ZEROGR: CONSTRAINT-DRIVEN EXECUTION FOR GENERATIVE RECOMMENDATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Constrained decoding is essential for generative recommendation, ensuring that generated semantic identifiers correspond to valid catalog items. At large catalog scales, existing approaches incur substantial overhead in constraint lookup and index storage. Even when constraint lookup is accelerated, applying constraints only after output projection wastes computation on tokens that cannot lead to valid recommendations. We present ZeroGR, a constraint-driven inference system that uses catalog information before projection to restrict scoring to valid continuations. ZeroGR represents catalog constraints as heterogeneous per-step mappings that directly expose valid next tokens and their successor states, avoiding repeated prefix traversal and explicit successor storage. A parallel streaming construction method builds these mappings from sorted semantic identifiers while overlapping CPU processing with transfer to the GPU. Specialized GPU kernels schedule valid projections through a compact work queue and reuse hidden vectors across candidates, avoiding both invalid projections and full candidate compaction. On a real-world recommendation dataset, ZeroGR reduces projection-and-selection latency by 34.8% compared with representative constrained-decoding approaches. Deployed on a production recommendation platform serving 400 million daily active users, ZeroGR reduces index memory usage by 66.2% compared with STATIC and end-to-end latency by up to 61.0% compared with the previously deployed CPU Trie serving approach.

1 INTRODUCTION

Generative retrieval produces item identifiers through autoregressive decoding (De Cao et al., 2021; Tay et al., 2022). A semantic identifier (SID) represents an item as a short sequence of discrete tokens (Rajput et al., 2023; Singh et al., 2024). During inference, beam search generates candidate SIDs one token at a time (Rajput et al., 2023; Deng et al., 2025). Arbitrary token sequences, however, need not correspond to items in the catalog. Constrained decoding is therefore essential: it restricts each partial SID to continuations that can reach a valid item (Zheng et al., 2024a; Su et al., 2026b). Enforcing this requirement at every decoding step makes constraint processing part of the latency and memory budget of an online recommendation service.

Existing methods typically enforce these constraints through an index that identifies valid continuations for each active prefix (De Cao et al., 2021; Bevilacqua et al., 2022; Willard & Louf, 2023). Repeated prefix traversal incurs lookup work at every step, while materializing transitions and successor states consumes memory alongside model weights and attention caches (Kwon et al., 2023). Accelerator-resident sparse indexes reduce lookup overhead (Su et al., 2026b), but a conventional decoding pipeline still applies constraints after output projection, which maps each beam's hidden vector to scores over the full vocabulary (Willard & Louf, 2023; Dong et al., 2025). Thus, accelerating constraint lookup alone leaves computation on infeasible continuations in the inference path.

Our design builds on an observation from real-world recommendation workloads, whose datasets are collected from the online services of industrial generative models, including OneSearch (Chen et al., 2026), OneReason (OneRec Team, 2026), and OneRec (Deng et al., 2025). For a specific catalog, the valid continuations of an active prefix are determined before model scores are computed. When token probabilities are normalized over each prefix's valid continuations, excluded tokens contribute neither to normalization nor to candidate selection. Our key insight is to use catalog constraints to

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.

determine the projection work itself. Moving constraint lookup ahead of output projection lets the system compute only the scores needed by that decoding rule.

Realizing this opportunity requires coordinating constraint representation with GPU execution. Valid continuations vary across prefixes and decoding steps, so the index must expose candidate sets directly while keeping its storage practical for large catalogs. These irregular candidate sets also complicate scheduling: projecting padded slots wastes computation, whereas compacting every candidate introduces additional data movement and synchronization (Billeter et al., 2009). Moreover, fewer projections do not automatically imply lower latency, since regular dense computation can reuse model weights more effectively (Williams et al., 2009; Gale et al., 2020). The system must separate the valid candidate set from the physical choices used to store and process it.

We present ZeroGR, a constraint-driven inference system that uses catalog structure to organize output-head execution. ZeroGR represents catalog constraints as heterogeneous per-step mappings from a constraint state to valid next tokens and their successor states. Each beam carries its state into the next step, allowing direct candidate lookup without retraversing the prefix. The representation derives successor states from transition order and omits outgoing entries for completed SIDs, reducing storage without sacrificing direct enumeration. A parallel streaming construction method builds these mappings from sorted SIDs, overlapping CPU processing with transfer to the GPU through bounded staging buffers.

For online execution, ZeroGR separates candidate storage from projection scheduling. It retains directly addressable candidate buffers while constructing a compact queue containing only valid projection tasks. Specialized GPU kernels reuse each beam's hidden vector across candidate projections and write logits back to their original slots, avoiding both invalid projections and full candidate compaction.

Our evaluation on a real-world recommendation dataset shows that ZeroGR reduces projection-and-selection latency by 34.8% compared with representative constrained-decoding approaches. Further comparisons isolate the storage savings from implicit successor states and the scheduling benefits of the compact work queue, while identifying conditions under which dense execution remains preferable. ZeroGR is deployed on a production recommendation platform serving 400 million daily active users, where it reduces index memory usage by 66.2% relative to STATIC and end-to-end latency by up to 61.0% relative to the previously deployed CPU Trie serving approach. This work makes three contributions:

- We analyze real-world recommendation workloads and identify projection redundancy that catalog constraints reveal before token scoring, motivating constraint-driven execution.
- Guided by this insight, we design ZeroGR with a compact per-step constraint index, parallel streaming construction, and GPU execution that schedules valid projections independently of candidate storage.
- We implement and deploy ZeroGR in production, and demonstrate a 34.8% reduction in projection-and-selection latency over representative constrained-decoding approaches on real-world recommendation data.

2 Background and Motivation

2.1 Semantic Identifier Decoding

SID-based generative recommendation represents each item i by a sequence $\mathrm{SID}(i) = (s_1^{(i)}, \ldots, s_L^{(i)})$ drawn from position-specific codebooks $\mathcal{V}_1, \ldots, \mathcal{V}_L$ (Rajput et al., 2023), typically learned by residual quantization of item embeddings (Zeghidour et al., 2022; Lee et al., 2022; Deng et al., 2025). Given user context x, an autoregressive decoder predicts the codes sequentially. The resulting sequence is resolved to one or more item identifiers by a separate catalog lookup; SID collisions and item-level deduplication are properties of this resolution stage. We optimize the inference path for a fixed model and a fixed catalog snapshot.

At step ℓ, let $A_\ell \le B$ be the number of active beams for one request, where B is the beam width. The root has $A_1 = 1$; subsequent steps usually have B active beams. For beam b with prefix p_b, the decoder produces $h_{\ell,b} \in \mathbb{R}^d$. A dense output head computes $z_{\ell,b,v} = \langle h_{\ell,b}, W_v \rangle + \beta_v$ for each token

v in the model output vocabulary $\mathcal{V}$. Some models have separate per-position heads; others share a global head whose rows include all position-specific codebooks. In our Search implementation, the shared vocabulary contains 18,432 rows, while a step uses only its assigned code band. This distinction matters because reducing a full-vocabulary head to a position-specific band and reducing a band to prefix-valid candidates remove different sources of work.

2.2 Catalog Constraints and Score Semantics

Let $\mathcal{S} = \{\mathrm{SID}(i) : i \in \mathcal{I}\}$ be the distinct SID sequences in the catalog. For a prefix $p = (s_1, \ldots, s_{\ell-1})$, its legal next tokens are

$$\mathrm{Next}_\ell(p) = \{v \in \mathcal{V}_\ell : \exists\, \mathbf{s} \in \mathcal{S},\ \mathbf{s}_{1:\ell-1} = p,\ s_\ell = v\}. \tag{1}$$

All returned finite-score prefixes must have at least one completion in $\mathcal{S}$. Post-generation filtering checks this condition only after decoding, so it can return fewer items than requested (Rajput et al., 2023). Prefix-constrained decoding instead removes illegal extensions before beam selection, as in trie-based entity retrieval, FM-index-based passage retrieval, and SID-based recommendation (De Cao et al., 2021; Bevilacqua et al., 2022; Zheng et al., 2024a).

