LATLM: LINE-PARALLEL LANGUAGE MODELS WITH DEFERRED COMMITMENT

Anonymous authors
Paper under double-blind review

ABSTRACT

Line-parallel language models advance multiple lines concurrently for substantial speedups. However, frameworks like the Line-Coupled Language Model (LCLM) enforce a fixed commitment schedule across lines. Because dependencies vary within text, a fixed schedule cannot hold back a line whose prerequisites remain uncomputed, forcing premature commitments and cascading reasoning errors. To address this limitation, we introduce the *Lattice Language Model* (LATLM), making commitment schedules an adaptive decoding-time choice. Text occupies a 2D lattice where eligible frontier tokens are scored each round; a policy commits confident candidates while deferring unready ones for extended context. One forward pass supervises entire sampled trajectories via 2D causal masking, while an append-only cache retains exact states. At 482M parameters, we establish two findings. (1) LatLM matches a data- and capacity-matched autoregressive model on held-out text when decoded sequentially, while its adaptive parallel decoder achieves near-autoregressive accuracy on independent-line tasks using 25–41% as many decoding rounds. (2) When cross-line dependencies cause LCLM's fixed schedule to fail, deferred commitment substantially improves accuracy while retaining parallelism where dependencies permit. On mixed dependency chains, LatLM achieves 96.5% accuracy versus LCLM's 78.5%, using fewer than half the decoding rounds of the autoregressive baseline.

1 INTRODUCTION

Standard autoregressive decoding generates one token per forward pass, requiring a sequence of model evaluations as output length grows. Parallel generation methods aim to produce multiple tokens per step while maintaining output quality.

However, existing parallel decoders each pay somewhere. Masked and block diffusion denoise many positions at once (Sahoo et al., 2024; Nie et al., 2025; Arriola et al., 2025) but sample them independently within a step, and masked diffusion has no causal cache while block diffusion caches only completed blocks; multi-token and speculative methods (Gloeckle et al., 2024; Cai et al., 2024; Leviathan et al., 2023; Chen et al., 2023) draft ahead and are bounded by verification; insertion and non-autoregressive decoders (Stern et al., 2019; Gu et al., 2018) generate in flexible orders at the cost of consistency among concurrent tokens; and structured approaches (Ning et al., 2024; Liu et al., 2024; Jin et al., 2025) run predefined segments concurrently but cannot pause a segment that depends on another.

The shared difficulty is dependence. Tokens committed in one step are conditioned on the same context and cannot condition on one another, which is harmless when they carry little mutual information given that context and unreliable otherwise. With exact marginals, the forward-KL cost of independent parallel commitment equals the conditional total correlation of the committed tokens (Zhang et al., 2026). Learned marginals add a second error: when a token's own history is incomplete, its marginal can be confidently wrong. The Line-Coupled Language Model (LCLM; Li et al., 2026c) interleaves lines into one causal sequence and advances every active line by one token per step. This schedule computes more than one might expect: in our state-tracking task it solves every chain through 24 steps, restating each line's input before the result it restates has been written, so the state can only have come from the prompt. But where a line needs a result the model does not recover this way, the successor is written against a guess, and in the traces we examine the predecessor's

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.

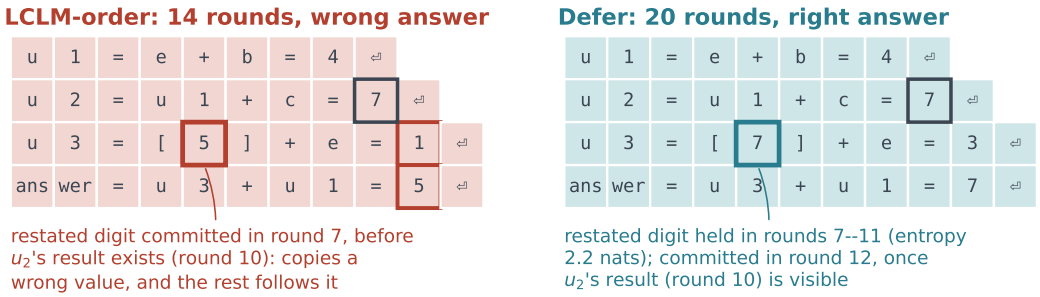


Figure 1: **Why a line sometimes has to wait: one problem, one checkpoint, two policies.** A mixed chain (Section 3.3) decoded by the same LatLM under the fixed line-parallel schedule (left) and the entropy gate (right); prompt rows omitted. Row u_2 reads u_1's result in its last digit; row u_3 restates u_2's result as a bracketed source digit (black frame: the result it needs). The fixed schedule reaches that token in round 7, three rounds before u_2's result exists, copies a wrong value, and every later token is consistent with it. The gate holds that one cell (entropy 2.2 nats) while the other rows proceed, and commits it once u_2's result is visible: 20 rounds and right, against 14 rounds and wrong; sequential decoding takes 41.

arithmetic then agrees with that guess rather than with its own operands. A decoder should be able to let such a line wait.

Our central question is: *Can one model support line-parallel generation while allowing individual commitments to be deferred?* Our answer is YES. We introduce the *Lattice Language Model* (LatLM; Figure 1) to provide this flexibility. At each round, a policy uses predictive entropy to select tokens for commitment. Several lines can advance together, or some can wait for additional context. Entropy is a heuristic for this choice: it does not determine whether a semantic dependency has been resolved, and a progress fallback commits one token if none passes the gate. The gate itself is not new: any-order models such as A3 already commit confidence-selected subsets of positions (Du et al., 2026). What LatLM adds is the object the gate runs on: a frontier that is legal by construction over variable-length lines, one residual stream per cell whose token enters only the key/value path, and a cache that stays exact along whatever trajectory the policy takes, so that holding a cell back costs one re-score and nothing else. The price is flexibility: a line can only wait for the lines above it.

Our contributions are the model and three findings about it:

- **A line-parallel language model with deferred commitment**. LatLM supports sequential, fixed line-parallel, and adaptive decoding with one set of weights. Its lattice structure allows individual frontier tokens to wait for additional context, while trajectory-based training supervises every cell in one forward pass and an append-only cache remains exact along each legal decoding trajectory.

- **Autoregressive modeling quality with adaptive parallel decoding**. Decoded sequentially, LatLM matches a data- and capacity-matched autoregressive model on held-out text (0.928 vs. 0.931 bits per byte), at approximately 1.3× the training compute per token. On independent-line tasks, adaptive decoding achieves near-autoregressive accuracy using 25–41% as many decoding rounds.

- **Improved accuracy over fixed line-parallel decoding under cross-line dependencies**. Deferred commitment recovers near-sequential accuracy where LCLM's fixed schedule fails, retaining parallelism when dependencies permit and falling back toward sequential decoding when necessary. On mixed dependency chains, LatLM achieves 96.5% accuracy versus LCLM's 78.5%, using 28.9 rounds versus the autoregressive baseline's 64.8.

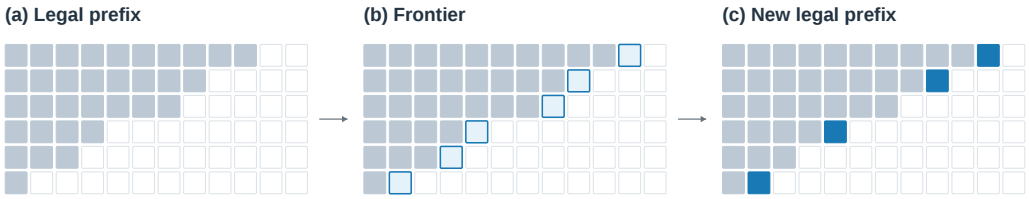


Figure 2: **One round of lattice generation.** (a) Gray cells form a legal prefix. (b) Blue outlines mark the six frontier positions. (c) Accepting four frontier tokens, shown in solid blue, produces a new legal prefix.

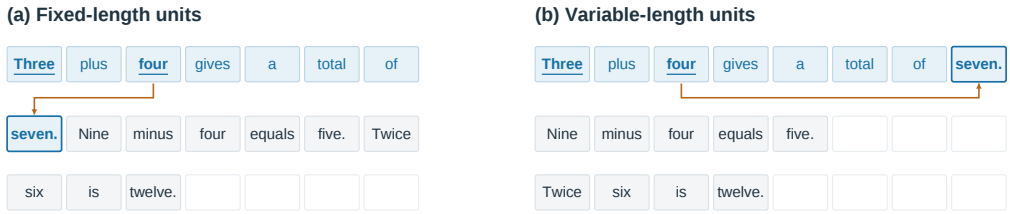


Figure 3: **Unit boundaries change where dependencies fall.** The same text is arranged into (a) fixed-length units of seven words and (b) variable-length units following sentence boundaries. Blue cells mark the first sentence; arrows highlight the dependency of its answer on an earlier operand. Fixed partitioning moves the answer to the next unit, whereas sentence-based partitioning keeps the calculation together. Cells show words for readability; actual token boundaries may differ.

2 Lattice Language Models

2.1 Generating across lines, with the option to wait

A document is a grid of token cells, one line per row and one token position per column. The committed cells form a *legal prefix* if every committed cell's left and upper neighbours are committed, wherever they exist: each line grows left to right, and a line cannot advance beyond the line above it (Figure 2(a)). The *frontier* is the set of uncommitted cells whose left and upper neighbours are committed, at most one per line (b). Eligibility is structural: a frontier cell near the start of one line may still need a result that has not yet appeared at the end of the line above.

In each round the model predicts a distribution at every frontier cell from the committed context, and a policy commits a subset (c). The rest stay open and are predicted again once more context exists. Tokens committed in the same round cannot condition on one another, so the choice of subset matters. Committing any subset of the frontier yields another legal prefix, and generation repeats predict, commit, extend; legal prefixes ordered by inclusion form a lattice (Appendix A).

2.2 Mapping text to the grid

Text units. Each row of the lattice holds a contiguous span of text, which we call a *unit*. Unit boundaries determine which tokens follow one another within a row and which can be generated in parallel across rows. We distinguish *fixed-length units*, cut to a prescribed block size, from *variable-length units* whose boundaries align with sentences or reasoning steps.

Variable-length units preserve dependencies. Fixed boundaries can artificially split dependencies, forcing the model to anticipate unproduced information (e.g., making "seven" eligible before "four" is committed in Figure 3a). We therefore use variable-length units aligned with natural reasoning steps (Figure 3b). This keeps an answer and its premises within the same row, leveraging the strict left-to-right constraint to ensure operands are committed first without introducing arbitrary cross-unit bottlenecks.

