When and Where Can Learned Tokenization Be Refined at Test Time?

Anonymous authors
Paper under double-blind review

Abstract

Learned byte-level tokenizers expose an inference-time degree of freedom that fixed subword tokenizers do not: the model can vary how much of its learned latent resolution is used for an observed prefix. We study two forms of test-time refinement—*where* to restore boundaries inside a reduced native mesh and *when* to return a prefix to full native resolution. The experiments reveal a clear granularity asymmetry. Fine-grained boundary placement is unstable: dense insertion hurts and first-layer geometry or attention does not yield a reliable long-horizon location rule. Prefix-level allocation is substantially more predictable. A lightweight router that estimates continuation-likelihood gain per added latent improves over a strong gate-count allocation baseline by 0.00553 and 0.00716 BPB on two disjoint Python test sets at matched latent spend. Repetition and local code-structure features are especially useful for ranking prefixes, even though the corresponding signals are too weak to localize individual gates. This suggests that test-time adaptation of learned tokenization is better posed first as a compute-allocation problem over prefixes, and only second as a boundary-placement problem within a prefix.

1 Introduction

Fixed subword tokenization commits computation to a segmentation before a language model observes the prompt. Learned byte-level boundaries move that decision inside the model: a causal encoder chooses which byte positions become latent tokens, a shorter middle network processes those positions, and an upsampler returns to bytes (Dauncey & Wattenhofer, 2026; Hwang et al., 2025; Owodunni et al., 2025). At inference time, the prompt is already observed. It is tempting to restore boundaries in earlier parts of that prompt to improve predictions that come after it. Unlike changing the training-time average compression rate, this is a per-prompt decision about both the amount and location of computation.

We distinguish three action spaces. Dense insertion adds boundaries beyond the model's native gate set; empirically, this lowers continuation likelihood and is therefore a poor refinement strategy for the trained models we study. Within native support, a fixed-size action asks *where* a limited number of omitted states should be restored. At a coarser level, a budgeted action asks *when* an entire prefix should return from the reduced mesh to native resolution. The last two problems require different signals: location depends on the marginal value of individual states, whereas allocation depends on the aggregate value of resolution for the prefix as a whole.

We test this hierarchy with paired mesh interventions on independently trained byte U-Nets. The main empirical result is that the useful unit of adaptation is the prefix rather than the individual gate. Equal-count location rules lose most of their advantage over a 64-byte horizon, while a prefix router consistently identifies code examples on which native resolution is unusually valuable. Two disjoint Python test sets give gains of 0.00553 and 0.00716 BPB over an offline gate-count allocator at matched latent spend. JavaScript shows smaller gains and FineWeb does not reproduce the effect, pointing to code structure as an important source of routing signal. We also report prefill measurements to relate the latent-count objective to realized computation.

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.

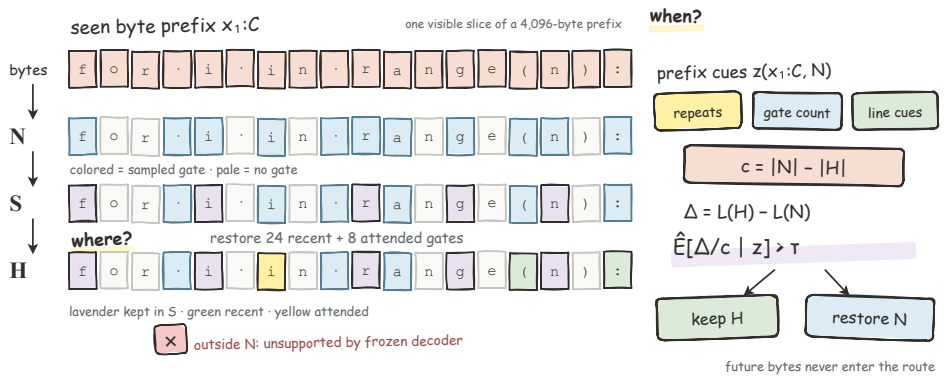

Figure 1: Two levels of test-time refinement. The location problem restores a fixed number of positions from the model's native support, while the allocation problem chooses between a reduced mesh and full native resolution using features of the observed prefix.

2 Native-support planning

2.1 First-layer attention geometry

For the final retained native query, the byte encoder and the first middle-layer query, key, and value projections are unchanged when a subset of native states is selected. For one attention head, let $e_i = \exp(s_i)$ denote the exponentiated score at native position i, including the model's positional transformation and score scaling. For a selected set $T \subseteq N$, its first-layer output is

$$o_T = \frac{\sum_{i \in T} e_i v_i}{Z_T}, \qquad Z_T = \sum_{i \in T} e_i, \qquad o_{T \cup \{j\}} - o_T = \frac{e_j}{Z_T + e_j}(v_j - o_T). \tag{1}$$

The last identity gives the exact effect of restoring candidate j on this query and layer; it requires no decoder replay for every candidate. If $R_T = o_N - o_T$ and $d_j = o_{T \cup \{j\}} - o_T$, the reduction in squared discrepancy from the native output is

$$\|R_T\|^2 - \|R_T - d_j\|^2 = 2\langle R_T, d_j \rangle - \|d_j\|^2. \tag{2}$$

Summing this quantity over heads yields a candidate geometry score. A development-set planner restores 32 positions in groups of eight. The identity is exact only for the specified first-layer query: later layers, other queries, upsampling, and output probabilities change with the selected mesh. Consequently, a smaller geometric residual need not mean lower continuation loss. Implementation parity checks are reported in Appendix H.