Constraint validity alone does not specify how candidates are ranked; different ways of combining constraints with model probabilities induce different output distributions (Park et al., 2024). The integrated implementation uses *locally constrained normalization*: for $C_b = \mathrm{Next}_\ell(p_b)$ and parent score a_b,

$$s_{b,v} = a_b + z_{b,v} - \log \sum_{u \in C_b} \exp z_{b,u}, \qquad v \in C_b. \tag{2}$$

This is the result of setting illegal logits to $-\infty$ before applying log-softmax (Bevilacqua et al., 2022; Willard & Louf, 2023), and matches the GPU-mask serving baseline. The beam search retains the top B finite scores across the union of all parents' legal extensions for each request. Normalizing over the full vocabulary before masking instead retains illegal-token probability mass in a parent-dependent denominator and can change cross-parent ranking. The score contract therefore determines whether those logits may be omitted. The standalone microbenchmarks use unnormalized scores $a_b + z_{b,v}$ to isolate projection and selection; we distinguish them from normalized serving throughout the evaluation.

2.3 Constraint-Exposed Redundancy

Define the number of valid candidate instances and their position-local density as

$$M_\ell = \sum_{b=1}^{A_\ell} |\,\mathrm{Next}_\ell(p_b)|, \qquad \rho_\ell = \frac{M_\ell}{A_\ell |\mathcal{V}_\ell|}. \tag{3}$$

A token appearing under two parents contributes two candidate instances, since its hidden states and scores can differ. For a shared global output head, we additionally use $\rho_\ell^{\mathrm{global}} = M_\ell/(A_\ell|\mathcal{V}|)$. Position-only band restriction removes work outside $\mathcal{V}_\ell$; prefix-dependent selection further removes illegal entries within that band.

Constraints expose work before scores are available. Figure 1 illustrates that many projected entries cannot extend an active prefix to any catalog item. Their infeasibility follows from the catalog, regardless of whether the model would assign them high or low scores. A well-trained model that seldom selects invalid outputs can therefore still spend substantial computation producing invalid-token logits. When the score contract is Equation 2, those logits contribute neither selected candidates nor normalization terms.

Sparsity does not determine kernel efficiency. Candidate density varies across steps, active prefixes, and catalogs. A selective kernel reduces the number of dot products but accesses a different set of weight rows for each beam (Gale et al., 2020). A dense multiplication can reuse weight tiles across many beams and exploit regular matrix hardware (Williams et al., 2009; Markidis et al., 2018). The relevant tradeoff is consequently between irregular computation on M_ℓ instances and regular computation on A_ℓ times the step's token-band width, including the cost of lookup, materialization, and selection.

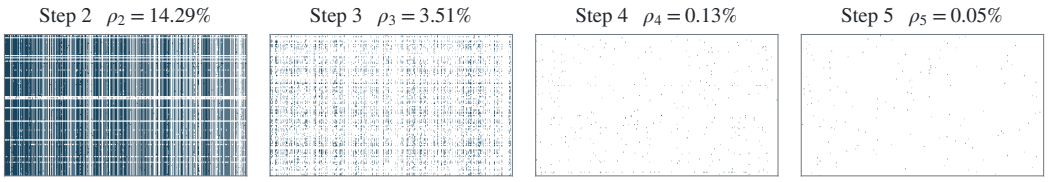


Figure 1: Legal state–token transitions in Search, SID steps 2–5, with $B = 128$. The displayed local densities are 14.29%, 3.51%, 0.13%, and 0.05%. Prefixes are selected using deterministic random scores (seed 1234), not a trained model or the microbenchmark's projection scores. Columns are max-pooled for display; densities use the full matrices.

3 Constraint-Driven Execution

ZeroGR uses catalog constraints before output projection. Each beam carries a constraint state, which directly identifies its legal next tokens; only those tokens enter projection, normalization, and selection. The decoder and its attention cache are unchanged, and selected candidates retain their parent references for cache reordering. The execution interface separates an immutable catalog mapping from request-dependent beam states, candidate buffers, and projection tasks. Selective projection is integrated into serving; band-dense scoring uses the same legal-candidate interface in a separately timed prototype.

3.1 Constraint Index: Per-Step Sparse Mappings

For a catalog prefix $p = (s_1, \ldots, s_{\ell-1})$, let $q_\ell(p)$ identify the prefix at step ℓ. The mapping

$$\mathcal{M}_\ell(q_\ell(p)) = \{(v, q_{\ell+1}(p \circ v)) : v \in \mathrm{Next}_\ell(p)\} \tag{4}$$

enumerates exactly its legal extensions. Tokens for each state occupy a contiguous segment in a step-local array, located by two row offsets. The same token under different prefixes occupies separate entries, while illegal extensions occupy none. Each step also records its maximum segment length and token range for workspace sizing and band scoring.

The array position of an entry is also its next-step state: if (p, v) occupies position j, set $q_{\ell+1}(p \circ v) = j$. Each extended prefix has one predecessor, so no explicit successor array is needed. A selected beam carries that position into the next step and locates its legal continuations without a prefix traversal. The final step stores completion tokens but no terminal rows. Figure 2 illustrates this layout.

In the example, the empty-prefix mapping first assigns states 0 and 1 to prefixes (1) and (3). At step 2, state 1 addresses tokens $\{2, 5\}$; token 5 occupies position 3, which becomes the state for shared prefix $(3, 5)$ at step 3. Token 2 appears under both first-step prefixes and therefore has two distinct entries. This direct state assignment removes a field from every transition without changing which next tokens can be enumerated.

For N distinct fixed-length SIDs and E distinct nonempty prefixes, the index has $E - N + 1$ nonterminal states and E transitions. With w_o-byte offsets and w_t-byte tokens, its size is

$$S_{\mathrm{index}} = w_o(E - N + 2) + w_t E, \tag{5}$$

up to $O(L)$ step metadata. The evaluated layout uses four-byte offsets and two-byte tokens.

The representation retains one entry per distinct prefix–token extension, rather than one per input SID row. Consequently, repeated SIDs and common prefixes do not multiply transition storage. Row offsets are stored only for states that can be queried again; complete SIDs contribute final tokens but no terminal rows. Both choices matter at catalog scale, where the last layer can dominate the number of prefixes.

3.2 Streaming Index Construction

The index is rebuilt when a catalog snapshot changes. Given lexicographically sorted SIDs in host memory, adjacent-row prefix boundaries identify precisely where new transitions begin. A parallel first pass counts transitions at each step and prefix-sums the counts, assigning disjoint output ranges to

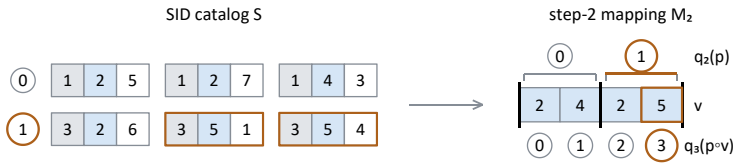


Figure 2: Step-2 mapping for six three-token SIDs. Each state addresses one token segment; an entry's position supplies its next-step state.

worker chunks. A second pass emits row offsets and tokens into those ranges. Each chunk compares its first row with the preceding global row, so prefixes spanning a chunk boundary are counted once; duplicate SIDs add no transitions.

Instead of materializing a complete host index, the builder alternates bounded pinned buffers: CPU workers fill one buffer while the previous buffer transfers to its final GPU range. Construction takes $O(N_{\text{in}}L + E)$ work after sorting, with bounded staging storage beyond the input catalog. The device performs transfers but no construction kernel. The new index and SID resolver are published together only after all transfers finish, so requests use one consistent snapshot.

