AUTOINDEXER: FLEXIBLE-ORDER DECODING BY TRAINING CAUSAL MODELS ON CHAINS OF EDITS

Anonymous authors
Paper under double-blind review

ABSTRACT

Autoregressive decoding faces challenges for data that are not causally ordered, e.g., containing cross-references or future dependencies. Furthermore, once tokens are generated, they are fixed and cannot be modified or improved, unlike how humans can iteratively refine their thoughts. While diffusion LLMs partially address this issue, they do not naturally lend themselves to adding or removing arbitrary numbers of tokens to/from a pre-generated context. We design AutoIndexer as a generative model with a novel training and decoding mechanism that simulates a *Chain of Edits*, allowing the model to flexibly modify its prior outputs in hindsight via token insertions, substitutions, and deletions. The model learns to predict marker tokens for opening and closing edits, simultaneously moving a cursor to positions in the existing token sequence, analogous to a text editor. Training sequences are obtained by perturbing ground-truth labels with inverse chained edit operations, and the model learns to edit by undoing those inverses. The perturbation training allows AutoIndexer to be robust to erroneous context and adaptable to editing tasks such as code correction. We demonstrate that our method exhibits post-generation context-editing capabilities that most autoregressive and diffusion models lack without compromising performance on standard LLM benchmarks. The code is available at `https://anonymous.4open.science/w/autoindexer-CE1F/`.

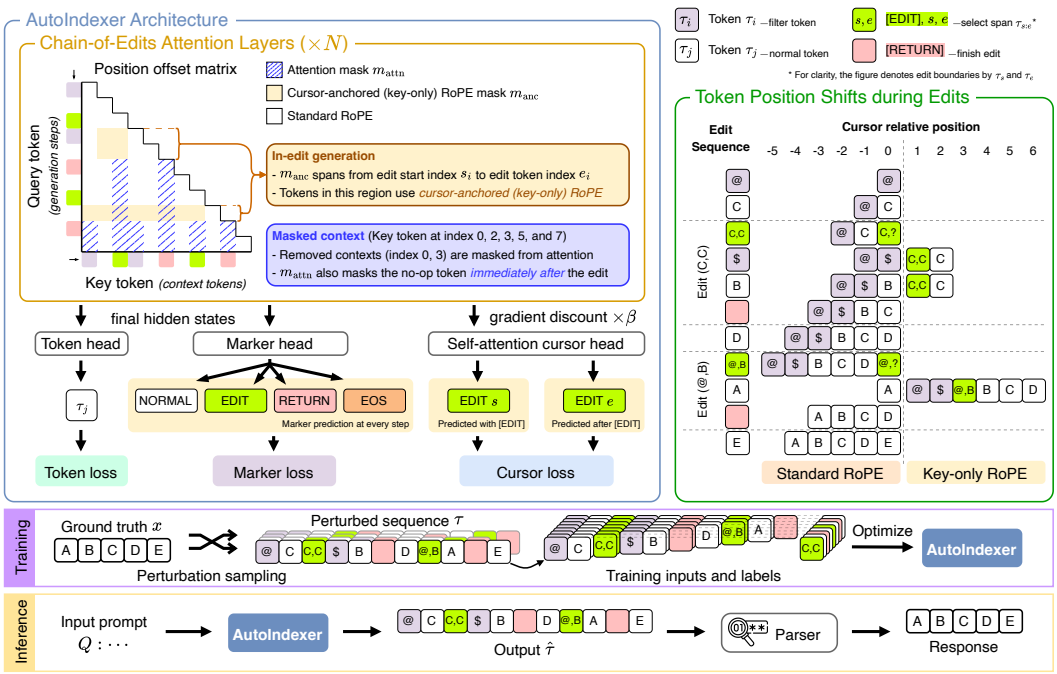

Figure 1: Overview of the AutoIndexer method. AutoIndexer can edit its prior outputs via insertion, substitution, and deletion. Analogous to using a text editor, the model selects a subset of the tokens in the context for *edits* by predicting 'start' and 'end' cursor positions. The positional encodings of the tokens are modified after each edit and efficiently computed by our specialized attention kernel.

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.

1 INTRODUCTION

Generative sequential models such as Large Language Models (LLMs) (Radford et al., 2018; Achiam et al., 2023; Bai et al., 2023; Grattafiori et al., 2024; Guo et al., 2025) have typically been trained autoregressively, due to the ease of formulating a generative task as a next-token prediction problem, and parallelizing the learning of its conditional probability distribution.

Autoregressive modeling is most effective when the order in which the data is presented is aligned with the direction of causality. However, not all tasks align well with autoregressive generation. For example, LLMs struggle to generate a large document with self-consistent reference sections, chapters, and word count for blocks of text. These references and numbers should be determined after the majority of the text has been generated, but because of the autoregressive constraint, the model may have to choose a section number arbitrarily and then later have to force the word count or numbers of sections to fit the requirement. A related challenge is the difficulty of modifying or improving generated outputs. A useful ability for a generative model would be to insert, edit, or delete text with ease, as one would do with a text editor. This would also allow for a more hierarchical generation, where the model can first output an outline of the solution as some form of high-level blueprint, summary, or bullet points, and then edit and refine the details to obtain the final solution.

This motivates a decoding mechanism that is more flexible and allows amendments in hindsight. A recent surge in work on discrete diffusion (Austin et al., 2021; Lou et al., 2024; Nie et al., 2025) attempts to address this shortcoming as an iterative token demasking and correction task. However, there are several limitations to this approach. First, most diffusion architectures generate a fixed-length output, which is not effective for applications such as chat systems, where model generations are of arbitrary length. Second, discrete diffusion regenerates the entire text using a bi-directional encoder, which cannot reuse previous computations with KV caching, making inference less efficient. Third, even though the token-correction mechanism could allow modifications to previously generated output, having a fixed position for each token prevents insertion or deletion of arbitrary numbers of tokens. Block diffusion (Arriola et al., 2025) combines autoregressive generation with block-wise discrete diffusion, alleviating the first two issues; however, such approaches are still constrained by the absolute positions at which tokens were generated, with limited edit capability.

We postulate that the limiting factor in existing work is that they do not distinguish between the *generated positions* and the *display positions* of tokens when they apply positional encoding. By decoupling these two positions, the model can generate tokens in a flexible order that is then parsed and reordered into the display order. In this work, we propose a novel training paradigm, which applies a set of inverse edit operations to the target sequence, and train the model to undo the inverse edit operations, effectively making the model learn to edit and refine the context that it generated. This is analogous to diffusion training, in which the model is trained to undo a noising operation.

The uniqueness of our method is that the model reassigns a new set of positional encodings to tokens in the context according to the *display order* during and after each edit, as opposed to the *generated order*. The model can perform operations such as insertion, editing, and deletion to manipulate existing outputs in a similar way to how we write documents using a text editor. Once tokens are substituted or deleted, they are also masked out so that the model only sees the tokens currently in context in the *display order* during further generation. Therefore, the underlying model is still autoregressive in its generation order, retaining its benefits such as KV-caching and early stopping of generation via an End-of-Sequence token `[EOS]`, but can be non-autoregressive in the display order.

```
RAW     <h4 className="col-12 col-md-12 my-3 mx-auto contact-t «<...»>
        question or are you a hairstylist looking to join
        us?</h4>...[EDIT 61,61]itles">Do you have a [RETURN]...
PARSED  <h4 className="col-12 col-md-12 my-3 mx-auto contact-titles">Do
        you have a question or are you a hairstylist looking to join
        us?</h4>
```

Figure 2: Example of a successful edit made by AutoIndexer. The model opens an edit at display cursor `[61, 61)`, inserts the missing heading text, and returns to continue generation. «<...»> is where the omission of the prompt took place (omission location is not disclosed to the model). More examples are available in Appendix A.7.1.

Table 1: Comparison between decoding strategies of transformers.

	Autoregressive	Discrete Diffusion	Chain of Edits (Ours)
Generation order of tokens	Autoregressive	Non-autoregressive	Autoregressive
Display order of tokens	Fixed-order	Fixed-order	Decoupled (flexible)
Prediction target	Next token	Token demasking	Undo perturbation
KV-caching	Exact	Approximate	Exact
Amendable context	No	Limited	Yes

We name the method AutoIndexer because it learns to identify the positions in the context that require editing, and applies self-correction. AutoIndexer has a marker head to determine the timing of the edit, and a cursor head that predicts the edit cursor positions, in addition to the vocabulary token head. This is in contrast to Fill-in-the-Middle (FIM) (Bavarian et al., 2022), where the section to be edited is given, both during training and at inference.

Our contributions are the following: (a) **Active Context Refinement:** a model that can modify its output in any order without length constraints, by making it decide when and where an edit is required and move a cursor to any position in the sequence, unlike Fill-in-the-Middle (Bavarian et al., 2022) where the edit span is given externally; (b) **Perturbation training:** a self-supervised learning paradigm to teach the model to edit by perturbing each ground-truth sequence with randomly sampled inverse edit operations, and training the model to autoregressively undo this perturbation, analogous to a diffusion objective but over discrete edit inverses rather than noise; (c) **Fused Kernel with Chain-of-Edits RoPE:** an attention kernel that computes the relative positional encodings using RoPE (Su et al., 2024) but for multiple edits in one forward pass for training efficiency, informing the model of the updated token positions after each edit operation; and (d) **Sequence Parser:** a parser that converts the model's generation with multiple edit operations back to *display order*.

2 AUTOINDEXER

AutoIndexer's decoding mechanism decouples generation order and display order. While reusing the transformer backbone, AutoIndexer augments conventional next-token prediction with a redesigned model training method, a cursor prediction head, edit-aware positional encoding, and a parser that maps the raw generation to a display sequence. The overview of the method is shown in Figure 1.