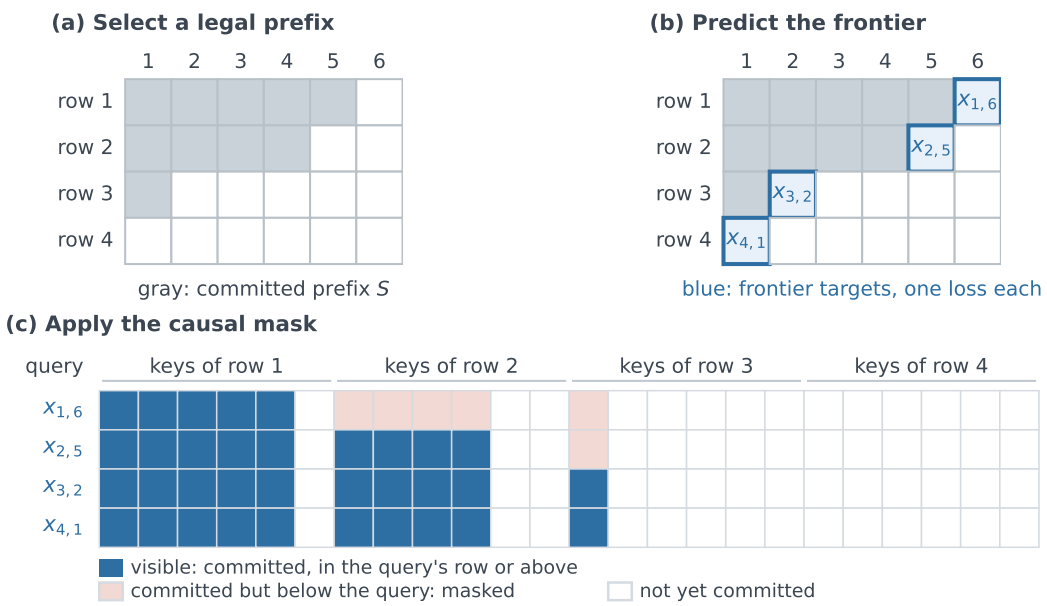


Figure 4: **Training from a legal prefix.** (a) Select a legal prefix, shown in gray. (b) Predict the frontier tokens and compute their cross-entropy losses. (c) The frontier-query rows of the attention mask (Eq. 2): a target attends to committed cells in its own row and the rows above it, so $x_{1,6}$ sees row 1 only and $x_{4,1}$ all ten committed cells; committed cells below a query are masked, and so are the other targets.

Two-dimensional positional encoding. To support parallel generation of variable-length units, we identify each cell directly by its 2D coordinates (r, c) and apply separate rotary positional encodings (Su et al., 2024) to the row and column axes. A cell's coordinates thus remain independent of preceding lengths and commitment rounds. Additional coordinate features are described in Appendix B.

Unit boundaries and termination. Each unit begins with an anchor and ends with a model-predicted <eol> token. Subsequent positions act as virtual padding omitted from network inputs. Units exceeding the maximum content capacity are simply wrapped into additional rows.

2.3 Training on legal prefixes

Training predicts frontier tokens from a partially completed document: select a legal prefix S of a reference, condition on its committed cells and the prompt, and apply a cross-entropy loss at every frontier cell (Figure 4(a,b)). The targets share one context but carry their own coordinates, and none may observe another. The committed set S fixes which cells are legal; the history h (cells with their commit rounds) fixes the cached states, so two histories with the same S need not predict alike. A prediction stream starts from a learned query embedding at its coordinates, without its token; at every layer the token x_v enters only the key/value path,

$$Q_v = W_Q z_v, \qquad K_v = W_K\big(z_v + e(x_v)\big), \qquad V_v = W_V\big(z_v + e(x_v)\big), \tag{1}$$

so a target cannot leak into its own stream and one residual stream per cell serves both roles (Appendix B).

Packing training contexts into one forward pass. To supervise every cell of a document at once we sample a *commit trajectory* $\tau = (A_1, \ldots, A_T)$, each A_t a nonempty subset of the current frontier, which assigns each cell v a round $t_\tau(v)$ and a history $h_\tau(v)$ of the prompt and the cells committed in earlier rounds. Cells are arranged in round order with their two-dimensional coordinates, and a query at u attends to v when

$$M_{uv} = \mathbf{1}\{t_\tau(v) < t_\tau(u)\}\ \mathbf{1}\{0 \le r(u) - r(v) < W\}, \tag{2}$$

where $r(\cdot)$ is the row and W a row window ($W = 6$ in pretraining, 32 or 64 in the task fine-tunes; the prompt rows are always within reach of the window as configured). The first factor is causal: a

cell sees only what was committed in earlier rounds, never its own token or a concurrent one. The second restricts attention to the cell's own row and the $W - 1$ rows above it, so a cell does not read rows below it even once they are committed. Each cell is supervised once, at the history immediately before its commitment, and not at the earlier histories in which a policy might have scored and rejected it; all rounds are computed in one forward pass, the rounds setting attention visibility, not separate passes.

The training objective is

$$\mathcal{L}(\theta) = \mathbb{E}_{x\sim\mathcal{D},\,\tau\sim\Pi(\cdot|x)}\left[\sum_{v\in V_{\text{target}}} w_v\left[-\log q_{\theta,v}(x_v \mid h_\tau(v))\right]\right]. \tag{3}$$

The expectation is realized as a weight-normalized mean over the cells of a minibatch, not a per-document unweighted sum. Targets include content and unit termination; unit-start targets have weight four, others one. Pretraining samples five schedules with equal probability: row-major, column-major, simultaneous frontier updates, random nonempty frontier subsets, and a serialized diagonal order (one cell at a time by increasing row plus column). Prompt cells take their rounds from the sampled schedule like any other cell, with their content loss masked (row-ending supervision is not) so the model cannot fit the rules' phrasing without reading them, whereas inference prefills the prompt. The mixture does not cover every history an adaptive policy may visit.

2.4 Decoding strategies

One set of weights supports three decoding strategies (Figure 5). They use the same eligibility rule but produce different commitment schedules: each round's choices determine the context and eligible positions in subsequent rounds. AR-order commits the first row-major frontier cell, finishing one line before starting the next. LCLM-order commits the first $\min(K, |F(S)|)$ frontier cells each round, advancing multiple lines on a fixed rule regardless of confidence. Defer scans the frontier in row-major order and commits the first up to K cells whose predictive entropy $H_{\theta,v}(h) = -\sum_a q_{\theta,v}(a \mid h)\log q_{\theta,v}(a \mid h)$ is at most a threshold η, so a line whose next token the model cannot yet predict with confidence waits while later lines take its place. If no candidate passes, the first row-major cell is committed so that every round makes progress. All commitments are irreversible (Algorithm 1).

LCLM-order is a fixed-width stand-in for native LCLM's schedule, which advances every active line by one token without a width cap and ends with a sequential answer tail. Native LCLM's interleaved causal sequence also lets an earlier line read a later line's committed tokens; our row-causal mask never allows this (Appendix D.7).

For these tasks, η is the q-quantile of entropies observed during LCLM-order decoding of 32 or 64 validation problems, without their answer labels; each task and width has its own calibration, and we report explicit quantiles rather than a "best lossless" point chosen on test accuracy. A quantile fixes a threshold, not the subsequent acceptance fraction or error rate. The quantile featured for a task is the loosest of the calibrated set at which no error is observed on the same calibration problems, the one use of their labels (Appendix D.6); zero errors on 32 problems is consistent with a 9% error rate at the 95% level, so this is a choice among a few quantiles, not a guarantee. Low entropy is a readiness heuristic, not a dependence detector: exact marginals bound within-round dependence by their summed entropies, but learned marginals can be confidently wrong (Appendix A).

2.5 Incremental execution and state reuse

Committed cells contribute keys and values to an append-only cache. Each round fuses the previous round's commits and the current frontier queries into one layerwise pass, querying at most one cell per open row, so the commit width changes how many queries are kept, not how many are computed; static buffers and CUDA-graph replay reduce overhead, as they do for the AR reference. Optionally, a frontier query's per-layer states are stored and, once it is committed, reused to build its keys and values without a second attention and feed-forward pass; this is exact in exact arithmetic because the stream already excludes its own token, though BF16 rounding can change trajectories (Appendix E). Exactness is relative to the realized trajectory: a committed cell's state depends on the history it was

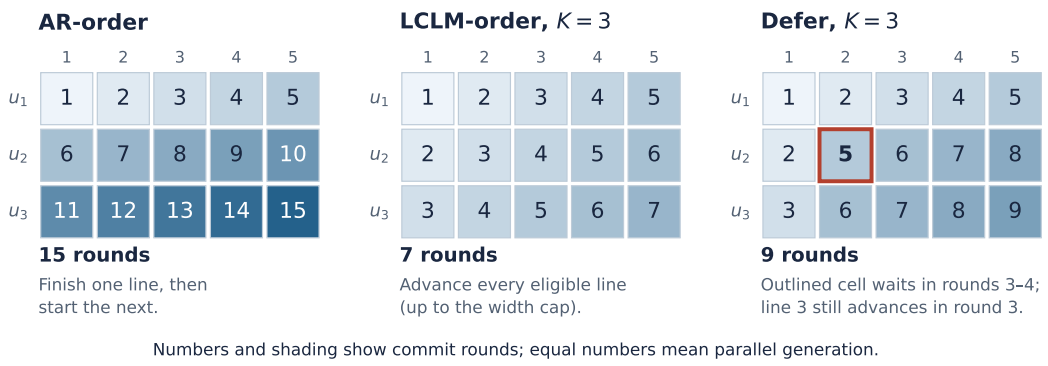


Figure 5: **Three decoding strategies on the same output layout.** Each cell shows its commitment round; equal numbers denote parallel generation. AR-ORDER completes lines in order. LCLM-ORDER ($K = 3$) advances all eligible lines. DEFER ($K = 3$) selectively waits: in this schematic, cell $(2, 2)$ exceeds the entropy threshold in rounds 3–4 and passes on re-scoring in round 5; all other candidates pass. Line 3 still advances in round 3 while that cell waits. Every schedule respects left and upper prerequisites from earlier rounds. The layout omits prompt and structural cells; the gate scores are illustrative.

Table 1: **Held-out content modeling at matched capacity and data.** 482M parameters, about 13.1B content tokens each, the same 1,024 windows; gap intervals are paired over documents; LCLM is scored under its own layout.

Model	BPB ↓ (95% CI)	Relative gap (95% CI)
Autoregressive reference	0.9313 [0.9180, 0.9454]	—
LatLM, AR-ORDER	0.9277 [0.9144, 0.9417]	−0.39% [−0.59, −0.19]
LatLM, parallel frontier	0.9479	+1.77% [+1.53, +2.02]
LCLM, own layout	0.9580 [0.9446, 0.9720]	+2.86% [+2.65, +3.06]

committed under, so the same committed set reached in two orders need not give the same predictions (Appendix A). Reuse removes repeated work, not the cost of scoring candidates the policy rejects.