The first pass gives each chunk its exact destination range, allowing CPU workers to emit transitions without synchronizing on individual states. A staging buffer is reused only after its earlier transfer completes. The measured builder uses two 16 MiB buffers; its output row offsets and tokens are bytewise equal to the general-purpose builder's, so construction changes neither lookup behavior nor score semantics.

3.3 Work-Queue Execution over the Index

For A_ℓ active beams and maximum catalog branching factor D_ℓ, candidate buffers reserve $P_\ell = A_\ell D_\ell$ slots even though only M_ℓ are valid. Each legal slot carries its token and successor state, while its row identifies the parent beam. This padded layout gives direct addresses and predictable workspace allocation without requiring projection over unused slots. Lookup reads each beam's two segment offsets, fills only legal slots, and initializes other scores to $-\infty$. It partitions each legal segment into tasks of at most $T = 64$ candidates; each beam reserves its queue entries with one atomic increment, yielding

$$Q_\ell = \sum_{b=1}^{A_\ell} \left\lceil \frac{|C_b|}{T} \right\rceil \tag{6}$$

projection tasks. A task records its beam, starting slot, and count. Thus projection follows valid work while candidate storage remains directly addressable; no global candidate scan or compaction pass is required.

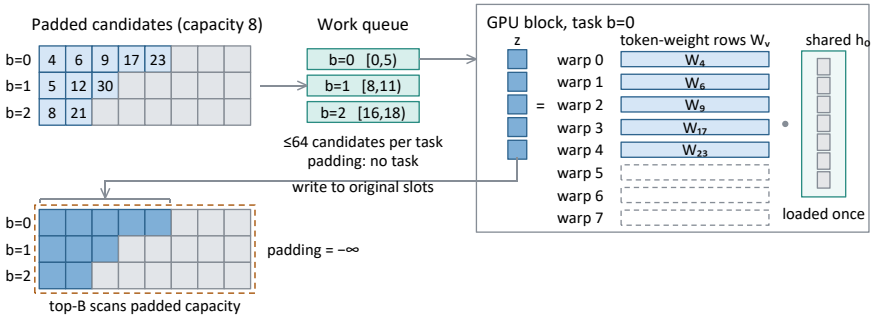


Figure 3: Padded candidate rows and a compact projection queue. Each task scores valid entries and writes logits back to their original slots.

GPU blocks process the queue by striding over its tasks. A block cooperatively loads the parent hidden vector once; eight warps then gather output-weight rows and compute candidate dot products in waves, writing logits to their original padded slots. This amortizes hidden-vector loading across a task but can still load the same weight row for different parents. The bounded grid follows the number of valid tasks rather than the much larger padded candidate capacity illustrated in Figure 3.

After projection, a per-parent constrained log-softmax uses only legal logits and adds each parent's accumulated score. Top-B selects candidates across parents within each request. The selected slot identifies the token and successor state, while its parent reference reorders the decoder's attention cache; otherwise the next hidden state could correspond to a different prefix than the constraint lookup. At the final SID step, the successor state is not dereferenced. Selection still scans padded capacity, so the queue removes invalid projection work rather than all padding overhead.

This schedule changes where logits are computed, not the mathematical legal set. With identical legal logits, a shared score contract and tie order, and consistent cache lineage, dense masking and indexed scoring select the same beams. Finite-precision differences near a top-B boundary can still change an ordering; the implementation therefore checks legal-candidate arithmetic separately from serving throughput.

3.4 Dense and Selective Scoring

Selective projection performs approximately $2M_\ell d$ operations for M_ℓ legal candidate instances. A band-dense projection over R_ℓ contiguous token rows performs approximately $2A_\ell R_\ell d$ operations, but can reuse weights across beams and exploit regular matrix hardware. Ignoring caching, selective scoring loads roughly $M_\ell d$ weight elements, while an ideally reused band-dense operation loads $R_\ell d$. Their traffic ratio, M_ℓ / R_ℓ, shows why fewer dot products need not mean lower latency at a wide frontier; it is not a universal dispatch threshold.

Here R_ℓ is the actual projected token interval, which can be narrower than a nominal code band. Writing band density as $\rho_\ell = M_\ell/(A_\ell R_\ell)$ makes the traffic ratio $A_\ell \rho_\ell$: even a sparse band can reload many weight rows when many beams choose overlapping tokens. Backend choice must therefore account for beam count and data reuse as well as candidate density.

The band-dense path projects the contiguous interval containing the step's legal tokens, then gathers only scores at legal positions. Tokens inside that interval but outside a parent's legal set never enter its normalization or selection; normalizing the full band first would change the ranking rule. Serving uses selective scoring, while the separately timed prototype freezes a per-step band/selective plan before measurement.

4 Evaluation

We evaluate serving performance, selective projection, and index footprint on industrial SID catalogs. Serving results and isolated projection benchmarks have different timing boundaries and score contracts, so their speedups are reported separately.

4.1 Experimental Setup

Search contains about one billion distinct five-token SIDs and an 18,432-row shared vocabulary; Watch contains about four billion distinct three-token SIDs and an 8,192-row vocabulary. Several catalog rows can share an SID, so SID counts do not imply unique-item counts. The Search serving experiment uses a BART-style model with hidden width 1,024, beam width 128, trained weights, and production requests on one data center GPU. A single service stream processes batches of 8–64 requests; reported throughput is the median of positive five-second windows after index initialization. The selective sweep lasts 105 seconds per batch and reuses earlier baseline runs, so the medians do not provide independent-run confidence intervals.

The serving baselines are the previously deployed CPU Trie, a GPU-mask implementation that retains full-vocabulary projection, and STATIC in eager and compiled-graph forms (Su et al., 2026b). GPU mask and ZeroGR use the locally constrained normalization of Equation 2. STATIC normalizes before masking, so its numbers compare complete systems with different ranking contracts. Isolated

projection benchmarks use matched candidate sets and selection code. They measure lookup, projection, selection, and state propagation, excluding the Transformer, softmax, cache management, and service loop; their numbers are not interchangeable with serving throughput.

The data center GPU microbenchmark holds hidden width 4,096 and beam width 128, with the same fixed hidden vector for each beam at a given depth; the dense path uses pedantic cuBLAS without TF32 Tensor Cores. The separate consumer GPU experiment holds both backends to the same lower-precision operands and hidden width 1,024. Primary full-path totals and fine-grained stage profiles come from separate timing trials and are not added together.

4.2 Serving Performance

On Search, ZeroGR reaches 497.6–934.4 requests/s and exceeds GPU mask by 16.0–24.5% across the measured batches (Figure 4). Against CPU Trie at batch 64, the throughput ratio corresponds to 61.0% lower effective batch latency under the single-stream protocol. On the synthetic catalog, the improvement over GPU mask is 16.4–25.0%.

STATIC graph reaches 272.0–780.8 requests/s, placing ZeroGR 19.7–82.9% higher. This is a system comparison: the compiled STATIC adapter incurs index-constant copies and uses a different normalization order. The selective serving sweep records 118,376 successful requests with no failed requests, overflows, or NaNs. An oracle checks legal candidate scores and constrained normalization, but ordered beam agreement and recommendation quality are not yet measured.

The batch-one oracle checks all five SID depths against CPU projections and legal-set log-softmax; its maximum observed absolute error is 0.004820, below the configured 0.25 tolerance. This validates local arithmetic and candidate membership under the tested requests, while finite-precision differences can still change beam ordering near a score tie.

4.3 Where Serving Time Changes

At batch 32, the decode-side residual of ZeroGR is 0.369 of CPU Trie and 29.2% lower than GPU mask (Figure 5). Its post-root step latencies are below GPU mask at every measured step. These reductions cannot all be attributed to the output head: mean CPU item-resolution time also changes across the reused captures and accounts for about 84% of their enqueue-time difference. Subtracting that component yields a 1.40% reduction in the decode-side residual; the per-step medians have a different timing boundary and cannot be summed to explain it.