2.1 ACTIVE CONTEXT REFINEMENT VIA MARKER TOKENS AND EDIT INDICES

AutoIndexer is equipped with an additional marker head to initiate an edit, and a cursor head to define the position of the edit. The marker head can predict `[NORMAL]` for normal vocabulary tokens, `[EOS]` for End-of-Sequence, `[EDIT]` or `[RETURN]`, the latter two marking the start and end of an edit, analogous to how a cursor and selection work in a text editor. The cursor head is a two-headed pointer network (an attention layer with just the query/key heads) on top of the final hidden states of the transformer backbone that predict an edit-start index S and an edit-end index E. The edit starts and ends immediately *before* the tokens at indices S and E. While S and E are defined in the *generated order*, the edit indices s and e in the corresponding current *display* sequence are guaranteed to satisfy $s \leq e$. This single operation covers insertion ($s = e$), deletion ($s < e$ with no replacement tokens), and substitution ($s < e$ with replacement tokens).

The edit start cursor is sampled simultaneously when an `[EDIT]` is emitted, and the end cursor is sampled immediately following the `[EDIT]` (rather than at `[RETURN]`). This defines the edit span (the tokens in the context that are selected for edits) so that the model can perform its edit with the end state in mind. Note that unlike Fill-in-the-Middle (Bavarian et al., 2022), where the edit span is supplied as input, the model learns to decide the location in the context to be edited.

`[EDIT]` moves the cursor to position s within the current *display* sequence, and shifts the positional encoding of existing tokens so that the newly generated token always has a relative position of 1. The `[EDIT]` token itself is initially positioned at the end of the current *display* sequence, but is moved to the edit frontier once the edit starts, just after the tokens to be removed, as shown in Figure 1.

`[RETURN]` returns the cursor to the end of the current *display* sequence and reassigns the post-edit positional encodings to tokens. This is when $e - s$ tokens are removed from the context. Removed

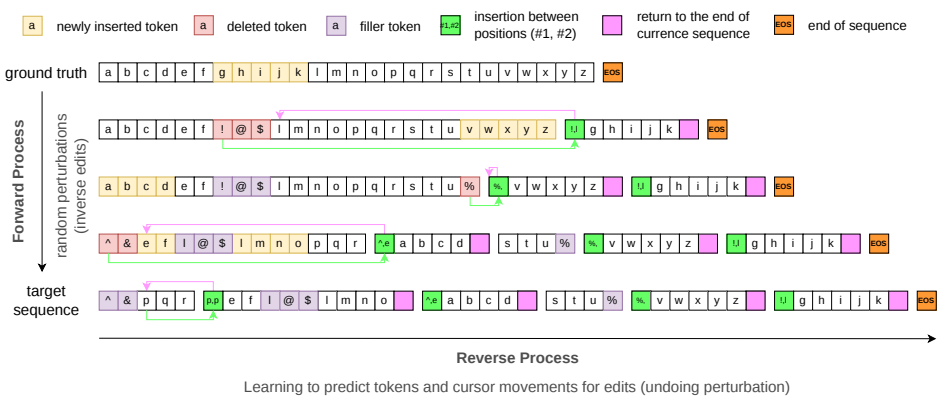

Figure 3: A simulated edit sequence for training. The ground-truth clean sequence (top) in display order is randomly perturbed to obtain a target sequence (bottom) with edit/return markers, which the model learns to generate. A sequence parser recovers the intended display order of the tokens.

tokens are masked from future attention and omitted from the decoded output rather than physically deleted from the KV cache. This also allows hidden edit spans to serve as *registers* that store temporary computation (Goyal et al., 2024; Darcet et al., 2024; Pfau et al., 2024), or as drafts and intermediate plans like Chain of Thought (Wei et al., 2022) that are not a part of the final output.

2.2 Training via Simulated Edit Sequences

At training time, AutoIndexer turns a clean sequence $\boldsymbol{x} = (x_0, \ldots, x_{T-1})$ into a perturbed target sequence by applying the *inverse* of a randomly sampled edit program. We sample n edit operators $\mathcal{E}_0, \ldots, \mathcal{E}_{n-1}$ with randomly drawn (Poisson-distributed) insertion lengths, deletion lengths, cursor spans $[s_i, e_i)$, and post-edit append lengths. Each edit replaces the context tokens in the range of $[s_i, e_i)$ in the display order with newly generated tokens in the `[EDIT]` … `[RETURN]` block, dropping the markers after completion. The operators are numbered in the order of generation, so $\mathcal{E}_0$ is the first edit and $\mathcal{E}_{n-1}$ produces the final output $\boldsymbol{x}$.

An inverse edit operation $\mathcal{E}_i^{-1}$ undoes this by re-injecting the edit block. Draft tokens that were later edited out do not appear in the parsed output. We fill these positions with independently sampled valid tokens, so the model learns to detect and remove erroneous context without those tokens affecting the decoded target.

The target sequence τ is built by starting from the clean sequence $\boldsymbol{x}$ in *display* and undoing the edits one at a time, from last to first:

$$\tau = \mathcal{E}_0^{-1} \circ \cdots \circ \mathcal{E}_{n-1}^{-1}(\boldsymbol{x}). \tag{1}$$

This procedure is described in Figure 3. The way clean data are corrupted in the forward process and the model is trained to invert the corruption in the reverse process is analogous to the diffusion training objective, except that the corruption is a discrete edit program rather than masking or Gaussian noise. Since τ can be constructed from any sequential dataset, no human edit history is required, though real histories such as git diffs can be useful for task-specific fine-tuning.

2.3 Position Reassignment via Chain-of-Edits Rotary Position Encoding

A technical challenge in training the model to perform multiple edits in one forward pass is how to handle positional encoding for tokens in the context. Since the model does not know the exact *display order* of the tokens to be generated at inference time until all edits are completed, the token positions must be reassigned after each edit operation. Emulating this positional shift over multiple edits for training is non-trivial, however, since the RoPE (Su et al., 2024) trick of factorizing the $T \times T$ relative position matrix into query/key rotations only works for static token sequences.

To solve this issue, we generalized RoPE to a *Chain of Edits*, factorizing the attention computation into edit-wise segments whenever the token positions are shifted. Each segment corresponds to a contiguous token subsequence over which the token positions monotonically increase. The segment

is either a normal segment (normal next token generation) or an in-edit segment containing the replacement tokens between `[EDIT]` and `[RETURN]`. For a normal segment, a standard RoPE is applied. For an in-edit segment, as mentioned in Section 2.1, `[EDIT]` signifies an ongoing edit and will be assigned a relative position of 1. All tokens preceding the edit cursor will have a normal RoPE applied, and all tokens succeeding the cursor will maintain the same position throughout the edit, meaning that the rotations are no longer relative to the query position for post-cursor tokens.

For every segment, we compute and store (i) display positions $\boldsymbol{\theta}_q$ for the query tokens in the block, (ii) display positions $\boldsymbol{\theta}_k$ for all previously generated visible key tokens, (iii) an attention mask m_{attn} that excludes removed tokens and resolved edit/return markers, and (iv) a cursor-anchor mask m_{anc} for keys *after* the cursor in the display order when an edit is underway. Where the cursor-anchor masks are applied, only the key is rotated and the query remains unrotated. The segment-wise attention is:

$$\text{attn_weight}(\boldsymbol{q}, \boldsymbol{k}, \boldsymbol{\theta}_q, \boldsymbol{\theta}_k, m_{\text{anc}}, m_{\text{attn}}) = (\boldsymbol{q}_{\text{rope}} \cdot \boldsymbol{k}_{\text{rope}} \odot \overline{m}_{\text{anc}} + \boldsymbol{q} \cdot \boldsymbol{k}_{\text{rope}} \odot m_{\text{anc}}) \odot m_{\text{attn}}, \quad (2)$$

where $\boldsymbol{q}_{\text{rope}}, \boldsymbol{k}_{\text{rope}}$ denote the RoPE-rotated query/key. The expression reduces to standard RoPE attention in the absence of edits, since m_{anc} will be false throughout.

By decoupling the token's fixed *generation order* from its more flexible *display order* via the edit mechanism, AutoIndexer enables exact KV-caching even when the token *display order* is non-monotonic. Keys and values are cached once in unrotated form to allow KV cache reuse, and the rotary angle is applied when attention is computed.

Although the edit operations are sequential, AutoIndexer does not train by replaying edits one at a time. Our custom fused attention kernel evaluates the complete packed sequence in one forward pass. It consumes the full query/key tensors, segment boundaries, and position tuples $(\boldsymbol{\theta}_q, \boldsymbol{\theta}_k, m_{\text{anc}}, m_{\text{attn}})$ per segment, and fuses RoPE, masking, dot product, softmax, dropout, and value aggregation without materializing the full attention matrix. More details about this kernel are described in Appendix A.1.

2.4 Decoding and Sequence Parsing

At inference time, AutoIndexer samples a sequence $\hat{\tau}$ autoregressively, with KV-caching for both token and cursor heads. At each step, a dedicated marker head samples a marker class of `[NORMAL]`, `[EDIT]`, `[RETURN]`, or `[EOS]`. Illegal classes are masked to $-\infty$. A normal vocabulary head sampling takes place when `[NORMAL]` is sampled. A decoding constraint enforces alternating `[EDIT]` and `[RETURN]` markers, so each edit is closed before another begins. When `[EDIT]` is emitted at display position t, the cursor head samples an edit-start index s ($0 \le s \le t$), and an edit-end index e ($s \le e \le t$) is sampled at the immediate following step (after the edit cursor and token positions have been updated). A sequence parser updates the token display-position mappings $\boldsymbol{\theta}_q$ and $\boldsymbol{\theta}_k$, and masks m_{anc} and m_{attn}, as required in Section 2.3. When `[EOS]` or the maximum length is reached, the parser masks out deleted spans and emits the remaining tokens in display order.