3 EXPERIMENTS

One pretrained lattice, fine-tuned per task, is asked to do three things that usually pull apart: model text as well as an autoregressive model of the same size (Section 3.1), advance independent lines in parallel as LCLM does (Section 3.2), and stay correct where LCLM's fixed schedule fails (Section 3.3).

Setup. AR, native LCLM and LatLM have 482M parameters and were pretrained on the same ClimbMix split (Diao et al., 2025) with one tokenizer, each for about 13.1B content-token exposures. Native LCLM is our port of its interleaved layout and waterfall decoder. Each task gets a short fine-tune per model at the same budget; LatLM's policies share one checkpoint per task. Every model predicts its own row endings, gate thresholds are frozen on validation problems, and we count decoder rounds; request latency is a separate question (Section 3.4). Protocol details are in Appendix C.

3.1 SEQUENTIAL CONTENT MODELING

Does the lattice pay for its flexibility with worse language modeling? It does not. Decoded row by row it matches the autoregressive twin on held-out text, 0.928 against 0.931 bits per byte with a paired interval that excludes zero (Table 1), for about $1.3\times$ the training compute per token. The flexibility is paid for only when it is used: decoding every frontier cell at once costs the same weights about 2%, and native LCLM pays about 3% under its own layout, in both cases at the first tokens of each line, where a line starts before the one above has ended (Appendix D.10). Free-generation wall-clock measurements for the same model pair appear in Section 3.4.

Table 2: **Simple independent tasks.** Accuracy (%) and mean rounds on 400 fresh problems per task, own row endings; LatLM width six, one frozen threshold per task (records $q = 1.0$, list $q = 0.95$). Ratios report round-count reduction, not wall-clock speedup.

	Records			List batch		
Method	Acc.	Rounds	vs. AR	Acc.	Rounds	vs. AR
AR	100.0	130	1.00×	100.0	129	1.00×
Native LCLM	99.8	33	3.89×	94.8	34	3.74×
LatLM, AR-ORDER	100.0	130	1.00×	98.8	129	1.00×
LatLM, LCLM-ORDER	100.0	33	3.94×	77.5	33	3.96×
LatLM, DEFER	100.0	33	3.93×	98.2	53	2.44×

3.2 SIMPLE TASKS: KEEPING LINE PARALLELISM

Records and list batch have independent lines: records renders prompt-specified records as sentences, list batch performs one small addition per line. Here the option to wait should cost nothing, and on records it does not: the gate finishes in LCLM's round count, a quarter of AR's, at full accuracy (Table 2). List batch shows the trade the gate makes when lines are independent but individually harder. Native LCLM keeps its rounds and loses five points; the gate spends half again as many rounds and stays within a point of sequential decoding. Where it is not confident it spends rounds rather than accuracy, which is the behaviour a fixed schedule cannot have.

3.3 DEPENDENT COMPUTATIONS UNDER FIXED AND ADAPTIVE SCHEDULES

We test if the option to wait recovers accuracy where LCLM's fixed schedule fails. We evaluate on mixed chains (where dependent cells are scattered, allowing some parallelism) and branching traces (where every row depends on the last, severely limiting parallelism). All tasks are scored on full-trace accuracy.

Dependence scattered through the output. Mixed chains interleave independent, dependent, and restating rows (which copy a previous result mid-trace). The models diverge strongly here (Table 3). LCLM reaches restated inputs before the previous row writes them, losing 21% of traces. The autoregressive model (AR) achieves 100% accuracy but pays maximum rounds (64.8). The entropy gate, however, achieves 96.5% accuracy in 28.9 rounds and 98.5% in 33.2, about half the AR round count. Contrasting LatLM's own fixed schedule (29.0%) with its gated schedule explicitly isolates this scheduling benefit. The gate succeeds by selectively holding most prematurely eligible restated digits until their inputs arrive, while allowing independent rows to advance. This accuracy stems from smart selection, not simply taking more rounds: randomly deferring cells or matching the gate's per-round counts with a fixed row-major order both fail (Appendix D.6).

Budget and density. Increased training budgets (2× or 4×) help LCLM recover shallow dependencies, but deep dependencies still fail; the gate makes few errors at any tested depth by waiting instead of recomputing (Appendix D.9). Furthermore, as dependence density increases from 30% to 100% of rows, a gated fine-tune automatically adjusts its rounds from one-third to one-half of AR's to maintain near-sequential accuracy. At the highest densities, one of two LatLM seeds has no error-free calibration quantile. Conversely, LCLM's accuracy collapses. Even a single checkpoint with a frozen threshold adapts gracefully to varying densities (Appendix D.8).

Chains where every row waits. When every row sequentially depends on the previous one, the critical path spans the entire chain. In branching traces, successors must agree with the chosen operation of the row above. LCLM's accuracy drops rapidly as branches increase, while LatLM with DEFER remains accurate (Figure 6). Similarly, in 32-step state tracking, LCLM fails at deeper steps because its waterfall must recompute running states from the prompt (Appendix D.2). LatLM with DEFER stays close to sequential accuracy on both tasks, though it sacrifices nearly all parallelism to do so—decoding about one cell per round in strict dependency order. While an oracle scheduler could still achieve 60–80% of sequential rounds by parallelizing templated row prefixes, the gate successfully secures the necessary serial fallback that a fixed schedule cannot offer.

Table 3: **Mixed chains: every policy on one LatLM checkpoint, and the two baselines, on the same 200 fresh problems.** Full-trace accuracy (%) and mean rounds. "Answer tail" holds only the last row; "row-major at the gate's counts" replays the featured gate's per-round counts with row-major cells. Bold: the loosest quantile with no error on the 32 calibration problems.

Policy	Acc.	Rounds
AR	100.0	64.8
Native LCLM	78.5	25.6
Native LCLM, 24,000 updates	92.0	25.6
LatLM, AR-ORDER	99.5	64.8
LatLM, LCLM-ORDER	29.0	22.2
LatLM, LCLM-ORDER + answer tail	29.0	29.4
LatLM, random veto 0.5	27.0	38.1
LatLM, row-major at the gate's counts	33.5	31.5
LatLM, DEFER, $q = 0.9$	98.5	33.2
LatLM, DEFER, $q = 0.95$	**96.5**	**28.9**
LatLM, DEFER, $q = 0.99$	29.0	22.2

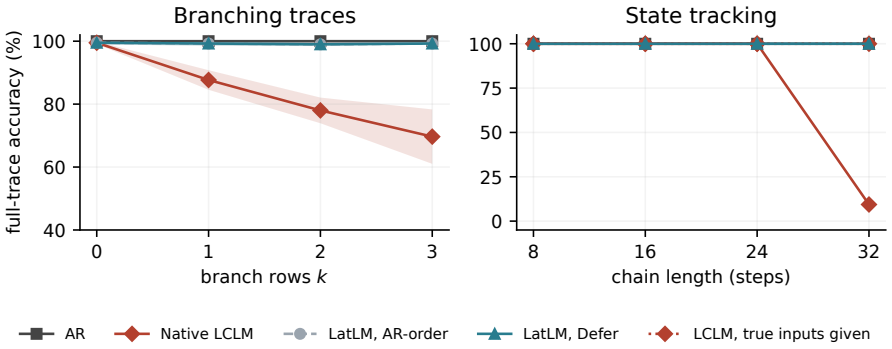


Figure 6: **Chains where every row waits: the gate is right where LCLM is not.** Left: branching traces, full-trace accuracy against the number of branch rows k (AR and LCLM: mean over three seeds, band one standard deviation; LatLM two seeds). Right: state tracking, accuracy against chain length; the dotted line gives LCLM the true inputs in the prompt. LatLM with DEFER matches its own sequential decode everywhere, at sequential cost.

What a dependent cell does when its input is missing. A probe (Appendix D.4) reveals that if a previous row's result is missing, a dependent cell often copies the most recent visible result; state step cells are an exception because they can recompute one lookup. If a wrong value is planted, it copies that instead. The lattice reads but does not verify. The gate prevents these confident-but-wrong copies by enforcing waits, though its entropy heuristic is imperfect: loosened thresholds let exactly these false copies slip through (Appendix D.7).

3.4 WALL-CLOCK DECODING COST

Table 4 reports wall-clock costs for the 482M pair. Single-request text generation with static caches and CUDA graphs takes 1.81 ms/cell under sampled width-six LCLM-ORDER and 2.54 under DEFER, versus AR's 3.59 (1.98× and 1.41×). Timing includes prompt seeding, scheduling and sampling, excluding graph setup. These are three repeats of one prompt with differing outputs, not a matched-quality latency comparison.

At batch 1024, query-state reuse reduces LatLM's round cost from 80.09 to 48.55 ms (1.65×), versus AR's 24.27 ms/step: the kernel-level break-even is about two commits per round. This synthetic benchmark fixes six query/commit slots and a 200-position visible prefix, excluding dynamic scheduling and sampling. Task accuracy and held-out BPB are evaluated separately from these timed operating points.

Table 4: **Wall-clock costs (482M, BF16, H100 NVL).** Left: batch-one sampled text; speedup compares ms/cell. Right: synthetic kernels using FlashAttention, ms/AR step or ms/LatLM round. Protocols and full results: Appendix E.

Decoder	ms/cell	Speedup
AR	3.59	1.00×
LatLM, LCLM-ORDER	1.81	1.98×
LatLM, DEFER	2.54	1.41×

Batch	AR	LatLM	+ reuse
1	3.08	3.57	3.80
64	4.48	6.56	5.77
256	8.34	20.11	13.19
512	13.58	38.94	24.19
1024	24.27	80.09	48.55

4 RELATED WORK

Flexible autoregressive order and structured generation. Any-order autoregressive models train under multiple orderings (Shih et al., 2022); A3 (Du et al., 2026) is the closest inference procedure, committing a confidence- or entropy-selected subset of positions each step and re-predicting the rest, over arbitrary groups with two-stream attention (Pynadath & Zhang, 2026); Esoteric LMs cache exactly under causal masks (Sahoo et al., 2025). The gate is therefore not new; what the lattice adds is the partial order, one residual stream per cell with the token entering only the key/value path, a cache exact along the realized trajectory, and the within-model controls of Section 3.3. Skeleton-of-Thought, APAR, PASTA, ASPD, Multiverse, Hogwild! and Multi-Stream LLMs generate response parts as parallel branches or streams (surveyed by Zhang et al., 2025; Ning et al., 2024; Liu et al., 2024; Jin et al., 2025; Chen et al., 2025; Yang et al., 2025; Rodionov et al., 2025; Su et al., 2026). The Line-Coupled Language Model (Li et al., 2026c) runs a waterfall schedule as a plain causal model, without the option to hold a line back.