4.4 Separating Candidate Storage from Scheduling

On Search, the work queue has 34.8% lower latency than per-candidate scheduling and 27.2% lower latency than full candidate compaction (Figure 6). Its latency is 76.0% lower than full-vocabulary dense execution in the same FP32 benchmark. This last comparison includes both position-only and prefix-dependent work reduction; a static band-dense control is not available on that device.

The measured stage profile helps explain the work-queue gain: lookup and task generation take 42.994 μs, compared with 211.307 μs for the degree/scan/emit path used by full compaction. Grouped scoring also falls from 435.341 to 362.203 μs. The queue therefore makes projection work compact while retaining directly addressed candidate rows; its selection stage still pays for their padded capacity.

The queue also improves the older selective path on Watch, but remains $7.64\times$ slower than dense execution: 90.15% of the second-step band is legal. Candidate sparsity alone therefore does not determine the faster backend. In a separate same-precision prototype on a consumer GPU, band-dense scoring is $2.50\times$ faster at Search step 2 despite $7.63\times$ more dot products; a frozen per-step plan reduces total benchmark latency by 12.15%. This policy is not part of the serving implementation.

4.5 When Band-Dense Scoring Wins

The prototype holds candidate sets, padded buffers, numerical format, and selection fixed. It computes the contiguous step band, then gathers only legal scores before selection. At the root, selective scoring is faster because only one beam is active; step 2 favors band-dense despite just 13.11% of its nominal

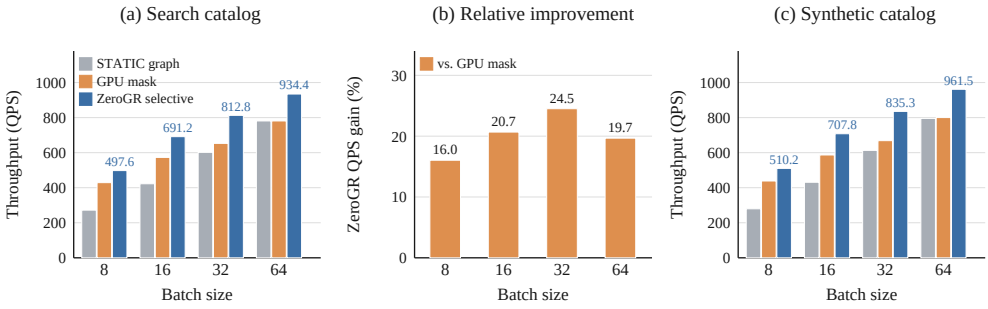


Figure 4: Measured serving throughput on Search and a synthetic catalog. All methods use the same request protocol; the plotted values are medians of steady-state windows.

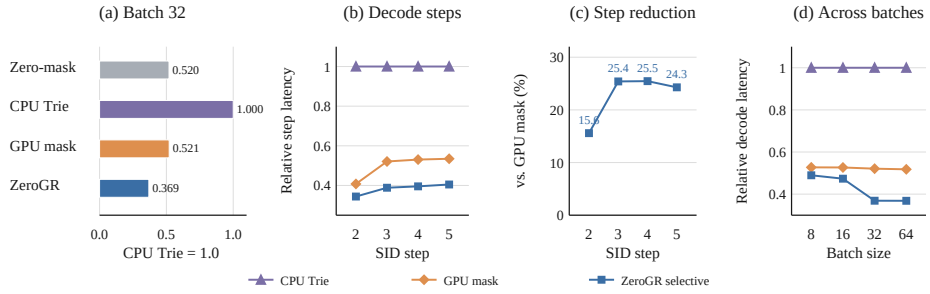


Figure 5: Search decode-side latency: batch-32 residual relative to CPU Trie, post-root steps, each step's reduction versus GPU mask, and the cross-batch trend. The residual is not an isolated GPU timer; step medians are not additive.

band being legal. The frozen plan chooses band-dense at the root and step 2, then selective scoring. Its root choice is suboptimal, and the prototype is measured separately from serving; it demonstrates the value of a per-step choice rather than an optimal deployed policy.

4.6 Index Footprint and Scope

The Search index occupies about 8 GiB. Relative to an otherwise identical layout with explicit successors, implicit successor positions save 49.0% of index bytes. The evaluated index uses 66.2% fewer bytes than STATIC's packed index; this comparison also includes narrower token and row fields. A minimal DAWG is 20.0% smaller than ZeroGR but requires explicit successors and more involved construction. In serving, sampled process-memory peaks are about 8–9 GiB lower than STATIC graph across batches 8–64; those peaks also include model and runtime allocations.

Omitting terminal rows is particularly important for Watch, whose stored state count drops by 98.5% while final-step tokens remain in the transition array. The measured process-memory difference includes allocator pools, activations, and model weights, so it is corroboration rather than an isolated byte count for the index.

On the Search catalog, streaming construction from sorted host SIDs to the device-resident index has a three-run median of 0.658 s versus 61.022 s for the earlier general-purpose builder, a 98.9% reduction. The two outputs are bytewise equal; sorting and catalog ingestion precede the timed construction. This result concerns index refresh rather than steady-state request latency.

The measurements cover fixed catalog snapshots, fixed-length SIDs, and beam width 128. The serving comparisons reuse earlier baseline runs and lack independent-run intervals. The isolated FP32 dense comparison uses a full vocabulary and does not establish a target-device band-dense crossover. Finally, execution checks do not establish SID validity rates, returned-item counts, or recommendation quality on held-out requests.

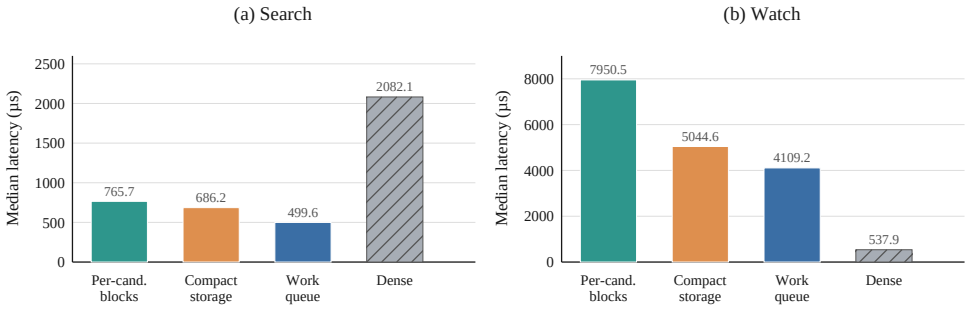


Figure 6: Matched projection-and-selection benchmarks on Search and Watch. Bars compare per-candidate blocks, full candidate compaction, the work queue, and full-vocabulary dense execution; lower is better.

5 Related Work

Constrained decoding. Prefix-constrained decoding restricts generation to valid outputs through tries, FM-indexes, or finite-state machines, and has been applied to entity and passage retrieval, structured generation, and SID-based recommendation (De Cao et al., 2021; Bevilacqua et al., 2022; Willard & Louf, 2023; Zheng et al., 2024a). Subsequent systems reduce the per-step cost of computing legal tokens, using compressed automata and cached grammar state for general grammars (Zheng et al., 2024b; Dong et al., 2025; Li et al., 2026) and accelerator-resident sparse indexes for SID catalogs (Su et al., 2026b). These pipelines typically apply the legal set to logits produced by full-vocabulary projection; ZeroGR instead uses it to schedule the projection itself. Succinct tries and minimal acyclic automata reduce index storage by eliminating pointers or merging suffix-equivalent states (Zhang et al., 2018; Daciuk et al., 2000); our index instead favors direct enumeration and derives successor states from transition order.

Output-layer reduction. Vocabulary selection and adaptive softmax reduce output-layer cost through approximate token shortlists or frequency-based vocabulary partitions (L'Hostis et al., 2016; Grave et al., 2017), and sampled dense–dense matrix multiplication evaluates a dense product only at prescribed sparse positions (Gale et al., 2020). ZeroGR instead derives its sparse support exactly from catalog feasibility without modifying the model, and specializes the product schedule to per-prefix segments and beam lineage.