The prediction is recovered by the sequence parser by iteratively applying edits as they are generated:

$$\hat{\boldsymbol{x}} = \mathcal{E}_{n-1} \circ \cdots \circ \mathcal{E}_0(\hat{\tau}). \quad (3)$$

Since $\mathcal{E}_i \circ \mathcal{E}_i^{-1} = \text{id}$ by construction, parsing the synthetic sequence τ recovers $\boldsymbol{x}$ exactly.

2.5 Training Objective

The perturbation program described in Section 2.1 yields both the perturbed target sequence τ and the target cursor positions (S_i, E_i) for each edit. Thus, both token prediction and edit localization are self-supervised. The model's token head is trained on $\mathcal{L}_{\text{token}}$, a standard next-token cross-entropy over τ, ignoring padding and masked tokens. The marker head is trained with a focal loss $\mathcal{L}_{\text{marker}} = -(1-p_t)^{\gamma}\log(p_t)$, where p_t is the predicted probability for the target marker class, to classify the marker class at each step, taking into account the class imbalance amongst `[NORMAL]`, `[EDIT]`, `[RETURN]`, and `[EOS]`. The start- and end-cursor heads are trained on $\mathcal{L}_{\text{cursor}}$, a cross-entropy loss over targets S_i and E_i per edit, evaluated immediately before and at each `[EDIT]` marker, respectively. The total training loss is $\mathcal{L} = \mathcal{L}_{\text{token}} + \mathcal{L}_{\text{marker}} + \mathcal{L}_{\text{cursor}}$. A stop-gradient mixing ratio controls how much the cursor loss updates the shared transformer backbone. In practice, we found that it is better to detach the cursor head gradients entirely from the final hidden states to learn a better latent representation first, unaffected by the cursor objective.

Table 2: Pre-training benchmark results. All models were evaluated under the same protocol. Numbers in parentheses are the number of few-shot examples used for in-context learning. The models with * were finetuned with 100K additional CPT training samples (∼200M tokens). **Bold** shows the best score out of flexible-order decoding models, and the underline shows the best overall.

Model	**Qwen3 8B**	**Qwen3 8B LoRA***	**LLaDA 8B**	**Dream 7B**	**AutoIndexer 8B LoRA***
Decoding strategy	AR	AR	Diffusion	Diffusion	CoE (Chain of Edits)
Editable context	No	No	Limited	Limited	Yes
General Tasks					
MMLU (5)	76.7	77.0	65.9	69.5	**76.9**
ARC-Challenge (0)	56.7	60.8	45.3	**59.7**	59.6
Hellaswag (0)	78.7	77.7	69.4	72.5	**77.5**
WinoGrande (5)	76.9	76.2	70.4	72.7	**76.1**
PIQA (0)	79.6	80.4	70.4	72.0	**79.8**
Mathematics & Science					
GSM8K (5)	85.5	85.0	71.3	75.1	**84.3**
MATH500 (4)	55.2	56.4	31.4	42.8	**55.0**
Code					
HumanEval (0)	45.7	51.2	10.4	0.6	**59.2**
– with instruction	81.7	81.1	36.0	56.7	**79.3**
MBPP (3)	68.4	67.6	39.4	54.4	**65.8**

Since AutoIndexer loads a standard transformer backbone, using the same attention layers apart from injecting the segment position tuples and the fused attention kernel in Section 2.3, the backbone can be LoRA-finetuned (Hu et al., 2022) from pre-trained models. Learnable components in addition to the backbone are the cursor head and the marker head.

3 EXPERIMENTS

We evaluated AutoIndexer along two axes: retention of LLM capabilities in standard natural language and coding tasks, and the newly acquired error correction capabilities. First, we benchmarked AutoIndexer against autoregressive (Yang et al., 2025) and diffusion (Nie et al., 2025; Ye et al., 2025) models on standard general, math/science, and code tasks in Gao et al. (2024), both for base tasks and instruction tasks. Diffusion models tend to trade off prediction quality for the advantage of any-order decoding, due to KV-cache approximation. Benchmarking on standard LLM tasks allows us to test whether AutoIndexer maintains prediction quality, even with the added Chain-of-Edits capability. Next, we evaluated AutoIndexer's capability of active revision. Using the same base checkpoint used for LLM benchmarking, we ran a targeted edit diagnostic that tests whether the model can detect a corrupted span in the context, localize it, generate a suitable replacement, and resume generation coherently. Experimental details, hyperparameters, and data mixes are given in Appendix A.

LLM benchmark protocol. All results were obtained with the dLLM (Zhou et al., 2026) evaluation stack, built on the LM Evaluation Harness (Gao et al., 2024). We trained and evaluated two checkpoints, one for the base tasks and one for the instruction tasks. AutoIndexer 8B LoRA Base was LoRA-finetuned (Hu et al., 2022) from the Qwen3 8B Base (Yang et al., 2025) checkpoint, while AutoIndexer 8B LoRA Instruct was finetuned from the instruction-finetuned Qwen3 8B checkpoint.

For the pre-training evaluation, we used base-model tasks from MMLU, ARC-Challenge, HellaSwag, WinoGrande, PIQA, GSM8K, MATH500, HumanEval, and MBPP. Generation was capped at 1024 tokens. The diffusion baselines were LLaDA 8B Base (Nie et al., 2025) and Dream 7B Base (Ye et al., 2025), and the autoregressive baseline was Qwen3 8B Base. We additionally included a Qwen3 8B LoRA model trained on the same Continued Pre-Training (CPT) datamix (Appendix A.4) as AutoIndexer 8B LoRA Base to control for the effect of fine-tuning itself. AutoIndexer's outputs were scored after parsing its generated edit stream back into display order. For the post-training evaluation, we used instruction-formatted task variants from ARC-Challenge, MMLU-Pro, GPQA-Diamond, GSM8K, MATH500, HumanEval, and MBPP, with responses rendered through each model's chat template. Generation was capped at 512 tokens. The diffusion baselines were LLaDA 8B Instruct and Dream 7B Instruct, and the autoregressive baseline was the instruction-tuned Qwen3 8B checkpoint.

Table 3: Post-training (instruction-tuned) benchmark results. All models were evaluated instruct-formatted, under the same protocol. Numbers in parentheses are the number of few-shot examples used for in-context learning. The models with * were supervised-finetuned with the same 100K. **Bold** shows the best score out of flexible-order decoding models, and the underline shows the best overall.

Model	**Qwen3 8B**	**Qwen3 8B LoRA***	**LLaDA 8B Instruct**	**Dream 7B Instruct**	**AutoIndexer 8B LoRA***
Decoding strategy	AR	AR	Diffusion	Diffusion	CoE (Chain of Edits)
Editable context	No	No	Limited	Limited	Yes
General Tasks					
ARC-Challenge (0)	90.61	90.70	86.26	88.23	**90.61**
MMLU-Pro (0)	42.38	47.90	32.65	44.39	**47.90**
Mathematics & Science					
GPQA-Diamond (5)	35.35	30.81	20.71	**35.86**	30.81
GSM8K (5)	86.96	82.11	75.74	80.59	**80.67**
MATH500 (4)	52.20	49.20	17.00	45.80	**49.00**
Code					
HumanEval (0)	83.54	73.17	45.12	52.44	**75.61**
MBPP (3)	67.40	7.60	37.60	58.00	**64.00**

The matched Qwen3 8B LoRA and AutoIndexer 8B LoRA models were further fine-tuned from the Qwen3 8B checkpoint on the Supervised Fine-Tuning (SFT) instruction mix (Appendix A.6).

Pre-training benchmark results. Table 2 shows the base model evaluation results for general, math, science, and coding tasks. AutoIndexer inherits the Qwen3 backbone's strong performance in downstream tasks even after conversion into a Chain-of-Edits model, consistently outperforming competing diffusion baselines for flexible-order decoding. The AutoIndexer model essentially tied with the matched Qwen3 LoRA baseline, suggesting that the introduction of marker and cursor heads, and token reordering due to Chain-of-Edits do not measurably affect performance compared to ordinary left-to-right decoding, in contrast to other diffusion baselines where degradation of performance is common. Notably, AutoIndexer outperformed all other baselines by a wide margin for the HumanEval evaluation without instructions, where diffusion models especially struggled. This suggests that the edit-training objective transfers especially well to code, possibly because iteratively revising a partial solution resembles how code is naturally produced.

Post-training benchmark results. Table 3 shows the evaluation results of supervised-fine-tuned models on instruction tasks. The results follow the same pattern of the AutoIndexer model outperforming the instruction-tuned diffusion baselines, and matching the Qwen3 8B LoRA baseline performance. Interestingly, there seems to be a more noticeable drop in performance of the LoRA-finetuned Qwen3 and AutoIndexer models compared to the original Qwen3 8B checkpoint for math/science and code tasks, despite the substantial share of math and coding samples in the instruction-following SFT datamix (Appendix A.6). General tasks such as ARC-Challenge and MMLU-Pro, on the other hand, had a performance boost after LoRA-finetuning. We hypothesize that the Qwen3 instruction-finetuned checkpoint additionally benefited from Reinforcement Learning with Verifiable Rewards (RLVR), especially for maths and coding tasks, whereas we did not perform RLVR fine-tuning in the post-training stage. Nevertheless, the observation holds that our AutoIndexer model maintains superior instruction-following capabilities over the diffusion baselines.

Edit-capability evaluation setup. To verify that AutoIndexer can actively revise its context during generation, we took the AutoIndexer 8B LoRA Base checkpoint obtained from CPT, and ran it against the held-out validation set of the CPT datamix (Appendix A.4) with one corruption. We tested the model on insertion (by removing tokens from the ground truth), deletion (by adding noise tokens), substitution (by replacing tokens with noise), and no-op (the context was the ground truth). We inspected the checkpoint both under a teacher forcing setup to measure the model's edit initiation and termination decisions, localization of the edit span (start and end cursor), and replacement token accuracy, and a free generation setup where the model was rolled out from a corrupted draft and a 64-token budget to test whether the model can fully execute edits on its own.