Parallel token selection and verification. Masked and block diffusion predict multiple positions (Ghazvininejad et al., 2019; Sahoo et al., 2024; Nie et al., 2025; Arriola et al., 2025); Fast-dLLM's confidence threshold (Wu et al., 2026) plays the role of our gate, over a bidirectional fixed-size block with an approximate cache; set diffusion updates an exact cache after every flexible-position set rather than every block (Arriola & Kuleshov, 2026); lattice rows are variable-length, ordered by dependence rather than a diffusion schedule, and committed keys are never recomputed. Later commit rules use the confidence profile (Kasa et al., 2026), an auxiliary AR model (Israel et al., 2025), estimated dependencies (Ringel et al., 2026; Zoabi et al., 2026; Park et al., 2026), a self-containedness test (Wang et al., 2026a), or a learned policy (Jazbec et al., 2026); Cai & Li (2026) analyse when confidence-based decoding is efficient, and Asaria et al. (2026) measure how a diffusion LM commits, as Appendix D.7 does for the gate; ReFusion fills parallel slots autoregressively (Li et al., 2026a) and ParallelBench measures dependence-induced failures (Kang et al., 2026). Training under many legal histories, as our schedule mixture does, is shared with MBD-LM, D2F, NAP, Jacobi forcing and certainty forcing (Jin et al., 2026; Wang et al., 2026b; Li et al., 2026b; Hu et al., 2025; Chen et al., 2026). Unlike speculative sampling (Leviathan et al., 2023; Chen et al., 2023) or parallel token prediction (Draxler et al., 2026), lattice commitment carries no verification guarantee.

5 DISCUSSION AND CONCLUSION

LATLM allocates sequential computation according to dependence. At 482M it matches the autoregressive twin's held-out BPB, retains LCLM-level round reductions on independent lines, and recovers near-sequential accuracy where fixed schedules fail. The cost approaches sequential decoding on serial chains and half of AR's rounds on mixed chains. Wall-clock measurements show per-cell gains in text generation and reduced batched kernel cost with query-state reuse. Task-specific fine-tunes, limited seeds, and cross-model differences in representation and training preclude a universal threshold or LCLM failure boundary. Matched-quality serving speedups and a readiness signal for confident-wrong copies remain open.

Ethics Statement

This work studies the efficiency and reliability of language generation. Lower inference cost can broaden access while also reducing the cost of producing incorrect or harmful text. The method adds no factual verification; confident parallel commitments can propagate errors. Applications requiring reliable reasoning should evaluate correctness under the commitment policy they deploy.

Reproducibility Statement

Appendix A provides the structural and probabilistic details; Appendices B and C describe the architecture, training schedules, and evaluation conventions. Appendix D reports task definitions and additional results, and Appendix E the wall-clock protocols and results. The supplementary material includes a minimal executable reference implementation and two decoding videos. Full training code, fine-tuning generators and evaluation scripts will be released.

AI Use Statement

AI assistance was used for manuscript editing, literature discovery, code inspection, figure scripting, and analysis of stored experimental outcomes. This assistance does not replace author verification of the methods, references, or reported results.

References

Marianne Arriola and Volodymyr Kuleshov. Set diffusion: Interpolating token orderings between autoregression and diffusion for fast and flexible decoding. *arXiv preprint arXiv:2607.01775*, 2026.

Marianne Arriola, Aaron Gokaslan, Justin Chiu, Zhihan Yang, Zhixuan Qi, Jiaqi Han, Subham Sahoo, and Volodymyr Kuleshov. Block diffusion: Interpolating between autoregressive and diffusion language models. In *International Conference on Learning Representations*, 2025.

Ali Asaria, Tony Salomone, and Deep Gandhi. Neither parallel nor sequential: How DiffusionGemma actually commits tokens. *arXiv preprint arXiv:2606.14620*, 2026.

Changxiao Cai and Gen Li. Confidence-based decoding is provably efficient for diffusion language models. *arXiv preprint arXiv:2603.22248*, 2026.

Tianle Cai, Yuhong Li, Zhengyang Geng, Hongwu Peng, Jason D. Lee, Deming Chen, and Tri Dao. Medusa: Simple LLM inference acceleration framework with multiple decoding heads. In *Proceedings of the 41st International Conference on Machine Learning*, 2024.

Charlie Chen, Sebastian Borgeaud, Geoffrey Irving, Jean-Baptiste Lespiau, Laurent Sifre, and John Jumper. Accelerating large language model decoding with speculative sampling. *arXiv preprint arXiv:2302.01318*, 2023.

Keyu Chen, Zhifeng Shen, Daohai Yu, Haoqian Wu, Wei Wen, Jianfeng He, Ruizhi Qiao, and Xing Sun. ASPD: Unlocking adaptive serial-parallel decoding by exploring intrinsic parallelism in LLMs. *arXiv preprint arXiv:2508.08895*, 2025.

Zigeng Chen, Gongfan Fang, Xinyin Ma, Ruonan Yu, and Xinchao Wang. dParallel: Learnable parallel decoding for dLLMs. In *International Conference on Learning Representations*, 2026.

Shizhe Diao, Yu Yang, Yonggan Fu, Xin Dong, Dan Su, Markus Kliegl, Zijia Chen, Peter Belcak, Yoshi Suhara, Hongxu Yin, Mostofa Patwary, Celine Lin, Jan Kautz, and Pavlo Molchanov. Nemotron-CLIMB: Clustering-based iterative data mixture bootstrapping for language model pre-training. *Advances in Neural Information Processing Systems*, 38, 2025.

Felix Draxler, Justus Will, Farrin Sofian, Theofanis Karaletsos, Sameer Singh, and Stephan Mandt. Parallel token prediction for language models. In *International Conference on Learning Representations*, 2026.

Tianqi Du, Lizhe Fang, Weijie Yang, Chenheng Zhang, Zeming Wei, Yifei Wang, and Yisen Wang. Autoregressive models rival diffusion models at any-order generation. In *International Conference on Learning Representations*, 2026.

Marjan Ghazvininejad, Omer Levy, Yinhan Liu, and Luke Zettlemoyer. Mask-Predict: Parallel decoding of conditional masked language models. In *Proceedings of the 2019 conference on empirical methods in natural language processing and the 9th international joint conference on natural language processing (EMNLP-IJCNLP)*, pp. 6112–6121, 2019.

Fabian Gloeckle, Badr Youbi Idrissi, Baptiste Roziere, David Lopez-Paz, and Gabriel Synnaeve. Better & faster large language models via multi-token prediction. In *Proceedings of the 41st International Conference on Machine Learning*, 2024.

Jiatao Gu, James Bradbury, Caiming Xiong, Victor OK Li, and Richard Socher. Non-autoregressive neural machine translation. In *International Conference on Learning Representations*, 2018.

Lanxiang Hu, Siqi Kou, Yichao Fu, Samyam Rajbhandari, Tajana Rosing, Yuxiong He, Zhijie Deng, and Hao Zhang. Fast and accurate causal parallel decoding using Jacobi forcing. *arXiv preprint arXiv:2512.14681*, 2025.

Daniel Israel, Guy Van den Broeck, and Aditya Grover. Accelerating diffusion LLMs via adaptive parallel decoding. *Advances in Neural Information Processing Systems*, 38:52870–52888, 2025.

Metod Jazbec, Theo X Olausson, Louis Béthune, Pierre Ablin, Michael Kirchhof, João Monteiro, Victor Turrisi, Jason Ramapuram, and Marco Cuturi. Learning unmasking policies for diffusion language models. In *International Conference on Machine Learning*, 2026.

Tian Jin, Ellie Y Cheng, Zack Ankner, Nikunj Saunshi, Blake M Elias, Amir Yazdanbakhsh, Jonathan Ragan-Kelley, Suvinay Subramanian, and Michael Carbin. Learning to keep a promise: Scaling language model decoding parallelism with learned asynchronous decoding. In *International Conference on Machine Learning*, 2025.

Yijie Jin, Jiajun Xu, Yuxuan Liu, Chenkai Xu, Yi Tu, Jiajun Li, Dandan Tu, Xiaohui Yan, Kai Yu, Pengfei Liu, and Zhijie Deng. Multi-block diffusion language models. *arXiv preprint arXiv:2606.29215*, 2026.

Wonjun Kang, Kevin Galim, Seunghyuk Oh, Minjae Lee, Yuchen Zeng, Shuibai Zhang, Coleman Hooper, Yuezhou Hu, Hyung Il Koo, Nam Ik Cho, and Kangwook Lee. ParallelBench: Understanding the trade-offs of parallel decoding in diffusion LLMs. In *International Conference on Learning Representations*, 2026.

Siva Rajesh Kasa, Yasong Dai, Sumit Negi, and Hongdong Li. Fast-dLLM++: Fréchet profile decoding for faster diffusion LLM inference. *arXiv preprint arXiv:2606.02955*, 2026.

Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. In *International Conference on Machine Learning*, pp. 19274–19286. PMLR, 2023.

Jia-Nan Li, Jian Guan, Wei Wu, and Chongxuan Li. ReFusion: A diffusion large language model with parallel autoregressive decoding. In *International Conference on Learning Representations*, 2026a.

Pengxiang Li, Dilxat Muhtar, Tianlong Chen, Lu Yin, and Shiwei Liu. Why diffusion language models struggle with truly parallel (non-autoregressive) decoding? *arXiv preprint arXiv:2602.23225*, 2026b.

Shiyuan Li, Shaorong Zhang, Zhaorui Yang, Qian Zhang, Greg Ver Steeg, and Bingyao Li. Line-coupled language model. *arXiv preprint arXiv:2609.07129*, 2026c.

Mingdao Liu, Aohan Zeng, Bowen Wang, Peng Zhang, Jie Tang, and Yuxiao Dong. APAR: Llms can do auto-parallel auto-regressive decoding. *arXiv preprint arXiv:2401.06761*, 2024.

Shen Nie, Fengqi Zhu, Zebin You, Xiaolu Zhang, Jingyang Ou, Jun Hu, Jun Zhou, Yankai Lin, Ji-Rong Wen, and Chongxuan Li. Large language diffusion models. *Advances in Neural Information Processing Systems*, 38:50608–50646, 2025.

Xuefei Ning, Zinan Lin, Zixuan Zhou, Zifu Wang, Huazhong Yang, and Yu Wang. Skeleton-of-Thought: Prompting LLMs for efficient parallel generation. In *International Conference on Learning Representations*, 2024.