Other inference optimizations. Generative recommendation and LLM inference are also accelerated by speculative decoding (Leviathan et al., 2023; Lin et al., 2025), quantization (Su et al., 2026a), and attention and KV-cache optimizations (Dao et al., 2022; Kwon et al., 2023). These techniques operate outside the output head and are orthogonal to ZeroGR.

6 Conclusion

We presented ZeroGR, a constraint-driven inference system for constrained decoding in generative recommendation. ZeroGR applies catalog constraints before output projection, combining a compact per-step constraint index, parallel streaming construction, and GPU kernels that compute only valid projections. It reduces latency relative to representative constrained-decoding approaches on real-world recommendation data, and its deployment on a production platform serving 400 million daily active users lowers both memory usage and end-to-end latency.

AI USE STATEMENT

Generative AI was used to adapt the manuscript's LaTeX formatting to the ICLR 2027 template. All technical content, experiments, and conclusions are the authors' own, and the authors take full responsibility for them.

REFERENCES

Michele Bevilacqua, Giuseppe Ottaviano, Patrick Lewis, Wen tau Yih, Sebastian Riedel, and Fabio Petroni. Autoregressive search engines: Generating substrings as document identifiers. In *Advances in Neural Information Processing Systems*, volume 35, 2022. URL https://arxiv.org/abs/2204.10628.

Markus Billeter, Ola Olsson, and Ulf Assarsson. Efficient stream compaction on wide SIMD many-core architectures. In *Proceedings of the Conference on High Performance Graphics*, pp. 159–166, 2009. doi: 10.1145/1572769.1572795.

Ben Chen, Xian Guo, Siyuan Wang, Zihan Liang, Yufei Ma, Yue Lv, Chenyi Lei, Yuqing Ding, Wenwu Ou, Han Li, and Kun Gai. OneSearch: A preliminary exploration of the unified end-to-end generative framework for e-commerce search. In *Proceedings of the 43rd International Conference on Machine Learning*, 2026. URL https://arxiv.org/abs/2509.03236.

Jan Daciuk, Stoyan Mihov, Bruce W. Watson, and Richard E. Watson. Incremental construction of minimal acyclic finite state automata. *Computational Linguistics*, 26(1):3–16, 2000. URL https://aclanthology.org/J00-1002/.

Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Re. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In *Advances in Neural Information Processing Systems*, volume 35, 2022. URL https://arxiv.org/abs/2205.14135.

Nicola De Cao, Gautier Izacard, Sebastian Riedel, and Fabio Petroni. Autoregressive entity retrieval. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=5k8F6UU39V.

Jiaxin Deng, Shiyao Wang, Kuo Cai, Lejian Ren, Qigen Hu, Weifeng Ding, Qiang Luo, and Guorui Zhou. OneRec: Unifying retrieve and rank with generative recommender and iterative preference alignment. *arXiv preprint arXiv:2502.18965*, 2025. URL https://arxiv.org/abs/2502.18965.

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

Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen. Sparse GPU kernels for deep learning. In *SC: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2020. URL https://arxiv.org/abs/2006.10901.

Edouard Grave, Armand Joulin, Moustapha Cisse, David Grangier, and Herve Jegou. Efficient softmax approximation for GPUs. In *Proceedings of the 34th International Conference on Machine Learning*, volume 70, pp. 1302–1310. PMLR, 2017. URL https://proceedings.mlr.press/v70/grave17a.html.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023. URL https://arxiv.org/abs/2309.06180.

Doyup Lee, Chiheon Kim, Saehoon Kim, Minsu Cho, and Wook-Shin Han. Autoregressive image generation using residual quantization. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 11523–11532, 2022. URL https://arxiv.org/abs/2203.01941.

Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202, pp. 19274–19286. PMLR, 2023. URL https://proceedings.mlr.press/v202/leviathan23a.html.

Gurvan L'Hostis, David Grangier, and Michael Auli. Vocabulary selection strategies for neural machine translation. *arXiv preprint arXiv:1610.00072*, 2016. URL https://arxiv.org/abs/1610.00072.

Linzhang Li, Yixin Dong, Guanjie Wang, Ziyi Xu, Alexander Jiang, and Tianqi Chen. XGrammar-2: Dynamic and efficient structured generation engine for agentic LLMs. *arXiv preprint arXiv:2601.04426*, 2026. URL https://arxiv.org/abs/2601.04426.

Xinyu Lin, Chaoqun Yang, Wenjie Wang, Yongqi Li, Cunxiao Du, Fuli Feng, See-Kiong Ng, and Tat-Seng Chua. Efficient inference for large language model-based generative recommendation. In *International Conference on Learning Representations*, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/e4bf5c3245fd92a4554a16af9803b757-Abstract-Conference.html.

Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S. Vetter. NVIDIA tensor core programmability, performance & precision. In *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 522–531, 2018. doi: 10.1109/IPDPSW.2018.00091.

OneRec Team. OneReason technical report. *arXiv preprint arXiv:2606.06260*, 2026. URL https://arxiv.org/abs/2606.06260.

Kanghee Park, Jiayu Wang, Taylor Berg-Kirkpatrick, Nadia Polikarpova, and Loris D'Antoni. Grammar-aligned decoding. In *Advances in Neural Information Processing Systems*, volume 37, 2024. URL https://arxiv.org/abs/2405.21047.

Shashank Rajput, Nikhil Mehta, Anima Singh, Raghunandan H. Keshavan, Trung Vu, Lukasz Heldt, Lichan Hong, Yi Tay, Vinh Q. Tran, Jonah Samost, Maciej Kula, Ed H. Chi, and Maheswaran Sathiamoorthy. Recommender systems with generative retrieval. In *Proceedings of the 37th International Conference on Neural Information Processing Systems (NeurIPS)*, NeurIPS '23, Red Hook, NY, USA, 2023. Curran Associates Inc.

Anima Singh, Trung Vu, Nikhil Mehta, Raghunandan Keshavan, Maheswaran Sathiamoorthy, Yilin Zheng, Lichan Hong, Lukasz Heldt, Li Wei, Devansh Tandon, Ed H. Chi, and Xinyang Yi. Better generalization with semantic IDs: A case study in ranking for recommendations. In *Proceedings of the 18th ACM Conference on Recommender Systems*, 2024. doi: 10.1145/3640457.3688190. URL https://arxiv.org/abs/2306.08121.

Yi Su, Xinchen Luo, Hongtao Cheng, Ziteng Shu, Yunfeng Zhao, Fangyu Zhang, Jiaqiang Liu, Xiao Liang, Yiwu Liu, and Ruiming Tang. Quantized inference for OneRec-V2. *arXiv preprint arXiv:2603.11486*, 2026a. URL https://arxiv.org/abs/2603.11486.

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*, 2026b. URL https://arxiv.org/abs/2602.22647.

Yi Tay, Vinh Q. Tran, Mostafa Dehghani, Jianmo Ni, Dara Bahri, Harsh Mehta, Zhen Qin, Kai Hui, Zhe Zhao, Jai Gupta, Tal Schuster, William W. Cohen, and Donald Metzler. Transformer memory as a differentiable search index. In *Advances in Neural Information Processing Systems*, volume 35, 2022. URL https://arxiv.org/abs/2202.06991.

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

Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009. doi: 10.1145/1498765.1498785.

Neil Zeghidour, Alejandro Luebs, Ahmed Omran, Jan Skoglund, and Marco Tagliasacchi. SoundStream: An end-to-end neural audio codec. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 30:495–507, 2022. doi: 10.1109/TASLP.2021.3129994.

Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G. Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. SuRF: Practical range query filtering with fast succinct tries. In *Proceedings of the ACM International Conference on Management of Data*, 2018. doi: 10.1145/3183713.3196931. URL https://db.cs.cmu.edu/papers/2018/mod601-zhangA-hm.pdf.