Edit-capability evaluation results. Table 4 shows the `[EDIT]` and `[RETURN]` emission probabilities and the start and end cursor locations (edit span) accuracy in a teacher-forced setting. The

Table 4: Teacher-forced edit execution with oracle-placed markers (window of 512 tokens). Measures the increase in $P(\text{EDIT})$ at the synthetic trigger position against the same position in a clean sequence, as well as the edit start and end localization, content improvement, and edit termination accuracy.

Measurement	Insert	Delete	Substitute
$P(\text{EDIT})$ at trigger (corrupted / clean)	0.045 / 0.027	0.088 / 0.026	0.090 / 0.027
– Elevation over clean control	$\sim$1.7$\times$	$\sim$3.4$\times$	$\sim$3.4$\times$
– Paired win-rate ($P(\text{EDIT}\|\text{CORRUPT}) > P(\text{EDIT}\|\text{CLEAN})$)	90%	100%	100%
Correct edit start	40%	76%	84%
Correct span (both start and end)	38%	69%	74%
Replacement token accuracy	76%	–	80%
$P(\text{RETURN})$ at correct stop	72%	84%	73%

model detects an incoherency in the text and correctly assigns a higher edit probability to the corrupt sequence than the clean sequence in almost all cases. The edit emission probability at the exact timestep as the ground truth is low because of an arbitrary lag between an error and its correction. The model also identifies the edit spans for substitution and deletion with around 70% accuracy. The weaker insertion signal is expected, since missing text leaves no explicit cue, unlike for substitution and deletion cases where noise tokens are randomly sampled.

The free-generation results are shown in Table 5. The AutoIndexer model was able to correctly distinguish corrupt sequences from clean sequences with around 90% accuracy for the delete, substitute, and no-op cases, and predicted the correct edit span for nearly 70% of the delete and substitute cases, resulting in 53 exact repairs for deletion and 18 exact repairs for substitution. "Net improvement" is the number of edits that reduced the Levenshtein distance to the ground truth. Correction via insertion remains the most challenging task, since the lack of tokens is easy to miss. Less than half of corrupted drafts (43%) triggered an edit, but when an edit occurred, it localized the insertion correctly 60% of the time. Qualitative results in Appendix A.7.1 indicate that AutoIndexer employs a semantic and syntactic understanding of the context to apply informed edits, rather than randomly modifying its context. Visualization of the attention weights (Figure A.2) and cursor head (Figure A.3) show that the model is able to localize erroneous content and focus on the surrounding context.

4 Related work

Transformers Transformers (Vaswani et al., 2017) are currently the most versatile architecture family for generative modeling, widely adopted across numerous application domains (Achiam et al., 2023; Li et al., 2022b; Dosovitskiy et al., 2021; Radford et al., 2023; Jumper et al., 2021; Hu et al., 2023; Ishida et al., 2024; Black et al., 2024; Kanervisto et al., 2025). While full attention encoder models have been effectively used to extract context (Vaswani et al., 2017), learn feature representations (Devlin et al., 2019), or generate fixed-size outputs such as images (Peebles & Xie, 2023), autoregressive decoder models have been more widely adopted for generative modeling in

Table 5: Free-generation per-outcome counts (100 examples per category, marker head sampled with the default settings of temperature T_{mark}=1.0 and marker top-p=1.0).

Outcome / 100	Start cursor	End cursor	Insert	Delete	Substitute	No modification
Repaired exactly	✓	✓	10	53	18	–
No edit needed (true negative)	–	–	–	–	–	89
No edit emitted (false negative)	–	–	57	10	8	–
Correct span, improved	✓	✓	12	9	29	–
Correct span, neutral	✓	✓	2	1	11	–
Correct span, worse	✓	✓	2	6	8	–
Correct edit start, deleted too few	✓	×	–	6	10	–
Correct edit start, deleted too many	✓	×	4	4	6	–
Incorrect position	×	–	13	11	10	11
Correct edit emission	–	–	43	90	92	89
Correct edit span	–	–	26	69	66	–
Net improvement	–	–	+11	+66	+31	–

sequential domains such as language, time series, and videos, because they model the joint probability of the outputs more accurately and can produce outputs of arbitrary length.

Discrete Diffusion There have been efforts to apply continuous diffusion to LLMs by learning a mapping between discrete and continuous spaces (Li et al., 2022a; Dieleman et al., 2022; Lovelace et al., 2023; Jo & Hwang, 2025), or to adapt the diffusion formulation for discrete spaces (Austin et al., 2021; Campbell et al., 2022; Lou et al., 2024). Alternatively, several works (Sahoo et al., 2024; Shi et al., 2024; Nie et al., 2025) have proposed a Masked Diffusion Model approach that iteratively replaces token masks with token predictions. While such approaches are easy to implement and can even be adapted from pre-trained autoregressive models (Gong et al., 2025), they share the inherent limitation that the models can only generate a fixed-length output, and that tokens are positionally constrained once generated, which limits operations such as insertion, editing, and deletion.

Block Diffusion (Arriola et al., 2025) alleviates the first limitation by retaining a block-wise autoregressive structure while applying masked diffusion within each block using a full attention model, thereby trying to achieve the best of both worlds of using full attention blocks (i.e. parallelized token generation leading to speed up in generation) and an autoregressive mechanism (i.e. flexible output length and being able to take advantage of KV caching during inference for efficiency).

Any-order autoregressive generation Building on permutation language models (Yang et al., 2019), recent work revives any-order autoregressive factorizations to match diffusion's flexibility. Any-subset speculative decoding predicts and verifies multiple positions per step (Guo & Ermon, 2025), and A3 (Du et al., 2026) extends this to arbitrary token groups via two-stream attention. Unlike AutoIndexer, both still commit upfront to a fixed set of positions to fill, rather than growing or shrinking the sequence through edits.

Edit-based sequence generation Classic insertion/deletion decoders (Stern et al., 2019; Gu et al., 2019; Chan et al., 2019) and their diffusion counterpart, DiffusER (Reid et al., 2023), have recently been generalized further: Edit Flows (Havasi et al., 2025) casts edit-based generation as a continuous-time Markov chain trained via flow-matching, and Beyond Masks (Ding et al., 2026) makes deletion/insertion the diffusion process itself, removing mask and pad tokens entirely. All of these, however, remain non-autoregressive and cannot reuse a KV cache, since token positions shift between steps. Two concurrent works keep edits strictly autoregressive: Reviser (Diab, 2026) predicts cursor actions (insert, move, stop) over a mutable canvas, and Stream of Revision (Yang et al., 2026) lets a code model backtrack via dedicated action tokens. Our method differs in that it uses a pointer network to localize edits, and adjusts the positional encoding after each edit, enabling continuous revision and extension without re-computing the KV.

5 Conclusion

AutoIndexer enhances autoregressive models with active context editing and refinement capabilities. It decouples the generation order of tokens and the displayed order of tokens, allowing the model to revisit earlier drafts and iteratively refine the output sequence. The experiments suggest that this edit capability can be introduced while preserving the broad capabilities of the autoregressive backbone. Unlike other editing methods such as tool-calling and Fill-in-the-Middle, the model and training strategy are architected to account for context editing, such that the attention implements edit-aware positional encodings, and the KV-cache at inference exactly matches the training distribution, enabling the model to perform multiple edits without re-computing previous forward passes.

Chain-of-Edits decoding is especially useful in domains where consistency and internal cross-referencing are solidified over iterative refinement, including code (e.g. finetuning the model directly on git diffs), long-form writing, mathematical derivations, and structured documents. It could also be compatible with drafting and planning, commonly seen in Chain-of-Thought and tool calling, where intermediate thought or tool call traces can be discarded before the final output. AutoIndexer is an initial demonstration of causal transformers with continual context editing capabilities, suggesting a path toward generative systems that have the entire loop of planning, self-correction, and in-place refinement of their outputs built into their optimization objective.

AI USE STATEMENT

In this work, we used generative AI tools to design and provide feedback on research methodology or experiments, implement methods, clean and reformat dataset, and interpret results. We have not used generative AI tools to generate synthetic data sets, help develop theoretical models or conceptual frameworks, propose or refine hypotheses, assist with translation, and support qualitative and thematic data analysis. AI use to formulate mathematical claims, provide critical ingredients for proving mathematical claims, and assist in the writing of proofs are not applicable to this work. Additionally, we used generative AI tools to suggest experimental parameters, create or edit software code, creation of artifacts, draft parts of a research paper, identify relevant literature, and analyze existing literature. We have reviewed all AI-assisted work. The core algorithm was rigorously and manually written and tested by the authors, and accelerations made by AI tools were cross-checked with the original manual implementation via unit tests. Most of the literature review was also done by the authors manually before extending and refining the survey with the help of AI tools. We manage our code development via pull requests which are peer-reviewed by other authors before approval. All parts of the manuscript were proof-read by the authors. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

The code that implements the AutoIndexer method along with the fused accelerated attention kernels and training scripts are attached as supplementary material, and will be released open-sourced. The anonymized project page at `https://anonymous.4open.science/w/autoindexer-CE1F/` also supplies the code along with further qualitative examples of AutoIndexer evaluation. Details on the training datasets and evaluation frameworks are available in the Appendix.

REFERENCES

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

Marianne Arriola, Aaron Gokaslan, Justin T Chiu, Zhihan Yang, Zhixuan Qi, Jiaqi Han, Subham Sekhar Sahoo, and Volodymyr Kuleshov. Block diffusion: Interpolating between autoregressive and diffusion language models. In *The Thirteenth International Conference on Learning Representations*, 2025. URL `https://openreview.net/forum?id=tyEyYT267x`.

Jacob Austin, Daniel D Johnson, Jonathan Ho, Daniel Tarlow, and Rianne van den Berg. Structured denoising diffusion models in discrete state-spaces. *Advances in neural information processing systems*, 34:17981–17993, 2021.

Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, et al. Qwen technical report. *arXiv preprint arXiv:2309.16609*, 2023.

Mohammad Bavarian, Heewoo Jun, Nikolas Tezak, John Schulman, Christine McLeavey, Jerry Tworek, and Mark Chen. Efficient training of language models to fill in the middle. *arXiv preprint arXiv:2207.14255*, 2022.

Kevin Black, Noah Brown, Danny Driess, Adnan Esmail, Michael Equi, Chelsea Finn, Niccolo Fusai, Lachy Groom, Karol Hausman, Brian Ichter, et al. π_0: A vision-language-action flow model for general robot control. *arXiv preprint arXiv:2410.24164*, 2024.

Andrew Campbell, Joe Benton, Valentin De Bortoli, Thomas Rainforth, George Deligiannidis, and Arnaud Doucet. A continuous time framework for discrete denoising models. *Advances in Neural Information Processing Systems*, 35:28266–28279, 2022.

William Chan, Nikita Kitaev, Kelvin Guu, Mitchell Stern, and Jakob Uszkoreit. Kermit: Generative insertion-based modeling for sequences. *arXiv preprint arXiv:1906.01604*, 2019.

Timothée Darcet, Maxime Oquab, Julien Mairal, and Piotr Bojanowski. Vision transformers need registers. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=2dnO3LLiJ1.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pp. 4171–4186. Association for Computational Linguistics, 2019. doi: 10.18653/v1/N19-1423. URL https://aclanthology.org/N19-1423/.

Sean Diab. Reviser: Revision-capable text generation via autoregressive cursor actions, 2026. Manuscript and accompanying code release, https://github.com/Sean-Diab/Reviser.

Sander Dieleman, Laurent Sartran, Arman Roshannai, Nikolay Savinov, Yaroslav Ganin, Pierre H Richemond, Arnaud Doucet, Robin Strudel, Chris Dyer, Conor Durkan, et al. Continuous diffusion for categorical data. *arXiv preprint arXiv:2211.15089*, 2022.

Fangyu Ding, Ding Ding, Sijin Chen, Kaibo Wang, Peng Xu, Zijin Feng, Haoli Bai, Kai Han, Youliang Yan, Binhang Yuan, and Jiacheng Sun. Beyond masks: Efficient, flexible diffusion language models via deletion-insertion processes. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=VbvXjs5f72.

Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. In *9th International Conference on Learning Representations, ICLR 2021*. OpenReview.net, 2021. URL https://openreview.net/forum?id=YicbFdNTTy.

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 *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=vtDUomlazQ.

Leo Gao, Jonathan Tow, Baber Abbasi, Stella Biderman, Sid Black, Anthony DiPofi, Charles Foster, Laurence Golding, Jeffrey Hsu, Alain Le Noac'h, Haonan Li, Kyle McDonell, Niklas Muennighoff, Chris Ociepa, Jason Phang, Laria Reynolds, Hailey Schoelkopf, Aviya Skowron, Lintang Sutawika, Eric Tang, Anish Thite, Ben Wang, Kevin Wang, and Andy Zou. The language model evaluation harness, 07 2024. URL https://zenodo.org/records/12608602.

Shansan Gong, Shivam Agarwal, Yizhe Zhang, Jiacheng Ye, Lin Zheng, Mukai Li, Chenxin An, Peilin Zhao, Wei Bi, Jiawei Han, Hao Peng, and Lingpeng Kong. Scaling diffusion language models via adaptation from autoregressive models. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=j1tSLYKwg8.

Sachin Goyal, Ziwei Ji, Ankit Singh Rawat, Aditya Krishna Menon, Sanjiv Kumar, and Vaishnavh Nagarajan. Think before you speak: Training language models with pause tokens. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=ph04CRkPdC.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

Jiatao Gu, Changhan Wang, and Jake Zhao. Levenshtein transformer. *Advances in neural information processing systems*, 32, 2019.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

Gabe Guo and Stefano Ermon. Reviving any-subset autoregressive models with principled parallel sampling and speculative decoding. *arXiv preprint arXiv:2504.20456*, 2025.

Marton Havasi, Brian Karrer, Itai Gat, and Ricky T. Q. Chen. Edit flows: Variable length discrete flow matching with sequence-level edit operations. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025. URL https://openreview.net/forum?id=FXWwYz1p8a.

Anthony Hu, Lloyd Russell, Hudson Yeo, Zak Murez, George Fedoseev, Alex Kendall, Jamie Shotton, and Gianluca Corrado. Gaia-1: A generative world model for autonomous driving. *arXiv preprint arXiv:2309.17080*, 2023.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=nZeVKeeFYf9.

Shu Ishida, Gianluca Corrado, George Fedoseev, Hudson Yeo, Lloyd Russell, Jamie Shotton, Joao F Henriques, and Anthony Hu. Langprop: A code optimization framework using large language models applied to driving. In *ICLR 2024 Workshop on Large Language Model (LLM) Agents*, 2024. URL https://openreview.net/forum?id=JQJJ9PkdYC.

Jaehyeong Jo and Sung Ju Hwang. Continuous diffusion model for language modeling. In *Advances in Neural Information Processing Systems*, 2025. URL https://proceedings.neurips.cc/paper_files/paper/2025/hash/8c4046a93293442e88f54e19a5849d68-Abstract-Conference.html.

John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, et al. Highly accurate protein structure prediction with AlphaFold. *Nature*, 596(7873):583–589, 2021.

Anssi Kanervisto, Dave Bignell, Linda Yilin Wen, Martin Grayson, Raluca Georgescu, Sergio Valcarcel Macua, Shan Zheng Tan, Tabish Rashid, Tim Pearce, Yuhan Cao, et al. World and human action models towards gameplay ideation. *Nature*, 638(8051):656–663, 2025.

Xiang Li, John Thickstun, Ishaan Gulrajani, Percy S Liang, and Tatsunori B Hashimoto. Diffusion-lm improves controllable text generation. *Advances in neural information processing systems*, 35: 4328–4343, 2022a.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022b.

Aaron Lou, Chenlin Meng, and Stefano Ermon. Discrete diffusion modeling by estimating the ratios of the data distribution. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 32819–32848. PMLR, 2024. URL https://proceedings.mlr.press/v235/lou24a.html.

Justin Lovelace, Varsha Kishore, Chao Wan, Eliot Shekhtman, and Kilian Q Weinberger. Latent diffusion for language generation. *Advances in Neural Information Processing Systems*, 36: 56998–57025, 2023.

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. In *Advances in Neural Information Processing Systems*, 2025. URL https://proceedings.neurips.cc/paper_files/paper/2025/hash/48b383b24230e0e6e649d9c98dae4d8c-Abstract-Conference.html.

NVIDIA. NVIDIA CUTLASS documentation: CuTe-DSL, 2025. URL https://docs.nvidia.com/cutlass/media/docs/pythonDSL/cute_dsl_general/dsl_introduction.html. Accessed 2026.

William Peebles and Saining Xie. Scalable diffusion models with transformers. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, pp. 4195–4205, October 2023.

Jacob Pfau, William Merrill, and Samuel R Bowman. Let's think dot by dot: Hidden computation in transformer language models. *arXiv preprint arXiv:2404.15758*, 2024.

Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. Improving language understanding by generative pre-training. 2018.

Alec Radford, Jong Wook Kim, Tao Xu, Greg Brockman, Christine Mcleavey, and Ilya Sutskever. Robust speech recognition via large-scale weak supervision. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pp. 28492–28518. PMLR, 2023. URL https://proceedings.mlr.press/v202/radford23a.html.

Machel Reid, Vincent Josua Hellendoorn, and Graham Neubig. DiffusER: Diffusion via edit-based reconstruction. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=nG9RF9z1yy3.

Subham Sekhar Sahoo, Marianne Arriola, Aaron Gokaslan, Edgar Marroquin, Alexander M Rush, Yair Schiff, Justin T Chiu, and Volodymyr Kuleshov. Simple and effective masked diffusion language models. In *Advances in Neural Information Processing Systems*, volume 37, pp. 130136–130184, 2024. URL https://openreview.net/forum?id=L4uaAR4ArM.

Jiaxin Shi, Kehang Han, Zhe Wang, Arnaud Doucet, and Michalis K Titsias. Simplified and generalized masked diffusion for discrete data. In *Advances in Neural Information Processing Systems*, volume 37, pp. 103131–103167, 2024. URL https://openreview.net/forum?id=xcqSOfHt4g.

Mitchell Stern, William Chan, Jamie Kiros, and Jakob Uszkoreit. Insertion transformer: Flexible sequence generation via insertion operations. In *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pp. 5976–5985. PMLR, 2019. URL https://proceedings.mlr.press/v97/stern19a.html.

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

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 technical report. 2025. URL https://arxiv.org/abs/2505.09388.

Chengran Yang, Zichao Wei, Heminghao Deng, Jinfeng Jiang, Zhensu Sun, Ting Zhang, Tianyi Wu, Ming Wen, and David Lo. Autoregressive, yet revisable: In decoding revision for secure code generation. 2026. URL https://arxiv.org/abs/2602.01187.

Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Russ R Salakhutdinov, and Quoc V Le. Xlnet: Generalized autoregressive pretraining for language understanding. *Advances in neural information processing systems*, 32, 2019.

Jiacheng Ye, Zhihui Xie, Lin Zheng, Jiahui Gao, Zirui Wu, Xin Jiang, Zhenguo Li, and Lingpeng Kong. Dream 7b: Diffusion large language models. *arXiv preprint arXiv:2508.15487*, 2025.

Ted Zadouri, Markus Hoehnerbach, Jay Shah, Timmy Liu, Vijay Thakkar, and Tri Dao. Flashattention-4: Algorithm and kernel pipelining co-design for asymmetric hardware scaling. *arXiv preprint arXiv:2603.05451*, 2026.