Wooje Park, Insu Lee, Minyoung Noh, Jaeyun Jang, Sungmin Lee, Kyuhong Shim, and Byonghyo Shim. Dependency-aware revocable decoding for efficient diffusion large language model inference. *arXiv preprint arXiv:2608.26574*, 2026.

Patrick Pynadath and Ruqi Zhang. Why any-order autoregressive models need two-stream attention: A structural-semantic tradeoff. *arXiv preprint arXiv:2602.16092*, 2026.

Liran Ringel, Ameen Ali, and Yaniv Romano. Dependency-guided parallel decoding in discrete diffusion language models. *arXiv preprint arXiv:2604.02560*, 2026.

Gleb Rodionov, Roman Garipov, Alina Shutova, George Yakushev, Erik Schultheis, Vage Egiazarian, Anton Sinitsin, Denis Kuznedelev, and Dan Alistarh. Hogwild! inference: Parallel LLM generation via concurrent attention. *Advances in Neural Information Processing Systems*, 38:46592–46633, 2025.

Subham S Sahoo, Marianne Arriola, Yair Schiff, Aaron Gokaslan, Edgar Marroquin, Justin T Chiu, Alexander Rush, and Volodymyr Kuleshov. Simple and effective masked diffusion language models. *Advances in Neural Information Processing Systems*, 37:130136–130184, 2024.

Subham Sekhar Sahoo, Zhihan Yang, Yash Akhauri, Johnna Liu, Deepansha Singh, Zhoujun Cheng, Zhengzhong Liu, Eric Xing, John Thickstun, and Arash Vahdat. Esoteric language models: A family of any-order diffusion LLMs. *arXiv preprint arXiv:2506.01928*, 2025.

Andy Shih, Dorsa Sadigh, and Stefano Ermon. Training and inference on any-order autoregressive models the right way. *Advances in Neural Information Processing Systems*, 35:2762–2775, 2022.

Mitchell Stern, William Chan, Jamie Kiros, and Jakob Uszkoreit. Insertion transformer: Flexible sequence generation via insertion operations. In *International Conference on Machine Learning*, pp. 5976–5985. PMLR, 2019.

Guinan Su, Yanwu Yang, Xueyan Li, and Jonas Geiping. Multi-Stream LLMs: Unblocking language models with parallel streams of thoughts, inputs and outputs. *arXiv preprint arXiv:2605.12460*, 2026.

Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. RoFormer: Enhanced transformer with rotary position embedding. *Neurocomputing*, 568:127063, 2024.

Danny Wang, Ruihong Qiu, and Zi Huang. When to commit? towards variable-size self-contained blocks for discrete diffusion language models. *arXiv preprint arXiv:2604.23994*, 2026a.

Xu Wang, Chenkai Xu, Yijie Jin, Jiachun Jin, Hao Zhang, Kai Yu, and Zhijie Deng. Diffusion LLMs can do faster-than-AR inference via discrete diffusion forcing. In *International Conference on Learning Representations*, 2026b.

Chengyue Wu, Hao Zhang, Shuchen Xue, Zhijian Liu, Shizhe Diao, Ligeng Zhu, Ping Luo, Song Han, and Enze Xie. Fast-dLLM: Training-free acceleration of diffusion LLM by enabling KV cache and parallel decoding. In *International Conference on Learning Representations*, 2026.

Xinyu Yang, Yuwei An, Hongyi Liu, Tianqi Chen, and Beidi Chen. Multiverse: Your language models secretly decide how to parallelize and merge generation. *Advances in Neural Information Processing Systems*, 38:100773–100804, 2025.

Lingzhe Zhang, Liancheng Fang, Chiming Duan, Minghua He, Leyi Pan, Pei Xiao, Shiyu Huang, Yunpeng Zhai, Xuming Hu, Philip S. Yu, and Aiwei Liu. A survey on parallel text generation: From parallel decoding to diffusion language models. *arXiv preprint arXiv:2508.08712*, 2025.

Shaorong Zhang, Longxuan Yu, Rob Brekelmans, Luhan Tang, Salman Asif, and Greg Ver Steeg. Generation order and parallel decoding in masked diffusion models: An information-theoretic perspective. *arXiv preprint arXiv:2602.00286*, 2026.

Tamim Zoabi, Ameen Ali, Liran Ringel, and Lior Wolf. Mean-field parallel decoding for discrete diffusion language models. *arXiv preprint arXiv:2606.15805*, 2026.

A Structural and Probabilistic Details

A.1 Legal sets and the frontier

Consider a rectangular grid $X = \{X_{r,c}\}$ with R rows and C columns, indexed from one, and define $(r, c) \preceq (r', c')$ when $r \leq r'$ and $c \leq c'$. A committed set S is legal if it is downward closed under $\preceq$: whenever a cell is committed, all its predecessors are committed. Equivalently, every committed cell has its left and upper neighbors committed, when they exist. The frontier is

$$F(S) = \{v \notin S : \text{all strict predecessors of } v \text{ belong to } S\}. \tag{4}$$

Proposition 1 (Legal parallel updates)**.** *The legal sets form a distributive lattice under union and intersection. For a legal S, $F(S)$ is an antichain with at most one cell per row and per column. Committing any $A \subseteq F(S)$ preserves legality.*

Proof. Let S_1, S_2 be order ideals. If $v \in S_1 \cup S_2$, then v belongs to at least one ideal, which contains all predecessors of v; hence $S_1 \cup S_2$ is an ideal. If $v \in S_1 \cap S_2$, its predecessors belong to both ideals, so their intersection is also an ideal. Union and intersection are the join and meet under inclusion and satisfy the distributive laws of sets.

Suppose distinct $u, v \in F(S)$ satisfy $u \preceq v$. The frontier definition requires every strict predecessor of v, including u, to belong to S, contradicting $u \in F(S)$. Thus $F(S)$ is an antichain. Cells in one row, or in one column, are comparable, giving at most one frontier cell per row and column. Finally, all predecessors of any $v \in A \subseteq F(S)$ already lie in S. Adding all of A therefore preserves downward closure, irrespective of their token values. □

Corollary 2 (Geometric parallel depth)**.** *For an initially empty $R \times C$ grid with no extra constraints and unlimited commit width, at least $R + C - 1$ rounds are required, and this number is attained by committing every frontier cell at each round.*

Proof. A monotone path from $(1, 1)$ to (R, C) contains $R + C - 1$ cells, each a predecessor of the next; no two can be committed in one round. This gives the lower bound. The schedule assigning round $r + c - 1$ to cell (r, c) is legal and attains it. □

This bound concerns the rectangular precedence graph only. A prompt, virtual padding, width limits, and additional dependency constraints change the decoding problem. It is not a lower bound on the number of neural evaluations required to solve a reasoning task by other means.

A.2 Histories and normalized generation

Let h_t contain the prompt and all earlier committed cells, their values, and their grouping into rounds; write S_t for its committed set. A policy selects a nonempty $A_t \subseteq F(S_t)$ with $|A_t| \leq K$, and a stochastic round uses the factorization

$$Q_{\theta, A_t}(x_{A_t} \mid h_t) = \prod_{v \in A_t} q_{\theta, v}(x_v \mid h_t), \qquad S_{t+1} = S_t \cup A_t. \tag{5}$$

Conditioning on h_t rather than S_t matters: cached representations retain their processing history, so different trajectories reaching the same committed set need not produce the same predictions. For a fixed trajectory, the product in Equation (5) over all rounds defines a normalized distribution over a finite grid because each round uses normalized conditionals on previous rounds. A deterministic adaptive policy with nonempty updates also defines a normalized sequential process when it depends only on the observed history and model predictions, not on unseen target values. Summing over next-round outcomes recursively verifies normalization. The per-token conditionals learned under different histories need not be marginals of a common, order-independent joint distribution. An adaptive schedule therefore need not preserve the distribution induced by AR-ORDER even when both use the same weights.

Algorithm 1 Entropy-gated commitment for a LATLM

Input: Given history h, legal set S, commit width K, threshold η.

1. Commit every legal structural cell without a forward pass: the anchor at column one of a newly legal row, and <void> padding after a row's <eol> (or after its last column), which closes the row.
2. Set $F \leftarrow F(S)$, ordered by row and then column. Stop if F is empty.
3. Compute $q_{\theta,v}(\cdot \mid h)$ and $H_{\theta,v}(h)$ for all $v \in F$.
4. Let A be the first K cells of F satisfying $H_{\theta,v}(h) \leq \eta$. If $A = \varnothing$, select the first row-major cell instead.
5. Predict each x_v, $v \in A$, using its distribution from step 3.
6. Append (A, x_A) to h, set $S \leftarrow S \cup A$, and return to step 1.

Output: Generated content.

A.3 Dependence and marginal prediction error

For a fixed history h and candidate set A, a standard decomposition separates the cost of independent commitment from marginal prediction error:

$$\mathrm{D_{KL}}\left(P_A \,\middle\|\, \prod_{v\in A} q_{\theta,v}\right) = \underbrace{\mathrm{TC}_P(X_A \mid h)}_{\text{dependence within the committed set}} + \underbrace{\sum_{v\in A} \mathrm{D_{KL}}(P_v \| q_{\theta,v})}_{\text{marginal prediction error}}, \tag{6}$$

where P_A is the target joint distribution, P_v its marginals, and $\mathrm{TC}_P = \mathrm{D_{KL}}(P_A \| \prod_v P_v)$ the total correlation; all distributions condition on h.

Proposition 3. *For categorical distributions with finite divergence and a fixed history h, Equation* (6) *holds.*

Proof. Suppress conditioning on h. Add and subtract the log of the product of target marginals:

$$\mathrm{D_{KL}}\left(P_A \,\middle\|\, \prod_{v\in A} q_{\theta,v}\right) = \mathbb{E}_{P_A}\left[\log \frac{P_A(X_A)}{\prod_{v\in A} P_v(X_v)}\right] + \sum_{v\in A} \mathbb{E}_{P_A}\left[\log \frac{P_v(X_v)}{q_{\theta,v}(X_v)}\right] \tag{7}$$

$$= \mathrm{TC}_P(X_A \mid h) + \sum_{v\in A} \mathrm{D_{KL}}(P_v \| q_{\theta,v}). \tag{8}$$

The last equality follows by marginalizing the expectation in each term. □

For finite categorical variables, $\mathrm{TC}_P(X_A \mid h) = \sum_{v\in A} H_P(X_v \mid h) - H_P(X_A \mid h)$, and non-negativity of joint entropy gives $\mathrm{TC}_P \leq \sum_v H_P(X_v \mid h)$, the bound behind the entropy heuristic of Section 2.4. Even exact marginals lose within-round dependence when sampled independently. Conversely, a unique deterministic answer given the prompt has zero target total correlation, yet a finite model may need intermediate results to compute it; the tasks of Section 3.3 test this reliance on context, not an information-theoretic need to reason serially.