Bowen Zheng, Yupeng Hou, Hongyu Lu, Yu Chen, Wayne Xin Zhao, Ming Chen, and Ji-Rong Wen. Adapting large language models by integrating collaborative semantics for recommendation. In *Proceedings of the 40th IEEE International Conference on Data Engineering (ICDE)*, pp. 1435–1448, 2024a. URL https://arxiv.org/abs/2311.09049.

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 *Advances in Neural Information Processing Systems*, volume 37, 2024b. URL https://arxiv.org/abs/2312.07104.

A Score Equivalence and Its Boundaries

A.1 Index soundness and completeness

For a fixed catalog snapshot, each nonterminal state corresponds to a distinct prefix. Its segment contains exactly the tokens that occur immediately after that prefix in at least one catalog SID. For a transition at position j in the step-local token array, the extended prefix has state identifier j in the next step. The assignment is injective because every nonempty prefix has one immediate predecessor and one final token, and it is surjective onto the next step's states because every such state is a catalog prefix. Removing terminal rows preserves the last-step transition relation: completed SIDs require no further outgoing lookup.

Starting from the empty prefix, induction shows that every finite-score extension produced by these mappings is a prefix of at least one SID in the catalog. After L steps, every such complete sequence belongs to $\mathcal{S}$. This property does not promise B distinct items: fewer than B finite candidates may exist, distinct SIDs may resolve to overlapping item sets, and an item resolver may apply additional policies. If fewer than B finite candidates exist, only those candidates should be treated as active; padded $-\infty$ slots are not valid outputs.

A.2 Beam-search equivalence

Lemma 1 (Constrained beam-search equivalence)**.** *Fix the input context, model, catalog snapshot, beam width, and a common tie-breaking order on logical (parent beam, token identifier) pairs, independent of their physical slot addresses. Assume identical candidate logits, exact arithmetic, and consistent parent-cache propagation. Replacing dense projection and pre-normalization masking with index enumeration and either scoring backend preserves every selected beam, score, and SID under Equation 2. The same result holds for unnormalized additive logits when both implementations use that score contract.*

At the first step, both implementations have the same empty prefix, decoder state, and parent score. Assume they enter step ℓ with identical selected prefixes, scores, and decoder-cache lineage. The fixed decoder then supplies the same hidden vector for each corresponding beam. The dense implementation sets logits outside C_b to $-\infty$. Index soundness and completeness give the selective implementation exactly C_b, with no missing or additional finite-score candidate. By assumption, the two implementations produce identical logits on this set. Hence

$$\sum_{u\in\mathcal{V}} \exp(\widetilde{z}_{b,u}) = \sum_{u\in C_b} \exp(z_{b,u}), \qquad \widetilde{z}_{b,u} = \begin{cases} z_{b,u} & u \in C_b, \\ -\infty & u \notin C_b. \end{cases}$$

Their log-softmax scores and accumulated candidate scores therefore agree. With the same tie-breaking order on logical (parent, token) pairs, top-B selects the same candidate identities for each request, even when their physical indices differ. The successor state, emitted token, and parent reference of each selected candidate also agree. Reordering the cache with that parent reference establishes the induction hypothesis for step $\ell + 1$. The terminal selections and SIDs agree after the last step. For unnormalized logits, the proof is unchanged except that the normalization equality is unnecessary. For band-dense execution, all legal rows are included in the band and the gather removes other rows before normalization, so the same argument applies.

A.3 Why full-vocabulary normalization is different

Suppose two parent beams have accumulated scores 0 and -0.1. Each has two legal next tokens with logits $(0, 0)$ and one illegal token. Let the illegal-token logits be 100 for the first parent and -100 for the second. With mask-before-normalization, the first parent's two legal extensions both score $-\log 2$, and the second parent's both score $-0.1-\log 2$. A beam of width two selects both extensions of the first parent. With full-vocabulary normalization followed by masking, the first parent's legal extensions instead score approximately -100, while the second parent's remain approximately $-0.1 - \log 2$. The beam now selects both extensions of the second parent. Discarding illegal logits has changed the selected prefixes even though the legal sets are identical.

More generally, converting between these score rules requires the parent-dependent correction

$$\Delta_b = \log \sum_{u \in \mathcal{V}} e^{z_{b,u}} - \log \sum_{u \in C_b} e^{z_{b,u}}.$$

An exact implementation of the full-vocabulary-normalized contract cannot simply omit its first normalizer. Likewise, per-step constrained normalization should not be described as globally conditioning the original model on eventual membership in $\mathcal{S}$: that distribution would depend on the probability of all feasible future completions.

A.4 Forced continuations under local normalization

If $|C_b| = 1$, Equation 2 assigns its sole extension score a_b, regardless of the next-token logit. A forced continuation therefore incurs no additional likelihood penalty under this contract, whereas full-vocabulary normalization can penalize it. This degree-dependent behavior motivates a separate quality comparison between scoring contracts; implementation equivalence alone cannot justify the choice of contract. It also permits omitting projection and normalization for singleton rows while retaining decoder/cache operations needed by later steps. The measured implementation does not exploit this shortcut. Additive-logit microbenchmarks still require the singleton logit, so their necessary work differs.

A.5 Finite-precision ranking

Different accumulation orders and output casts can perturb scores near a beam boundary even when the real-arithmetic functions agree. The total error below includes score-storage rounding, normalization, and parent-score accumulation, not just dot-product error. In the integrated FP16 score tensor, adjacent representable magnitudes in $[8, 16)$ and $[16, 32)$ are separated by 0.0078125 and 0.015625, respectively, so rounding can create exact ties. Ascending token order inside each candidate segment does not by itself guarantee that two top-B kernels use the same tie policy. If every candidate's total score error at a fixed step is bounded by ϵ and the reference gap between the Bth and $(B+1)$th candidates exceeds 2ϵ, the selected set is unchanged at that step. This follows because the worst adverse perturbation reduces the boundary gap by at most 2ϵ. The recorded candidate errors do not supply a global bound on accumulated beam-score error over trained requests, nor establish the required margin at every step. A trained-model regression must therefore inspect both numerical differences and the resulting beam lineage.

B Additional Experimental Detail

B.1 Catalog scale

Table 1: Catalog and index scale. Counts are rounded; states exclude terminal leaves, while transitions include the final SID token.

Property	Search	Watch
Distinct SIDs	∼1B	∼4B
SID length / $\lvert\mathcal{V}\rvert$	5 / 18,432	3 / 8,192
Stored nonterminal states	∼1B	∼60M
Transitions	∼2B	∼4B
Maximum branching by step	7,916/826/263/351/236	8,192/8,173/7,804

B.2 Dense/selective crossover

B.3 Index layouts

B.4 Protocol separation

The selective serving sweep runs for 105 seconds per request batch and retains 9–11 positive five-second throughput windows per configuration. GPU mask has 8–11 such windows. Effective batch

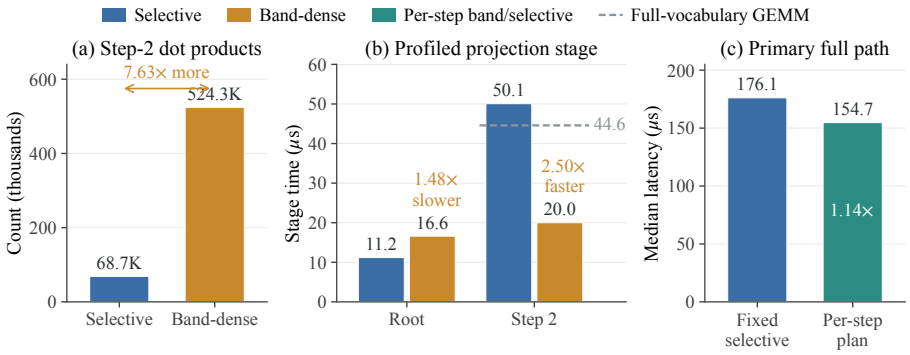


Figure 7: Same-precision Search microbenchmark. At step 2, band-dense scores 7.63× more dot products but runs 2.50× faster; the frozen per-step plan lowers full-path latency by 12.15%.