Zhanhui Zhou, Lingjie Chen, Hanghang Tong, and Dawn Song. dLLM: Simple diffusion language modeling, 2026. URL https://github.com/ZHZisZZ/dllm.

A EXPERIMENTAL DETAILS

A.1 MAPPING CHAIN-OF-EDITS ATTENTION ONTO FLASHATTENTION-4 VARLEN

A naive implementation of the routing rule in Section 2.3 materializes two full $T \times T$ attention score matrices per segment, $\boldsymbol{q}_{\text{rope}} \cdot \boldsymbol{k}_{\text{rope}}$ and $\boldsymbol{q} \cdot \boldsymbol{k}_{\text{rope}}$, before m_{anc} selects which one each key uses – consuming $2\times$ the FLOPs of a conventional attention score matrix. Our optimized version instead routes the relative-attention path (attending only to keys scored with $\boldsymbol{q}_{\text{rope}}$) and the cursor-anchored attention path (attending only to keys scored with $\boldsymbol{q}$) to two separate FlashAttention-4 `varlen` calls (Zadouri et al., 2026), packing each path's keys into contiguous memory beforehand rather than masking a dense score matrix afterward; since every key belongs to exactly one path, the two calls together score the same keys the naive implementation would, so the sum of FLOPs across both calls approximates $1\times$ conventional attention rather than $2\times$. The two paths' outputs are recombined with a log-sum-exp reweighting of their softmax statistics, which reconstructs the same joint softmax that Section 2.3's formula defines, so this restructuring changes only how the computation is organized, not the attention weights it produces. To reduce the overhead of the routing and packing logic, we implemented another kernel in CuTe-DSL (NVIDIA, 2025), fusing the routing logic with the rotation logic.

A.2 MARKER HEAD DECODING

At the inference step, AutoIndexer uses temperature T_{voc}=0.7, top-p=0.8, and top-k=50 to sample from the vocabulary token head, matching the Qwen3 baseline. Marker emission is a separate categorical decision from the marker head's softmax over NORMAL, EDIT, RETURN, and EOS, with illegal classes masked to match alternating-edit constraints. Marker temperature and marker top-p can be set independently of the vocabulary head. In the reported runs both default to 1.0 (disabled). Before sampling, the EDIT logit is biased by $\lambda \log P_{\text{start}}(\hat{s}) + b$ with λ=1.0 and $b = 0.0$, where $\hat{s}$ is the start index sampled for that step. This is to ensure that edits with a confident edit localization are more likely to be triggered. Because markers are outside the vocabulary distribution, vocabulary top-p and top-k no longer affect whether an edit fires.

A.3 CONTINUED PRE-TRAINING (CPT) HYPERPARAMETERS

Both matched LoRA (Hu et al., 2022) fine-tuning runs start from the Qwen3 8B Base model and use bfloat16 weights. The fine-tuning stream contains 100K training samples truncated to 2048 tokens, corresponding to approximately 200M tokens before AutoIndexer perturbation expands the raw training sequence. The optimizer settings are AdamW-compatible Hugging Face trainer defaults with learning rate 10^{-4}, weight decay 0.01, gradient clipping at 1.0, and 1000 warmup steps. The

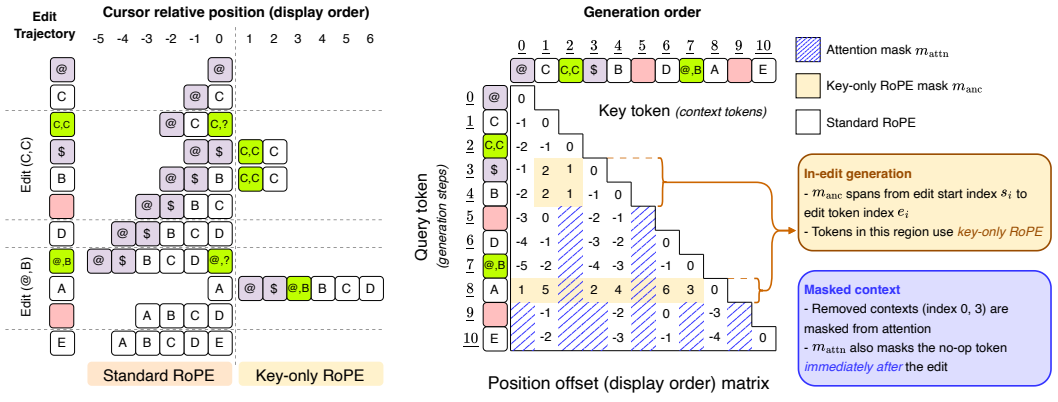

Figure A.1: Relationship between generation order and display order of tokens. The left-hand-side shows the generated tokens taking the display order as reference, whereas the right-hand-side shows the relative positions of the tokens, taking the generation order as reference.

Table A.1: CPT data mix used for the matched Qwen3 LoRA and AutoIndexer LoRA runs reported in Table 2.

Source	Dataset / subset	Weight
DCLM-baseline	`mlfoundations/dclm-baseline-1.0-parquet`	30
FineWeb-Edu	`HuggingFaceFW/fineweb-edu`	15
SlimPajama	`AlppAI/SlimPajama-chunked`	25
OpenWebMath	`open-web-math/open-web-math`	5
StarCoderData Python	`bigcode/starcoderdata/python`	7
StarCoderData JavaScript	`bigcode/starcoderdata/javascript`	4
StarCoderData Java	`bigcode/starcoderdata/java`	4
StarCoderData C++	`bigcode/starcoderdata/cpp`	3
StarCoderData TypeScript	`bigcode/starcoderdata/typescript`	2
StarCoderData C	`bigcode/starcoderdata/c`	2
StarCoderData Go	`bigcode/starcoderdata/go`	2
StarCoderData Shell	`bigcode/starcoderdata/shell`	1

validation budget is 6400 samples, with evaluation and checkpointing every 1000 steps. Generation during training uses Qwen3 base-model sampling settings with temperature 0.7 and top-$p = 0.8$.

For the matched Qwen3 LoRA baseline, we use rank $r = 16$, LoRA $\alpha = 64$, dropout 0.05, all linear modules as targets, no bias terms, and effective training batch size 8. For AutoIndexer 8B LoRA, we use the same rank, alpha, dropout, target-module, and bias choices. In addition to the LoRA adapter weights, AutoIndexer saves the cursor head and marker head, which are specific to edit localization and marker decisions.

We used eight NVIDIA H100 GPUs with 80 GB memory for fine-tuning with DeepSpeed ZeRO-2. Under that setting, the Qwen3 LoRA and AutoIndexer LoRA runs have an effective batch of 8 samples per optimizer step and therefore train for about 12,500 steps over 100K samples.

The AutoIndexer benchmark configuration uses perturbation probability 0.5, so half of training examples are converted into Chain-of-Edits sequences and the remainder are ordinary causal examples. The Chain-of-Edits model uses maximum raw sequence length 3072 to leave room for marker tokens and inserted content beyond the 2048-token clean examples. Synthetic edit sequences use mean number of edits 50, mean insertion length 5, mean deletion length 5, and mean extension length 40, with edits drawn uniformly at random from insertion, deletion, and substitution (a 1:1:1 ratio) rather than a fixed pure-edit fraction. Half of the edits are sampled with a short lag, with mean lag 5, and half remain uniformly placed to preserve long-range cursor supervision. The hidden state is detached before being passed to the cursor head, preventing the cursor head gradients from back-propagating through the shared backbone representation. The marker prediction loss uses a focal loss with focal parameter $\gamma = 1.0$ and weight 1.0.

A.4 Continued Pre-Training (CPT) Data Mix

The fine-tuning stream is an interleaved Hugging Face dataset mix. Weights are normalized to probabilities by the data loader, so the values below also give the approximate percentages used by the 100K-sample run. The composition is shown in Table A.1.

In aggregate, the mix assigns 30% weight to DCLM web text, 15% to educational web text, 25% to SlimPajama's diversified web/books/papers/Wikipedia/StackExchange mixture, 5% to math-heavy OpenWebMath, and 25% to source code from StarCoderData. The code portion is intentionally uneven across languages, with the largest share assigned to Python and smaller but nonzero shares for JavaScript, Java, C++, TypeScript, C, Go, and shell.

A.5 Supervised Fine-Tuning (SFT) Hyperparameters

The post-training results in Table 3 use the same LoRA (Hu et al., 2022) configuration, optimizer settings, and hardware as the CPT runs in Appendix A.3 (rank $r = 16$, LoRA $\alpha = 64$, dropout 0.05, all linear modules as targets, no bias terms, learning rate 10^{-4}, weight decay 0.01, gradient clipping

Table A.2: SFT instruction data mix used for the matched Qwen3 LoRA and AutoIndexer LoRA runs reported in Table 3.

Source	Dataset / subset	Weight
General-purpose instructions		
Alpaca	`tatsu-lab/alpaca`	15
Dolly-15k	`databricks/databricks-dolly-15k`	10
WizardLM Evol-Instruct-70k	`WizardLMTeam/WizardLM_evol_instruct_70k`	20
UltraChat-200k (multi-turn)	`HuggingFaceH4/ultrachat_200k`	25
Coding instructions		
CodeAlpaca-20k	`sahil2801/CodeAlpaca-20k`	10
Evol-Instruct-Code-80k	`nickrosh/Evol-Instruct-Code-80k-v1`	20
Magicoder OSS-Instruct-75k	`ise-uiuc/Magicoder-OSS-Instruct-75K`	20
Glaive-code-assistant	`glaiveai/glaive-code-assistant-v3`	20
Math instructions		
MetaMathQA	`meta-math/MetaMathQA`	20
General-capability replay		
DCLM-baseline (replay)	`mlfoundations/dclm-baseline-1.0-parquet`	10
StarCoderData Python (replay)	`bigcode/starcoderdata/python`	10

at 1.0, 1000 warmup steps, bfloat16 weights, eight NVIDIA H100 GPUs with DeepSpeed), but start from the instruction-tuned Qwen3 8B checkpoint (Yang et al., 2025) rather than Qwen3 8B Base, and are trained on the instruction mix in Appendix A.6 instead of the CPT mix. As with the CPT runs, responses are rendered through the chat template with thinking disabled, matching how the same tasks are scored at evaluation time. The AutoIndexer model configuration (perturbation probability, mean edit counts, edit-lag sampling, cursor head gradient ratio) is unchanged from Appendix A.3, except that edits are restricted to the response portion of each example so that the prompt is never perturbed, mirroring how the prompt is otherwise kept intact during instruction fine-tuning.