B Architecture and Execution Details

Token routing. With $z_v^{(\ell)}$ the residual stream at layer ℓ and $e(x_v)$ the cell's token embedding, the projections are those of Eq. 1 at every layer, up to normalization and positional rotation. The query's initial residual contains no x_v. By induction over layers, its attention can receive token identities only from strictly earlier rounds, which cannot have attended to v in their own updates. Tokenwise feed-forward layers preserve this property. For frontier inference, no token embedding is needed for an uncommitted query, since its key and value are not visible to another current query. The implementation uses special handling for given seed positions, whose values are already part of the prompt.

Coordinates and structure. The checkpoint combines two-axis RoPE with normalized absolute coordinate features and boundary flags; it is not purely translation invariant in the row coordinate. In pretraining and held-out text scoring, column normalization uses a fixed configuration value of 66 rather than the current document's maximum realized length, preventing that feature from revealing future row lengths. Task fine-tuning and prompt prefill normalize columns by the packed grid width; the reported task decodes use 66 for incremental scoring and cache updates. Rows longer than the content limit are wrapped. Once <eol> is produced, subsequent row padding is structural and omitted from transformer inputs. Task evaluations supply row anchors and the output row count; the model predicts <eol>, and generated <eol> cells are included in reported rounds.

Two gating implementations. Algorithm 1 specifies the reference reasoning policy: if every candidate exceeds the threshold, it commits the first row-major cell. The fused free-generation implementation used in Table 6 instead falls back to up to K lowest-entropy candidates. It also predicts row termination and uses the fixed 2.5-nat threshold in that benchmark. Its timing must therefore not be assigned to the reasoning policy without a matching evaluation.

C Training and Evaluation Configuration

Corpus and token budget. The corpus is ClimbMix (Diao et al., 2025), with an 8k BPE tokenizer. The corpus has approximately 14.2M training documents and 13.13B content tokens, with a held-out document split. The lattice samples six-row windows; the causal reference samples chunks of up to 512 tokens. Both sample documents uniformly, followed by offsets within documents. The training budget counts sampled content-token exposures, approximately one corpus token count; random window sampling does not mean that every corpus token is visited exactly once.

Architecture and optimizer. Both 482M models use 24 transformer layers, hidden dimension 1280, 16 attention heads, feed-forward dimension 5120, AdamW, weight decay 0.1, gradient clipping at 1.0, batch size 96, and bf16 training. The causal reference trains for 350,500 steps, uses a peak learning rate of 10^{-3} after 200 warm-up steps, and decays linearly over its last 15% of steps. The lattice is a continuation: initial pretraining uses 6×10^{-4}; adaptation to the padding-free layout and supervised row termination uses 5×10^{-5}; the final continuation keeps that rate before a final linear decay. The combined lattice exposure is approximately 13.1B content tokens. Row starts receive weight four in its training loss; evaluation is unweighted. Neither configuration was selected through a learning-rate sweep at the final budget.

BPB convention. For document i, let L_i be summed unweighted content-token NLL and B_i its byte denominator. We report $(\sum_i L_i)/(\log 2 \sum_i B_i)$. The evaluator computes B_i as the sum of UTF-8 lengths obtained by decoding each scored token individually; both models share this denominator. This tokenwise convention may differ from byte counting after decoding the entire concatenated text. Bootstrap resampling operates on paired document-level (L_i, B_i) records. Scores exclude structural-token losses and therefore are content modeling scores, rather than full-document code lengths including structural generation.

Training cost. Matched content-token exposure is not matched compute. Timed on one H100 at batch 96 with each model's pretraining configuration, the lattice trainer processes about $1.3\times$ fewer content tokens per GPU-second than the autoregressive twin, roughly 130 against 100 single-GPU hours for the two runs. The measured ratio is not explained by parameter count alone: a committed cell's token-to-key/value routing (Eq. 1) runs the shared query/key/value projection twice, on the residual stream with and without the token embedding, which adds roughly a quarter to each block's linear-layer cost relative to one autoregressive step; the packed two-dimensional attention and the six-row windows change the attention cost separately.

Fine-tuning cohorts. Every task fine-tune starts from the final lattice checkpoint (or the matched AR and LCLM checkpoints) and trains all three models for the same number of updates on the same generator: 3,000 for records and list, 6,000 for the branching traces, 12,000 for state tracking and the modular and mixed chains. Test problems use a separate generator split, with offset 10,000 for the simple tasks, modular and mixed chains, and fresh-set confirmations; the state and branching

evaluations use offset zero. There are 400 problems per simple task, 200 per condition otherwise, with 32 test problems per depth in the state scan. The state and branching scans use a 64-column cap and nine prompt rows; the modular and mixed chains 32 columns and four prompt rows. LatLM upweights row-start losses by four; AR and LCLM use unweighted losses. In a mixed chain each row after the first is independent with probability 0.4, dependent with 0.3 or restating with 0.3, and the answer adds the last result to an earlier independent one.

In the native LCLM task decoder, the answer-row anchor enters the cache in the round that starts the last reasoning row, one round earlier than in its training layout. Cross-model task comparisons therefore also include this difference in training and decoding context.

DEFER calibration. Thresholds are entropy quantiles observed under the LCLM-ORDER policy on validation problems (64 for records and list, 32 elsewhere), then frozen; within each scan the checkpoint, width, layout and threshold are fixed across test conditions. Every model generates its own row endings with the output row count specified. Equal aggregate accuracies do not imply identical per-example outputs.

D ADDITIONAL TASK RESULTS

D.1 DECODING EXAMPLES

Figures 7–9 show single decodes with every cell shaded by the round it was committed in. On the mixed chain (Figure 7) the restating row needs the value the row above has not yet written. Native LCLM's waterfall reaches that token first and restates a wrong digit; the rest of its trace is consistent with the wrong value. LatLM's fixed schedule does the same. The gate holds that one cell until the value above exists and commits the rest of the row on schedule, so it finishes in half the sequential rounds with the right answer. On the records task (Figure 8) nothing needs to wait: the gate advances every line together and produces the same text as sequential decoding in a third of the rounds. On held-out text (Figure 9) the gate commits about two cells per round; the cells it holds longest are line starts and the openings of clauses, which is where Appendix D.10 finds the waterfall's loss.

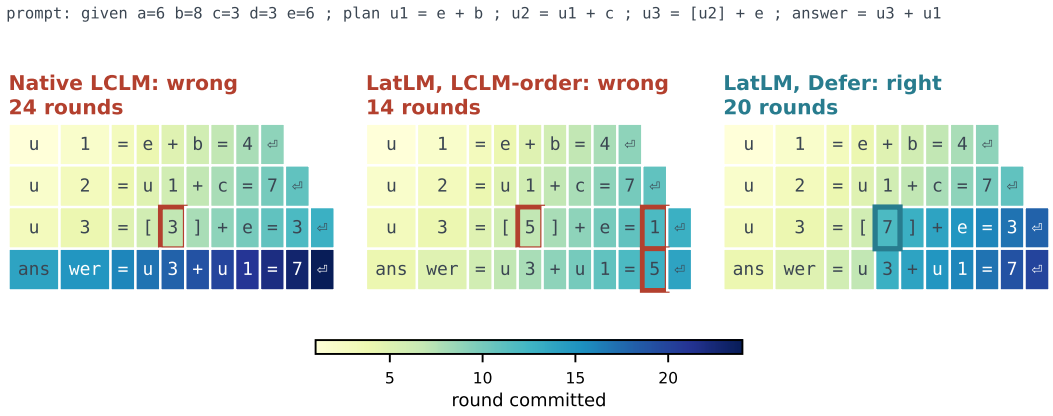


Figure 7: **One mixed chain under three decoders.** Cells shaded by commit round (native LCLM: the emission order its layout implies). Red frames mark tokens that differ from the reference; the teal frame marks the restated digit the gate held. LatLM with AR-ORDER takes 41 rounds.

D.2 STATE TRACKING ACROSS DEPTHS

A prompt gives two permutations over eight states, a start state and a sequence of operations; each output row records its input and the result of one lookup, so every row needs the result of the last. Through 24 steps every model is correct and native LCLM needs a fifth of the rounds (Figure 6, right). At 32 steps LCLM fails on nearly every trace; given the true inputs in the prompt it completes them all, so what it loses is not the lookup but the running state, which its waterfall must recompute from the prompt before the previous row has finished. LatLM with DEFER is correct throughout with both

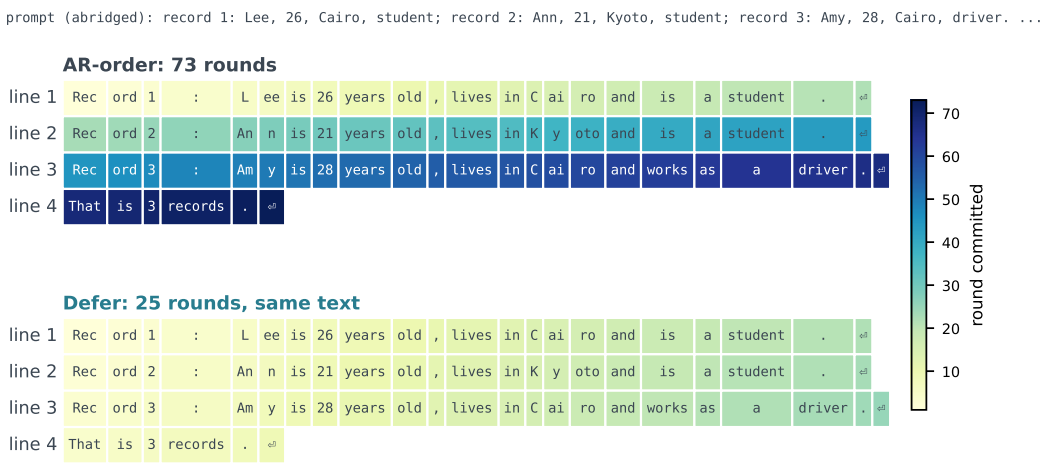


Figure 8: **One records problem, AR-ORDER and DEFER.** Independent lines advance together under the gate; the text is identical.