Table 2: Search device index footprint, normalized to the evaluated implicit-successor layout. Values describe index storage only.

Representation	Relative index size
Explicit successors, all state rows	2.40×
Implicit successors, all state rows	1.44×
Explicit successors, terminal rows omitted	1.96×
Implicit successors, terminal rows omitted	**1.00**×
STATIC packed index	2.96×
Minimal DAWG with explicit successors	0.80×

Table 3: Measurement contracts used in this draft. Distinct rows are not interchangeable speedup regimes.

	Data center GPU serving	Data center GPU microbench.	Consumer GPU microbench.
Input	Production requests; trained weights	Resident hidden vectors/weights	Resident hidden vectors/weights
Hidden width	1,024	4,096	1,024
Operands	FP16 output head	FP32	FP16
Candidate score storage	FP16	FP32	FP32
Score	Local constrained log-softmax	Additive logits	Additive logits
Beam width	128 per request	128	128
Statistics	Median of positive 5-s QPS windows	20 trials, 5 warmups	100 trials, 20 warmups
Includes decoder	Yes	No	No
Includes final item resolver	Yes	No	No
Includes index build	No	No	No

latency is derived as $1000 \times$ batch/QPS, not measured as a request-latency percentile. The data center GPU microbenchmark uses five warmups and 20 timed trials; the corrected consumer-GPU experiment uses 20 warmups and 100 timed trials. The historical FP16 comparison on the data center GPU mixes control configurations and is not used for the same-precision backend claim.

Within each microbenchmark platform, the compared variants share one driver and CUDA toolkit version. The serving profile uses Nsight Systems and an eight-second capture; the selective capture contains 193 enqueues, of which 192 have complete four-step boundaries. The three custom kernel classes each appear 963 times, rather than the ideal 965 for 193 complete five-step enqueues, because the capture cuts across two boundary invocations. The per-step plots use complete enqueue samples. The serving graph's root expansion is omitted from the four post-root step panels, not from throughput measurements.

The older BART configuration document lists a 19,000-row output head, while the later constrained integration documents an 18,432-row head. The latter is the relevant vocabulary for the reported constrained comparisons. The finalized experiment package should include the exact engine configuration and checkpoint shapes so that this historical configuration change cannot be mistaken for an optimization effect.

B.5 Full serving baseline tables

Table 4: Recorded requests/s. STATIC eager and STATIC graph use the published representation under different runtimes; these are system comparisons. The unconstrained reference neither enforces catalog validity nor performs the same final CPU resolution work.

Implementation	Batch 8	Batch 16	Batch 32	Batch 64
CPU trie	343.2	320.0	403.2	364.8
STATIC eager	67.05	73.07	75.88	77.30
STATIC graph	272.0	422.4	601.6	780.8
GPU mask	428.8	572.8	652.8	780.8
ZeroGR selective	497.6	691.2	812.8	934.4
Unconstrained reference	517.6	755.2	940.8	1088.0

The unconstrained reference returns through an empty final item-resolution path because it does not load the CPU trie. It is therefore not a quality-equivalent service baseline or a lower bound on every component of constrained execution. In the STATIC graph profile, three large index-related copy kernels precede the nominal output-head interval and consume 12.90 ms across the four decoder steps. They must not be attributed to the Transformer merely because the compiler schedules them inside that time interval. The total constraint-related time reported for this adapter is 14.096 ms, compared with 6.712 ms for GPU masking, excluding the output-head projection itself.

B.6 Backend calibration and timing statistics

The band-dense prototype calibrates candidate counts in an untimed pass and freezes the backend selected for each step. It chooses band-dense when the global candidate density satisfies

$$\frac{M_\ell}{A_\ell|\mathcal{V}|} \geq \frac{1}{B}, \qquad \text{equivalently} \qquad M_\ell \geq \frac{A_\ell|\mathcal{V}|}{B}.$$

It does not time both backends, and this threshold is not derived from the traffic ratio M_ℓ/R_ℓ in Section 3.4. The use of configured beam width B fails to distinguish the root's single active beam from later steps. The recorded root regression illustrates this limitation. A plan calibrated to fixed benchmark inputs is also not evidence of adaptation to changing online requests.

Both constrained paths use the same candidate sets, padded capacities, and top-B implementation. Their primary totals exclude profiling events. Per-stage values are obtained separately; variations in unchanged selection and state-propagation stages must not be credited to backend choice. The unconstrained reference does not enforce SID validity and does not load the index. Its dense projection stage is informative about regular computation, but its full-path timing does not provide a functionally equivalent constrained baseline.

Table 5: End-to-end requests/s and derived effective batch latency. Bold values are ZeroGR selective; the improvement columns use the corresponding batch's baseline.

Batch	GPU mask	ZeroGR	vs. mask	Batch ms
8	428.8	**497.6**	+16.04%	16.077
16	572.8	**691.2**	+20.67%	23.148
32	652.8	**812.8**	+24.51%	39.370
64	780.8	**934.4**	+19.67%	68.493

Table 6: Corrected same-precision Search comparison on the consumer GPU. All paths use FP16 operands, FP32 accumulation/scores, $d = 1024$, and $B = 128$. Confidence intervals accompany reported means across 100 timing trials; they do not represent cross-day variance.

Path	Mean $\pm$ 95% CI (μs)	Median (μs)
Fixed selective	176.139 ± 0.233	176.128
Per-step band/selective	155.239 ± 0.279	154.720
Unconstrained full vocabulary	304.484 ± 0.583	304.128

B.7 Representation accounting and integer ranges

Search stores one four-byte row offset per nonterminal state plus a sentinel, and one two-byte token per transition; the total equals Equation 5. An explicit successor array on the same transitions adds four bytes per transition, or 0.96$\times$ the evaluated index size. The evaluated STATIC layout is 2.96$\times$ the evaluated index size and already has approximately the nonterminal state count. Its packed transitions use two four-byte fields, with approximately eight bytes of index storage per state. On the native transition set, omitting explicit successors, narrowing tokens from four to two bytes, and narrowing per-state storage from approximately eight to four bytes save 32.5%, 16.2%, and 17.5% of the STATIC index size, respectively. Small remaining differences arise from the layouts' different entry counts. The overall STATIC comparison therefore includes width choices; terminal-row omission is not credited as an additional distinguishing saving against that baseline.

The native global successor is the edge offset plus one. Search's largest identifier fits signed 32-bit storage, whereas Watch's final transition identifiers exceed that range. The standalone layouts use unsigned 32-bit row offsets, and terminal identifiers are never dereferenced as state rows. A larger catalog would require wider offsets rather than wraparound. Index-array widths do not specify a framework's state-tensor type: the final TensorRT artifact must document state-tensor signedness and casts separately. Watch is not part of the reported TensorRT serving integration.

At request batches 8–64, 200-ms NVML samples record process-memory peaks that are about 8–9 GiB lower for ZeroGR than for STATIC graph. These differences describe sampled process capacity, not exact allocation-level index attribution.

B.8 Candidate counts and padding

Table 7: Search FP32 microbenchmark frontier on the data center GPU. Densities divide by the shared 18,432-token head and the number of active beams, not by candidate capacity or the local token band.

SID step	Active beams	Legal candidates	Padded capacity	Global density
1	1	7,916	7,916	42.9470%
2	128	71,189	105,728	3.0174%
3	128	9,487	33,664	0.4021%
4	128	273	44,928	0.0116%
5	128	144	30,208	0.0061%
Total		89,009	222,444	0.9413%

The capacity-to-valid ratio increases sharply in the last two steps. The work queue avoids assigning projection tasks to invalid slots, but the selection implementation retains those slots. Consequently,

the proportion of time spent on selection can increase after projection is optimized. The prototype with a task-aware small-candidate selection path is a separate exploratory implementation and is not included in the integrated ZeroGR throughput results.