A.6 Supervised Fine-Tuning (SFT) Data Mix

As with the CPT mix in Appendix A.4, the SFT stream in Table A.2 is an interleaved Hugging Face dataset mix with weights normalized to probabilities by the data loader. The mix combines eight instruction sources spanning general-purpose and coding instructions (Alpaca, Dolly-15k, WizardLM Evol-Instruct-70k, UltraChat-200k, CodeAlpaca-20k, Evol-Instruct-Code-80k, Magicoder OSS-Instruct-75k, and Glaive-code-assistant) with math instructions (MetaMathQA) added because the original eight sources contain no math data. In aggregate, roughly 37% of the mix is general-purpose instructions, 37% is coding instructions, and 11% is math instructions.

The remaining $\sim$17% of the mix addresses a regression observed after LoRA SFT on only the instruction sources above: few-shot-scored evaluation tasks such as GSM8K (CoT) and MBPP (instruct) are scored by packing several solved exemplars and the real question into a single turn, a shape none of the instruction sources reproduce, since each is a single instruction with no in-context exemplars. Two few-shot-packed replay sources reproduce this exact shape at the same shot counts used at evaluation time (5-shot for GSM8K, 3-shot for MBPP; $\sim$9% combined). Two general-capability replay sources reuse raw CPT-style text (the same underlying corpora as Appendix A.4, restricted to DCLM web text and StarCoderData Python) trained with unmasked full-sequence loss rather than an instruction/response split, to keep some pressure on general and few-shot ability that a purely instruction-formatted mix does not otherwise reinforce ($\sim$7% combined).

A.7 Edit-Capability Diagnostic

The edit diagnostic evaluates the same AutoIndexer 8B LoRA model reported in Table 2 on the validation set of the data mix in Appendix A.4. Synthetic sequences use mean number of edits 50, mean insertion and deletion lengths 5, mean extension length 40, an equal 1:1:1 ratio between insertion, deletion, and substitution edits, perturbation probability 0.5, edit-lag probability 0.5 with mean lag 5, and cursor head gradient ratio 0.0. Teacher-forced and free-generation runs each use 100

windows per operation, with an average edit size of k=5 and one 64-token rollout per window for free generation.

A.7.1 QUALITATIVE EXAMPLES

The examples below are drawn from free-generation rollouts under the decoding defaults in Appendix A.2. Each card's header shows an outcome tag (**Exact repair** , **Partial repair** , or **Failure**) and the Levenshtein distance to ground truth before → after the edit. Corrupted spans are shown as «<...»>, with the actual corrupted tokens shown inside for the delete and substitute cases (the tokens a correct edit must remove). `[?]` stands in for tokens outside the Latin-1 range (e.g., CJK, Thai, Hebrew, emoji), which the typeset font cannot render. **`[EDIT` s`,`e`]`** marks the opening of an edit at cursor positions s, e in the display sequence, and **`[RETURN]`** marks the return of the cursor to the end of the stream.

Exact repair — pure insertion (10/100). Insertion triggers least often of the three operations (43/100), but when it fires close to the omission it can still repair exactly. Six words are missing from an HTML heading. The model localizes the empty span correctly—`[61, 61)`—and reconstructs the missing text verbatim.

Exact repair Levenshtein **6 → 0**

```
ORIGINAL  <h4 className="col-12 col-md-12 my-3 mx-auto contact-titles">Do
          you have a question or are you a hairstylist looking to join
          us?</h4>
DRAFT     <h4 className="col-12 col-md-12 my-3 mx-auto contact-t «<...»>
          question or are you a hairstylist looking to join us?</h4>
EMITTED   ...className="col-12 col-md-12[EDIT 61,61]itles">Do you have
          a [RETURN] mx-auto">...
RESOLVED  <h4 className="col-12 col-md-12 my-3 mx-auto contact-titles">Do
          you have a question or are you a hairstylist looking to join
          us?</h4>
```

Exact repair — pure deletion (53/100). Five garbage tokens were inserted into a C++ command-line option string. The model continues past the corruption, then reaches back and deletes exactly the injected span with no replacement:

Exact repair Levenshtein **5 → 0**

```
ORIGINAL  auto mask_img_path = op.add<popl::Value<std::string»("",
          "mask", "mask image path", "");
DRAFT     auto mask_img_path = op.add<popl::Value<std::string»("",
          "mask", "mask image «<[?]  soaking principalmente»> path", "");
EMITTED   ...::Switch [EDIT 129,134] [RETURN] >("s", "no-sleep", "do
          not sleep after processing");...
RESOLVED  auto mask_img_path = op.add<popl::Value<std::string»("",
          "mask", "mask image path", "");
```

Exact repair — substitution (18/100). Three garbage tokens replaced part of a Java import statement. The model locates the corrupted span and rewrites it to match the intended package path exactly:

Exact repair Levenshtein **3 → 0**

```
ORIGINAL  import org.netbeans.modules.editor.*; import
          org.netbeans.modules.lsp.client.LSPBindings; import
          org.netbeans.modules.lsp.client.Utils;
```

```
DRAFT     import org.netbeans.modules.editor.*; import org.netbeans
          «<enco(keys brutal)>» .client.LSPBindings; import
          org.netbeans.modules.lsp.client.Utils;
EMITTED   ...modules.editor.*; [EDIT 185,188] .modules.lsp [RETURN]
          import org.openide.text.Line;...
RESOLVED  import org.netbeans.modules.editor.*; import
          org.netbeans.modules.lsp.client.LSPBindings; import
          org.netbeans.modules.lsp.client.Utils;
```

Correct abstention — no modification (89/100). On most uncorrupted drafts the model continues fluently and terminates without ever emitting a marker:

Abstained Levenshtein 0 → 0

```
ORIGINAL  (this window is uncorrupted, so it is identical to the draft below)
DRAFT     { 1, 2 } >= { 1, 2 } true Every set is a superset of itself.
EMITTED   ...(no marker emitted)...
RESOLVED  (unmodified)
```

Partial repair — pure deletion (9/100). Six garbage tokens were inserted into the middle of "Embassies". The model finds the exact span but, instead of deleting it cleanly, writes a short unwanted continuation in its place:

Partial repair Levenshtein 6 → 4

```
ORIGINAL  network of counselor or representatives in French Embassies
          around the world (see joint map).
DRAFT     network of counselor or representatives in French Emb «<
          educated[?]  Identified[?]  yup»> assies around the world (see
          joint map).
EMITTED   ...responsible for:  - [EDIT 38,44] assies and Cons [RETURN]
          issuing permits...
RESOLVED  network of counselor or representatives in French Embassies and
          Consassies around the world (see joint map).
```

Partial repair — substitution (29/100). Three garbage tokens replaced part of a Go comment. The model finds the correct span but writes a shorter, malformed replacement, leaving a doubled word behind:

Partial repair Levenshtein 3 → 2

```
ORIGINAL  // Analysis is the method that kicks off all of the analysis
          plugins
DRAFT     // Analysis is the method that kicks «<bilità astronauts[?]»>
          the analysis plugins
EMITTED   ...used to kick off the analysis [EDIT 15,18] off the [RETURN]
          of files as they are written...
RESOLVED  // Analysis is the method that kicks off the the analysis
          plugins
```

Partial repair — pure insertion (12/100). More often, when insertion fires it localizes correctly but the content falls short. Here the model localizes a missing brace-and-assignment correctly—empty span `[38, 38)`—but writes a shorter replacement than the gold six-token insertion:

Partial repair Levenshtein 6 → 5

```
ORIGINAL  if(!transpose) { start = -1; step = 1; } else { start = k; step
          = -1; } } else { ...}
DRAFT     if(!transpose) { start = -1; step = 1; } else «<...»> step =
          -1; } } else { ...}
EMITTED   ...while(j != -1) [EDIT 38,38] { [RETURN] { q[i][j] = 0; j +=
          step; } }...
RESOLVED  if(!transpose) { start = -1; step = 1; } else { step = -1; } }
          else { ...}
```

Failure — wrong position for pure deletion (11/100). Six garbage tokens were inserted mid-expression in a C++ file, but the model's cursor lands one token late. The first corrupted token survives as an orphaned word rather than being removed:

Failure Levenshtein 6 → 1

```
ORIGINAL  auto &scores_dim = scores->dims();
DRAFT     auto &scores_dim = «< mutated Edit_Max.related):?> fullName»>
          scores->dims();
EMITTED   ...float *r [EDIT 218,223] [RETURN] pn_rois_data =
          rpn_rois->data<float>();...
RESOLVED  auto &scores_dim = mutated scores->dims();
```

Failure — wrong position for insertion (13/100). Two characters are missing from a hex color literal at position 67, but the edit does not fire until position 219, 152 tokens later, right at the end of the draft. The empty-content marker pair fires back-to-back and inserts nothing, leaving the gap unfilled:

Failure Levenshtein 2 → 2

```
ORIGINAL  gray:  '#949BA6', lightGray:  "#e8e8e9",
DRAFT     gray:  '#949 «<...»> ', lightGray:  "#e8e8e9",
EMITTED   ...export const theme = createTheme [EDIT 219,219] [RETURN]
          (colors, fonts) [EOS]
RESOLVED  gray:  '#949', lightGray:  "#e8e8e9",
```

