while guaranteeing validity, though PE can retain an accuracy edge on strong models (gpt-oss CLINC150: PE 79.7% vs. trie strict 77.0%) at the cost of 89% validity. At large K ($\geq$1,000), PE validity approaches 0%. We recommend: (1) PE when $K < 50$, the model is strong, and retries are acceptable; (2) think+trie when validity must be 100% or $K > 50$; (3) trie-only when latency is critical.

When to use which backend. For the constrained decoding backend itself (independent of prompting strategy):

Scenario	Recommended	Rationale
$K < 500$, one-shot	XGrammar	Fastest total time (Table 3)
$K < 500$, schema diversity	LLGuidance	Near-zero compilation
$K \geq 500$, repeated	Trie	Amortized; 75× faster masking
$K \geq 500$, one-shot	LLG or Trie	LLG: compilation; Trie: masking
Batch ($B \geq 32$)	Trie	CPU masking bottleneck (Table 20)
$K > 10{,}000$	Trie + hierarchical	Masking scales with K

M LLGuidance Comparison

To directly compare against LLGuidance (Geng et al., 2025), we benchmark its Python library (v1.6.1) on the same hardware and tokenizer (Qwen3-8B, 151K vocabulary) used for our main experiments. We measure compilation time (grammar construction + first mask computation) and per-step mask computation time for enum schemas with $K \in \{10, 100, 1{,}000, 5{,}000, 10{,}000\}$ tool-like names with realistic prefix structure. Compilation is averaged over 5 runs; per-step masking over 200 runs.

Table 19: Compilation time and per-step mask computation: LLGuidance vs. XGrammar vs. Trie Automaton (Qwen3-8B, 151K vocabulary). LLGuidance achieves the fastest compilation via lazy Earley parsing, but its per-step masking is 110–215× slower than the trie's precomputed lookups.

	K=10	K=100	K=1K	K=5K	K=10K
Compilation time (seconds)					
LLGuidance	**.001**	**.001**	**.003**	**.011**	**.024**
Trie (ours)	*.030*	*.031*	*.033*	*.037*	*.040*
XGrammar	.003	.015	.075	.150	.239
Per-step mask computation (µs)					
Trie (ours)	**0.65**	**0.65**	**0.65**	**0.65**	**0.65**
XGrammar	*9.5*	*5.4*	*5.8*	*5.9*	*5.9*
LLGuidance	121	73	85	96	141

LLGuidance's lazy Earley parser achieves near-zero compilation cost (0.6ms at $K = 100$, 24ms at $K = 10{,}000$), outperforming both XGrammar and our trie on this dimension. However, its per-step mask computation (73–141μs) is 110–215× slower than the trie automaton's precomputed lookups (0.65μs) and 9–24× slower than XGrammar (5–10μs). This confirms the theoretical prediction: LLGuidance must traverse the full vocabulary trie at each step ($\mathcal{O}(V)$), while our precomputed masks reduce this to $\mathcal{O}(|\text{valid}[s_t]|)$. For a typical enum generation requiring 5–10 decoding steps, the per-step advantage dominates: the trie's total masking cost is $\sim$5μs versus LLGuidance's $\sim$500μs, a 100× difference that compounds in batch serving. The three approaches occupy distinct points in the compilation-vs-masking tradeoff: LLGuidance minimizes compilation, XGrammar balances both, and the trie automaton minimizes per-step cost through precomputation.

M.1 Batch Serving Throughput

The per-step masking cost differences above are measured for a single request. In production batch serving, the GPU forward pass is shared across B concurrent requests (one batched matrix multiply), but masking runs per-request on CPU since each request occupies a different decoding state. Table 20 shows the measured total CPU masking time per batch-step as B grows.

At $B = 128$, LLGuidance's per-step masking (3.7ms) consumes over a third of the GPU forward pass time, directly reducing serving throughput. XGrammar's 783μs (7.8%) is also significant. The trie's 10μs (0.1%) is negligible at any batch size. These results are K-independent for the trie: at $K = 10{,}000$, the trie still measures 10.3μs at $B = 128$, while XGrammar and LLGuidance show similar costs (781μs and 3,671μs respectively). This

Table 20: Measured batch masking cost (μs per batch-step) on Qwen3-8B (151K vocabulary), $K = 1{,}000$ tool names. A typical GPU forward pass takes $\sim$10ms; the "% fwd" column shows masking cost as a fraction of this. At $B = 128$, LLGuidance masking consumes 37% of the forward pass, becoming the throughput bottleneck.

	Trie (ours)		XGrammar		LLGuidance	
B	μs	% fwd	μs	% fwd	μs	% fwd
1	0.2	0.00	5.4	0.05	31	0.31
32	2.4	0.02	170	1.70	893	8.93
64	4.9	0.05	367	3.67	1,827	18.3
128	10	0.10	783	7.83	3,716	37.2

explains why per-step masking cost, not compilation, is the binding constraint for serving throughput at scale.