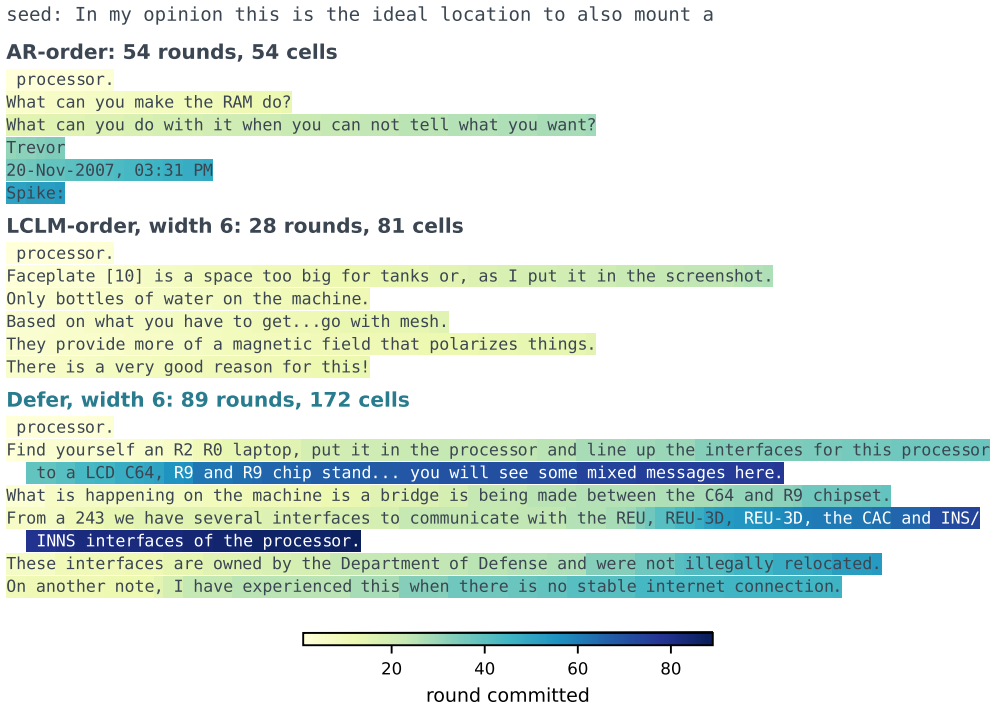


Figure 9: **One held-out seed continued by three policies of the same checkpoint** (sampled, temperature one, top-$p = 0.95$; each policy draws its own continuation). Shading gives the commit round; the gate uses the generation study's threshold.

training seeds, the separation replicates on four times as many fresh problems, and its cost is the full sequential round count. LCLM's failure point moves with training budget, so it is a limit at this budget rather than a bound on depth, and row count and depth are confounded.

D.3 SAMPLING DIAGNOSTIC FOR THE FORK SCAN

Under argmax a near-tie at a branch is broken by a deterministic function of the logits, and the successor row, reading the same prompt with the same weights, breaks it the same way; consistency is

then free without reading anything. Sampling (temperature one, top-$p = 0.95$, thresholds recalibrated under sampling) makes the branch a real coin flip. LCLM's accuracy at three branches roughly halves relative to greedy decoding and nearly all of its wrong rows are branch rows; the autoregressive model is unaffected, and LatLM with DEFER stays within a few points of its own sequential decode. This is the sense in which the failure is informational rather than computational: the value is not in the prompt.

D.4 REVEAL PROBE: WHAT A DEPENDENT CELL DOES WHEN ITS INPUT IS MISSING

On the fork and state LatLM checkpoints, we take a dependent row's source cell (or the answer row's number) and score the frontier once under teacher-forced histories that differ only in the previous row's result: *present*; *wrong* (another admissible value planted in that cell); *absent* (the previous row committed through its operator, result cell empty); and *holding* (rows above complete, the previous row committed only through the column that makes the probed cell legal – the state the gate is in while it holds a row). 32 fresh test problems, four dependent rows each plus the answer row; entropy is computed as in the evaluation decode. Table 5 reports the probability of the correct value, of the planted value, and of the most recent result visible on the grid (the row above the predecessor's). A diagnostic that uses the reference to build histories, not a decoding method.

Table 5: Reveal probe. Mean probability of the correct source value, the planted wrong value, and the most recent visible result; median entropy. The first and third can coincide when the most recent visible result is the correct value, so the columns are not disjoint. Answer rows are the final "Answer" / "ends with" cells.

Cells	History	p(correct)	p(planted)	p(most recent visible)	Entropy
fork, step	present	1.000	–	–	0.001
fork, step	wrong	0.001	0.994	–	0.001
fork, step	absent	0.122	–	0.730	0.483
fork, answer	holding	0.004	–	0.916	0.107
state, step	present	1.000	–	–	0.000
state, step	wrong	0.104	0.896	–	0.024
state, step	absent	0.985	–	0.128	0.015
state, answer	holding	0.209	–	0.900	0.224

Three things follow. The lattice reads the previous row's result rather than deriving it: with the result present it is exact, and a planted wrong value is copied. With the result missing it writes the most recent result it can see, the row above the predecessor's, on most fork step cells and on essentially every answer row; state step cells are the exception because a one-step table lookup is recomputable from the visible operator. And the gate's threshold has to be as tight as it is because these copies are often confident: in a looser greedy decode of the state task every loss was an answer row equal to the second-to-last result, and the probe at the holding history had that copied value as its argmax in every one of them.

D.5 WHEN ONLY ONE CELL MUST WAIT: MODULAR CHAINS

Modular chains make the dependence local. A plan of three to eight steps adds a named constant to the previous value modulo ten; every token of a row but its result digit is a copy from the plan, and a final row copies the last value. The prompt states the plan, so native LCLM, recomputing the running value from the prompt, is lossless here; the task isolates what the gate does when exactly one cell per row must wait. LatLM's fixed schedule fails on the answer row and almost nowhere else, because that row is written before the last result exists. Two controls isolate this effect. Holding the answer row until every other row has ended, LCLM's own answer tail, recovers the accuracy at the cost of the whole row; the gate recovers it within half a round of the fixed schedule, because it holds only the digit that is waiting (Figure 10, third panel).

D.6 DEFER VERSUS RANDOM DEFERRAL AT MATCHED COST

Is the gate's recovery a matter of deferring more cells or of deferring the right ones? Figure 10 places every policy of one checkpoint per task on the accuracy–rounds plane, with two controls.

Random deferral vetoes every frontier candidate independently with a fixed probability, under the same progress fallback; it approaches sequential accuracy only at nearly sequential cost. Row-major replay commits, in each round, as many cells as the gate committed in that round of the same problem, but the first cells in row-major order; it matches the gate's rounds approximately and collapses to the fixed schedule's accuracy. At the parallel operating points, matching the commit count alone does not recover the gate's accuracy; the choice of cells matters.

Two further controls (Figure 10) place the gate between a fixed rule and an oracle that sees the dependency edges but not the values. A *line stagger*, delaying each row a fixed number of columns behind the one above, matches the gate only on the modular chains, where a single cell waits; on the other three tasks it is safe only once it is nearly sequential. The oracle, which holds every dependent cell until the row it reads has ended, is lossless everywhere and marks how much of the near-sequential cost is the task's rather than the gate's: on the mixed and modular chains the gate is already close to it, but on the branching traces it captures only a small share of the oracle's saving over sequential decoding, and on the state chains none at all, where entropy's only safe quantile is fully sequential.

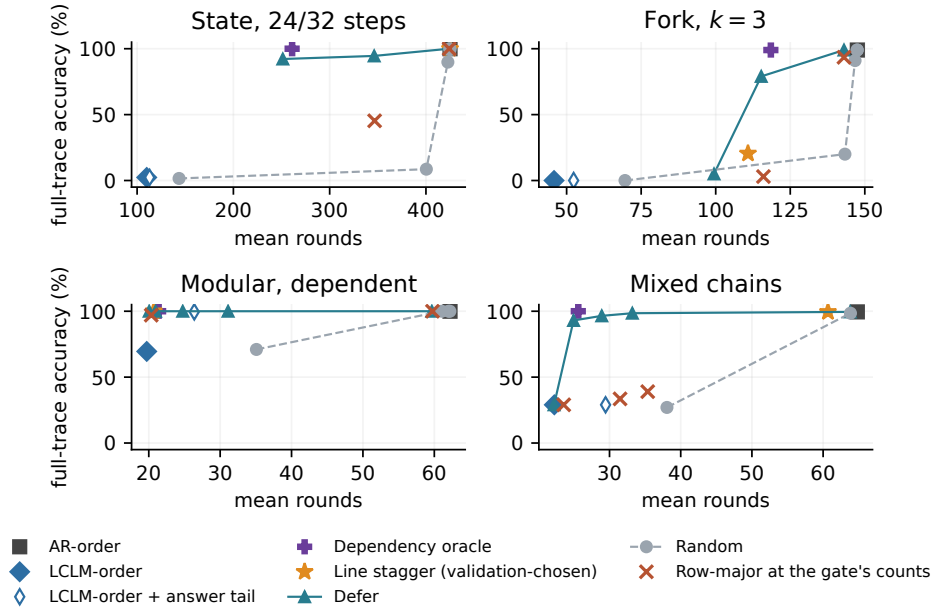


Figure 10: **Accuracy against rounds for every policy of one checkpoint.** One panel per dependent task, same fresh problems within a panel; left and up is better. DEFER points are the calibrated quantiles, random points the veto rates; single runs, no error bars. On the state and fork chains only the near-sequential end is accurate; on the modular chains the gate is lossless down to the fixed schedule's cost; on the mixed chains it trades a few points for less than half the rounds where the fixed schedule collapses. Crosses: the gate's per-round commit counts replayed with row-major cells; plus signs: the dependency oracle; stars: the line stagger chosen on validation.

D.7 WHAT THE GATE DOES AT EACH OPERATING POINT

Analysis of commitment schedules shows two regimes. At the median-entropy threshold of the state and branching tasks it is sequential decoding in all but name: about one cell per round, the fallback firing in nearly half the rounds, and no dependent cell ever committed before its input. Its accuracy there comes from staying in dependency order; random deferral can also approach this accuracy at nearly sequential cost. At looser thresholds the gate selects. On the branching task nearly a thousand input cells reached the frontier before their result; the gate held all but a few dozen until the result arrived, and every trace in which it let one through failed. On the mixed chains at the featured quantile it held nineteen in twenty of the restated inputs that reached the frontier early, which is where its

accuracy at half the rounds comes from. Where a cell it lets through is recomputable, as a state one step behind is, the trace usually survives; where it is not, it does not.