B.9 CPU ORACLE RESULTS

Table 8: Integrated batch-one CPU oracle, including projection bias and per-parent constrained log-softmax. These are maximum observed candidate-score errors in this smoke run, not a bound for arbitrary trained inputs.

SID step	Active rows	Legal candidates	Tasks	Max. absolute error
1	1	7,916	124	0.004284
2	128	68,448	1,150	0.004820
3	128	1,134	137	0.002336
4	128	133	128	0.001627
5	128	128	128	0

At the last step of this input, each active row has one candidate, so its constrained log-softmax is zero. This explains the zero reported error without implying that all last-step logits are exactly equal between kernels. The oracle frontier differs from Table 7, since the serving model and microbenchmark generate different hidden vectors and select different prefixes. The visualization in Figure 1 uses yet another, explicitly identified random-score frontier.

C STREAMING CONSTRUCTION AND TIMING

C.1 STREAMING CONSTRUCTION ALGORITHM

Updating the SID assignment requires rebuilding the GPU-resident constraint index. ZeroGR constructs a separate index for each decoding step directly from the lexicographically ordered SID library in host memory. This organization preserves step-specific differences in state count, branching factor, and token range behind the common lookup interface of § 3.1.

Lexicographic order places SIDs sharing a prefix contiguously, so each adjacent-pair longest common prefix identifies all steps at which the latter SID introduces a new transition. In the first pass, the builder partitions the SID library into chunks, counts these transitions per step in parallel, and prefix-sums the counts. The resulting offsets give the exact index sizes and disjoint output ranges before any transition is emitted.

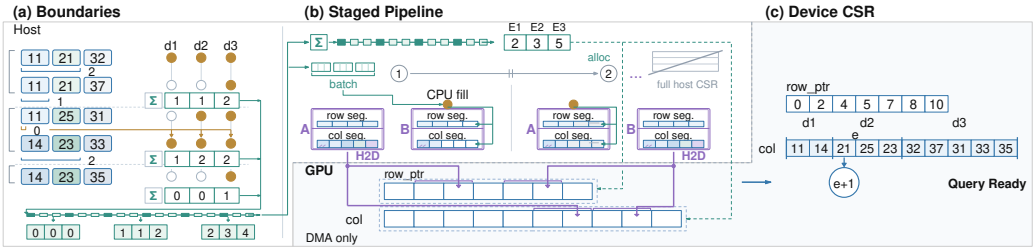


Figure 8: Streaming CPU construction of the per-step indexes. A parallel pass determines their exact layouts; a second pass alternates bounded index segments between two pinned buffers, overlapping CPU generation with H2D into the final device locations.

As Figure 8 shows, the second pass revisits the boundaries and emits each chunk's index segments directly into its assigned ranges; disjoint ranges avoid synchronization among CPU workers. The builder groups chunks into bounded transfer batches and alternates them between two pinned buffers, so that completed index segments in one buffer are transferred to the GPU while the CPU fills the other. The GPU performs allocation and DMA but no construction kernel; after the transfers complete, the staging buffers are released and only the query-ready per-step indexes remain. For N_{in} input rows

of length L and E emitted transitions, work is $O(N_{\text{in}}L + E)$; sorting and catalog ingestion precede the two passes. Additional host storage consists of two bounded staging buffers and $O(CL)$ chunk metadata for C chunks, beyond the resident input catalog. Duplicate rows introduce no transitions after their first occurrence, and each chunk compares its first row with the preceding global row so that a shared prefix crossing a chunk boundary is counted exactly once. Pinned-buffer reuse waits for its preceding copy to complete; publishing a refreshed index requires all copies to complete before requests use the new snapshot.

C.2 Construction timing

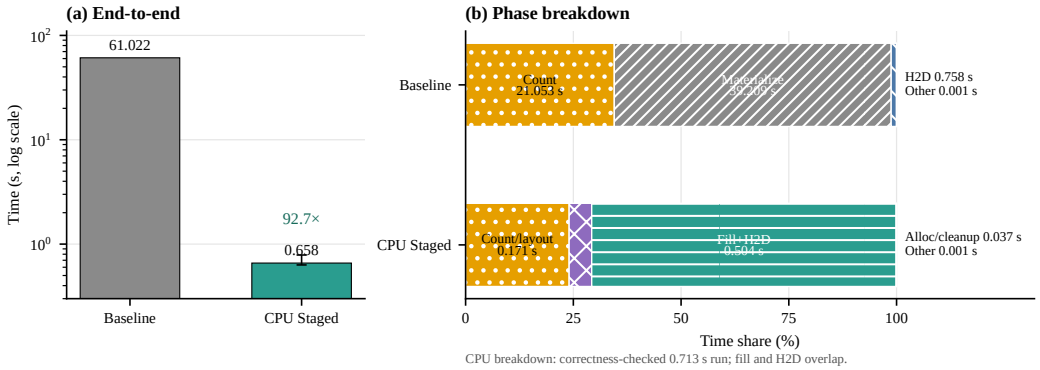


Figure 9: Construction time from sorted host-resident SIDs to a device-resident index. The baseline takes 61.022 s; streaming construction has a three-run median of 0.658 s (range 0.631–0.786 s).

The baseline breakdown attributes 21.053 s to level-wise counting, 39.209 s to general-purpose structure and per-SID state materialization, and 0.758 s to the final host-to-device transfer. The streaming breakdown records 0.171 s for boundary counting and layout, 0.504 s for overlapped filling and transfer, and 0.037 s for allocation and cleanup. The latter components sum to 0.712 s; the figure includes another 0.001 s of other work and labels the run as 0.713 s. The 0.713 s run is separate from the three-run median of 0.658 s and is not substituted for that median. The construction uses two 16 MiB staging buffers; the row-offset and token arrays are bytewise equal to the baseline output. The estimated bandwidth floor is 0.386 s.

D Completion Protocol for the Remaining Evidence

The draft contains two pending evidence groups. They can be completed without changing the paper's score contract or silently replacing any observed result with an estimate.

R1: trained-model correctness and recommendation quality. Use an identical checkpoint, tokenizer/SID assignment, catalog snapshot, and fixed held-out request set for GPU mask and selective execution. Record model and data fingerprints, the normalization order, tie policy, and any final item filters. Export every step's selected token, parent index, constraint state, candidate score, and final resolved item set for an audit subset. Report validity, requested versus returned distinct-item counts, exact ordered agreement, top-K item overlap, Recall@K, and NDCG@K. Define a common logical tie key and separately report agreement after grouping near-tied scores at a declared precision-dependent tolerance; do not replace the exact metric with the tolerant metric. Separate finite-precision boundary differences from score-contract differences and cache-lineage errors. No quality-preservation claim should be added on the basis of request success alone. In a separate dense-only experiment, compare local constrained normalization, full-vocabulary normalization followed by masking, and band normalization followed by masking on the same trained model. This tests the score contract's quality consequences without confounding them with a selective kernel.

R3: controlled serving repeats and hybrid coverage. First measure forced full-vocabulary dense, static band-dense, and selective projection on the target data center GPU with FP16 operands

and hidden width 1,024. Include a general sampled dense–dense multiplication baseline, and compare shared versus independently varying hidden vectors across beams. Any singleton-row shortcut must be evaluated under local constrained normalization and labeled as a new variant. Then repeat GPU mask, static band-dense, and selective execution in randomized order under one software/hardware configuration. Include integrated per-step execution if making a hybrid serving claim. Use independent process runs as the uncertainty unit; preserve within-run throughput windows for diagnosis. Use at least five independent runs per arm as an initial campaign, report all run values, and expand the campaign if intervals remain too wide to resolve the observed gap. Measure the complete GPU decoder interval including the root and report CPU item resolution separately. Report root-step regressions and calibration overhead, and retain the unmodified selective result as a control. Add trained-request candidate-density distributions to explain whether changes from synthetic inputs come from the model frontier or the execution backend.