Failure — wrong position for substitution (10/100). Twelve garbage tokens replaced a Java method-chain expression at position 42, but the model's oracle-free cursor lands one token late and cuts a shorter span (`[43, 49)` instead of the gold `[42, 54)`), leaving half of the garbage in place and stitching in an unrelated method call:

Failure Levenshtein 12 → 12

```
ORIGINAL  selector().get(BoxSelectionAspect.class).boxEnabled = false;
          selector().size.set(type.size);
DRAFT     selector().get(BoxSelection «<.middle[?]hots Leonard[?]-product
          ACC genome,default azureTextWriter pred»> .set(type.size);
EMITTED   ...getBlockTile(11, [EDIT 43,49] ().[RETURN] 3),
          orientation);...
RESOLVED  selector().get(BoxSelection.middle().  genome,default
          azureTextWriter pred.set(type.size);
```

Failure — spurious edit on a clean draft (11/100). The draft is uncorrupted, but the model fires anyway partway through its own continuation, cutting into text it just wrote and replacing it with something unrelated to the source:

Failure Levenshtein **0** → **6**

ORIGINAL	(this window is uncorrupted, so it is identical to the draft below)
DRAFT	`...less receptions but still WAY better numbers than Crabtree. Ever since the BCS was implemented, Strength of Schedule has become one of the biggest things in determining a team's grit and power....`
EMITTED	`...biggest things in determining a team's` **`[EDIT 46,46]`** `ranking. I think this is` **`[RETURN]`** `It is a bad thing...`
RESOLVED	`...one of the biggest things in determining a team's ranking. I think this is grit and power....`

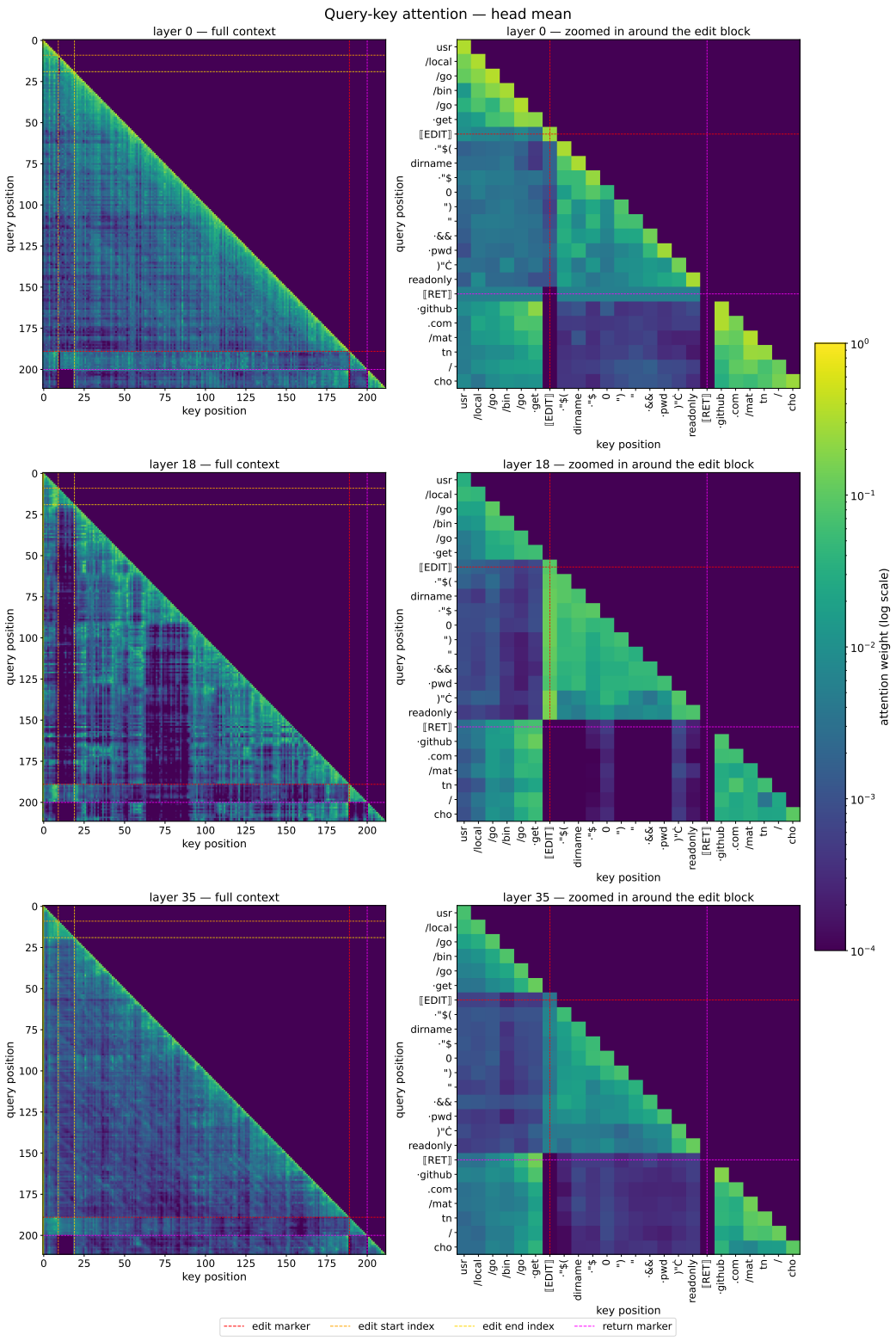

Figure A.2: Visualization of attention weights averaged over the attention heads. The left-hand-side shows the full attention weights, while the right-hand-side shows a close-up around the edit block (`[EDIT]` ... `[RETURN]`). `[EDIT]` and `[RETURN]` are masked out upon the completion of the edit, as well as the deleted tokens. It can be seen that attention is focused before and after the erroneous token region during the edit so that the newly inserted tokens are coherent with the context.

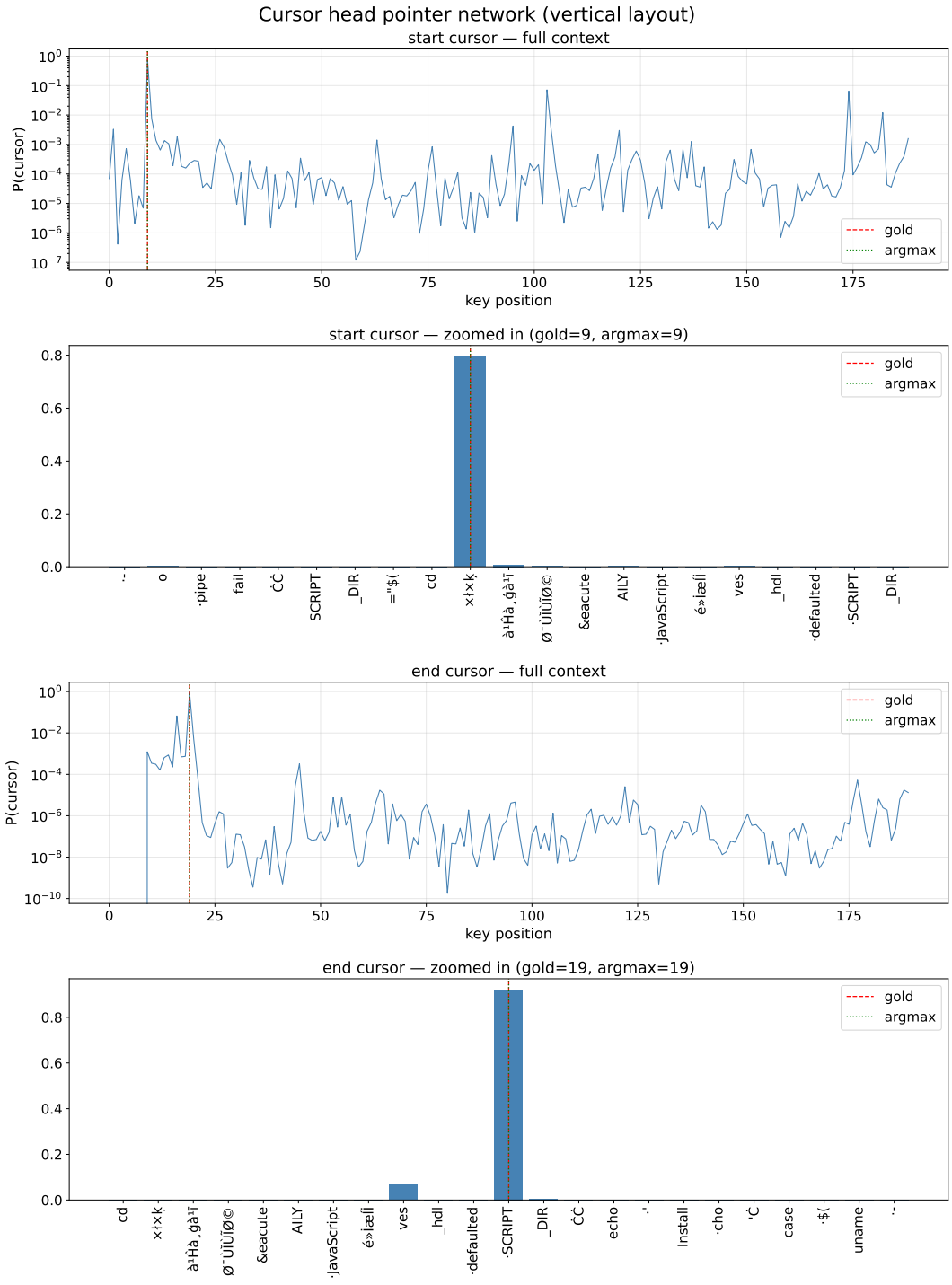

Figure A.3: Visualization of the cursor pointer network heads for edit start and edit end. The edit start head attends to all tokens in the previous context to identify the beginning of a potential erroneous content. The edit end head attends to all tokens following the edit start cursor to identify where the edit should end.