D.8 Mixed chains across dependence densities

The mixed chains of Section 3.3 draw each row after the first as independent with probability $1 - \rho$ and dependent or restating otherwise ($\rho = 0.6$ in the main text). Figure 11 repeats the protocol across densities, one fine-tune per model and density: LCLM's accuracy falls steadily, while the gate stays within a few points of sequential accuracy, its rounds rising from about a third of the autoregressive model's to about half. Calibration on 32 problems is not infallible: at the two highest densities, one of two LatLM seeds has no quantile free of calibration error and contributes no DEFER point rather than a misleadingly safe one, and at $\rho = 0$, where no row but the answer reads a written value, that fine-tune never learns the merged answer even though a $\rho = 0.6$ checkpoint solves the same problems – our reading is that the lookup gets too little training signal when nothing else in the output needs a written value, a limitation of the training scheme rather than the decoders. The more direct test of adaptivity is the bottom row: one checkpoint and one threshold, both fixed at $\rho = 0.6$, applied without retraining to every other density. Its rounds still track the density it meets while its accuracy stays close to sequential, whereas the same native LCLM checkpoint collapses from lossless to under a third – the gate adapts to the dependence it meets, the fixed schedule cannot.

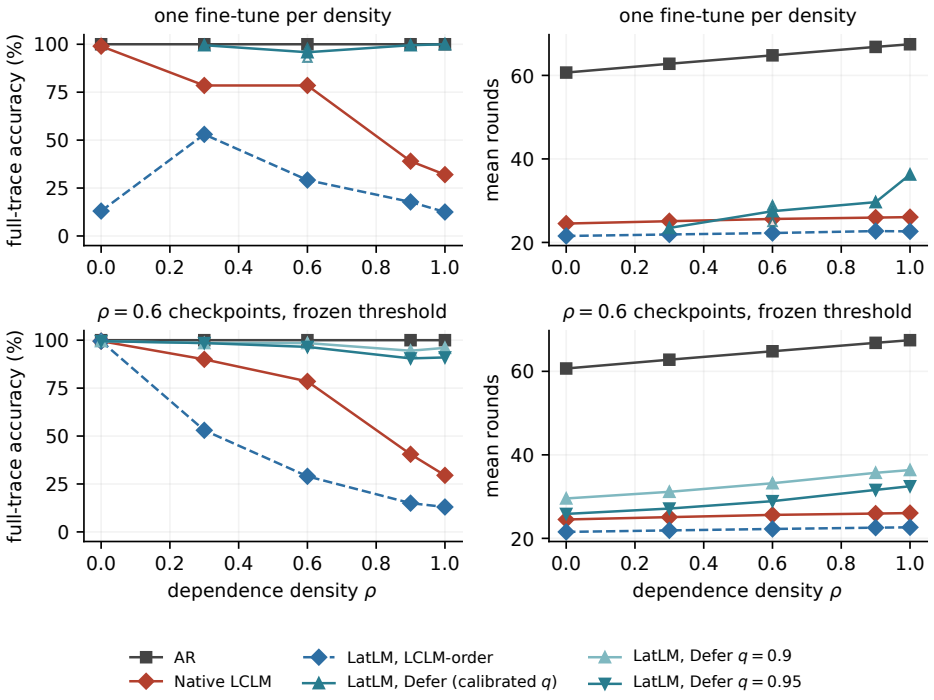


Figure 11: **Mixed chains by dependence density.** Full-trace accuracy (left) and mean rounds (right) against the fraction ρ of rows that need the previous result. Top: one fine-tune per model and density, the gate at its calibrated quantile. One fine-tune per model and density; where several LatLM seeds exist the DEFER line is their mean and hollow markers the seeds. Bottom: the $\rho = 0.6$ checkpoints of Table 3, with LatLM's threshold frozen at its $\rho = 0.6$ calibration, applied to the other densities.

D.9 Training budget and recomputation depth on the mixed chains

Native LCLM's waterfall reaches a restating row's bracketed source digit before the row above has written its result, so the value it restates must be recomputed from the nearest values the waterfall can see. Fine-tuned for twice and four times the updates, LCLM rises from 78.5% to 92% and stays there. Figure 12 locates the errors by the depth of the recomputation, the number of dependent rows since

the last independent one: more training removes the shallow errors and leaves the deep ones, while the gate, which waits for the value instead of recomputing it, has almost none at any depth. Doubling LatLM's own budget leaves its sequential decode and its gate at the featured quantiles within a point of the 12,000-update checkpoint; only the loosest operating points move.

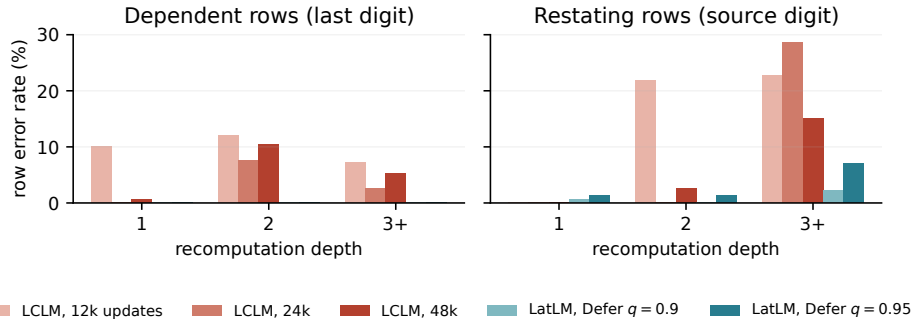


Figure 12: **Row error rate by recomputation depth on the mixed chains.** Depth d: the row is the d-th dependent or restating row since the last independent row; rows are counted while the trace above them is still correct. Dependent rows need the previous result in their last digit, restating rows in their bracketed source digit. AR and LatLM with AR-ORDER make no errors and are omitted.

D.10 HELD-OUT LOSS BY POSITION WITHIN THE LINE

Scoring the held-out windows of Table 1 by the position of each token within its line locates the waterfall's cost where LCLM's own analysis does (Li et al., 2026c): at the first token of a line, which is generated before the line above has ended, and to a lesser degree at the next three; LatLM's own schedules match autoregressive decoding from the fifth token on; native LCLM, scored under its own layout, does not – it remains about 0.02 bits per byte above AR there, a gap that is small per token but, summed over the many tokens at that position, is a majority of its total excess loss over AR. A teacher-forced entropy gate recovers this loss from the inside out: a loose threshold restores the later positions and half of the second to fourth while leaving the line start untouched, and only a threshold that makes the decode sequential reaches the line start. A decoding-time rule does better there: letting a line start only once the line above is a few cells ahead of the waterfall recovers most of the line-start loss at two cells per round, and at the same total loss uses fewer rounds than the gate alone. Entropy is one policy the lattice can run, and on free text not the best one.

D.11 FREE TEXT GENERATION

The generation study uses 48 held-out prompts, a 12-token seed, a 6×64 grid and top-$p = 0.95$ sampling, scored by an earlier autoregressive checkpoint as a likelihood judge. Sampled width-six continuations raise the judge's loss by about 6% relative to sequential lattice samples. Greedy width-six generation scores better on the judge and much worse on distinct trigrams: a favourable likelihood can accompany repetitive output, so the greedy throughput figure is not quality-preserving acceleration. The DEFER row uses its own threshold, distinct from the latency benchmark and the reasoning calibrations.

E WALL-CLOCK PROTOCOLS AND QUERY-STATE REUSE

Single-request text generation. The matched 482M models use BF16 weights, static caches and CUDA graphs on an H100 NVL. We use the first 12 tokenizer tokens of "The city council met on Tuesday to discuss the new budget." as the prompt and a 6×64 lattice. Sampled rows use temperature 1, top-$p = 0.95$ and random seed 0; DEFER uses an entropy threshold of 2.5 nats, committing up to six lowest-entropy candidates if none passes the threshold. After a setup call allocates buffers and captures the graph, we take the median of three complete generation calls, synchronizing CUDA before and after each timed call. Timing includes per-request prompt seeding, scheduler logic and

sampling, but excludes setup. Table 6 divides this measured duration by the number of generated cells, including row endings and excluding the prompt. Output lengths differ: AR and the fixed schedules end all six rows; DEFER ends five, with the remaining row reaching the grid boundary. These timings do not measure native LCLM or share the DEFER threshold of the 48-prompt quality study; the task decoder also uses a different progress fallback.

Table 6: **Single-request wall-clock latency.** H100 NVL, 482M, BF16, 6×64 grid, 12-token prompt, medians of three calls. Speedup is the ratio of AR to LatLM ms/cell; outputs differ across policies.

Decoder	Rounds	Cells	ms/round	ms/cell	Speedup
AR, static cache + graph	191	191	3.59	3.59	1.00×
LatLM, AR-ORDER greedy	66	66	5.32	5.32	0.67×
LatLM, LCLM-ORDER width 6, greedy	20	84	8.62	2.05	1.75×
LatLM, LCLM-ORDER width 6, sampled	59	181	5.56	1.81	1.98×
LatLM, DEFER width 6, sampled	102	198	4.94	2.54	1.41×

Batched kernel timing. We time the same model pair with a fixed 200-position visible prefix, six query and six commit slots per LatLM round, and batches from 1 to 1024 (Table 7). Both models use BF16, FlashAttention over the contiguous visible prefix, and CUDA graphs. Following three warm-up calls and graph capture, we report the median of 20 graph replays, with CUDA synchronized before and after each wall-clock measurement. Inputs and caches are synthetic; the prefix mask does not exercise arbitrary row-window masks. These measurements exclude prompt processing, sampling and dynamic batch scheduling.

Table 7: **Batched wall-clock kernel costs.** Milliseconds per AR step or LatLM round on H100 NVL. Reuse speedup compares the fused LatLM kernel with query-state reuse, not LatLM generation with AR generation. Break-even is reuse ms/round divided by AR ms/step: the required mean commits per round per sequence to match AR's kernel throughput.

Batch	AR	LatLM fused	LatLM reuse	Reuse speedup	Break-even
1	3.08	3.57	3.80	0.94×	1.23
64	4.48	6.56	5.77	1.14×	1.29
256	8.34	20.11	13.19	1.52×	1.58
512	13.58	38.94	24.19	1.61×	1.78
1024	24.27	80.09	48.55	1.65×	2.00

Query-state reuse. State reuse retains the normalized input to each attention layer for frontier queries; after selection, the relevant states are combined with the committed token embedding and projected into keys and values, so a committed cell's attention and feed-forward layers are not recomputed. Deferred candidates use their latest scoring states. Reuse adds overhead at batch one but reduces round cost at larger batches. At batch 1024, the kernel-throughput ratio to AR would be $c\,t_{\mathrm{AR}}/t_{\mathrm{reuse}} \simeq c/2$ for a mean of c committed cells per round; c is not measured by this synthetic benchmark. On shared trajectories reuse changes no argmax in FP32 and three of 515 in BF16, and its held-out loss shift is within the bootstrap interval of zero.