2.2 Attention mass and recency

We also consider a less expensive prefix-only signal: the attention mass assigned to each omitted native position by the final native query. Normalizing each head over the full native support gives

$$M_i = \sum_h \frac{\exp(s_{hi})}{\sum_{j \in N} \exp(s_{hj})}, \qquad i \in N \setminus S. \tag{3}$$

Mass measures where this query attends, but it is not the marginal effect in Eq. (1): that effect also depends on the value vector and the displacement caused by changing the softmax denominator. We therefore treat attention as a location heuristic, not as an estimate of predictive gain.

The code-domain hybrid first restores the 24 most recent omitted native positions, then assigns eight remaining slots to the highest-mass candidates outside that recent set. Recent-32 and attention-only-32 are equal-count controls. This comparison asks whether attention supplies useful nonlocal locations beyond a recency prior, without attributing any gain merely to a larger latent budget. The

hybrid needs only the first-layer query and keys; it requires only the final-query attention scores and no candidate-wise decoder replay. The batched implementation uses the same deterministic selection rule.

2.3 Prefix-conditioned action choice

Choosing between fixed meshes is a separate problem from constructing them. Write T_{hyb} for the 24+8 mesh, denoted H in Fig. 1, and R for recent-32. Their per-request equal-count advantage is $D = L_{64}(R) - L_{64}(T_{\mathrm{hyb}})$. If a prefix-only policy chooses the hybrid when $I(Z) = 1$, its expected gain over always using recency is

$$\mathbb{E}[ID] = \mathbb{E}[I]\,\mathbb{E}[D] + \mathrm{Cov}(I, D). \tag{4}$$

The first term is the gain obtained from selecting the hybrid at the same overall frequency without conditioning on the request. The covariance term therefore measures the value of prefix-conditioned action choice. We evaluate learned controllers against fixed-action baselines at matched action count.

2.4 Budgeted native escalation

The fixed 24+8 mesh poses a different decision: whether a particular prefix merits returning to its full native support. Let

$$\Delta = L_{64}(T_{\mathrm{hyb}}) - L_{64}(N), \qquad c = |N| - |T_{\mathrm{hyb}}| > 0.$$

Here $\Delta > 0$ means that escalation improves likelihood, while c counts its additional latent states. Let Z contain prefix-visible information, including the native and hybrid gate counts, so that $c = c(Z)$ is known when the decision is made. For a population budget B, the ideal prefix-only route solves

$$\max_{I(Z)\in\{0,1\}} \mathbb{E}[I(Z)\Delta] \quad \text{subject to} \quad \mathbb{E}[I(Z)c(Z)] \le B. \tag{5}$$

With $m(Z) = \mathbb{E}[\Delta \mid Z]$ and multiplier $\lambda \ge 0$, the part of the Lagrangian controlled by each decision is $I(Z)\{m(Z) - \lambda c(Z)\}$. Hence its population-optimal threshold rule, apart from ties, is

$$I_\lambda(Z) = \mathbf{1}\left\{\frac{m(Z)}{c(Z)} > \lambda\right\} = \mathbf{1}\{\mathbb{E}[\Delta/c \mid Z] > \lambda\}\,. \tag{6}$$

The second equality holds because $c(Z)$ is observed before the continuation. A zero-cost case requires no escalation decision: the reduced and native actions coincide. This derivation explains why the relevant prediction target is future gain *per added latent*, rather than raw gain or gate count alone.

We regress paired training labels Δ/c on ten cheap gate, early-head, and byte features and 25 fixed observed-code features measuring repeated suffixes, identifier recurrence, and local line structure. A ridge regressor and checkpoint-specific thresholds are chosen using repository-disjoint training and calibration groups. The threshold aims to spend 25% of the available incremental latent states on calibration data; 50% is a secondary operating point. Because the distribution of c and predicted values may change, the realized evaluation spend need not equal either target.

The route is compared with online count-only routing and with a stronger offline count baseline that ranks all evaluation prompts by native gate count at each run's *realized* spend. The latter sees the test batch's gate counts but no future outcomes. If expected native benefit were constant across requests, spending first on low-cost requests would be appropriate. An advantage over this baseline therefore isolates whether observed prefix structure predicts variation in the value of native resolution beyond its known latent cost. Wall-clock effects are evaluated separately.

3 Experimental design

We use the publicly released byte U-Net implementation of Dauncey & Wattenhofer (2026).[1] The main study uses two independently trained 147M-parameter backbones; three 73M instances probe

[1] https://github.com/SamD770/bitter-lesson-tokenization

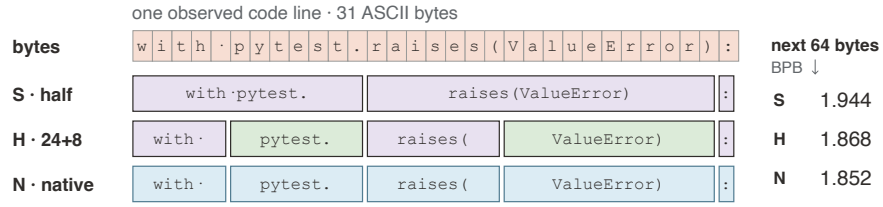

Figure 2: A concrete mesh intervention on one observed Python line. S is the half-native starting mesh, H is the 24+8 refinement, and N is full native support. On this visible line, H and N happen to coincide, although they differ elsewhere in the 4,096-byte prefix. Their next-64-byte BPB values are 1.944, 1.868, and 1.852 for S, H, and N, respectively, and the prefix router selects N. The example illustrates why routing is a full-prefix decision: local boundary appearance alone need not reveal the value of additional resolution.

rate stability. Each 147M run saw 6.00B training-byte exposures drawn from about 1.074B distinct FineWeb bytes (Penedo et al., 2024). The out-of-domain Python source is the deduplicated CodeParrot-clean collection, absent from foundation training.[2] Software versions, data construction, and checkpoint selection are documented in Appendix A.

Development data are used to choose the 24+8 location rule and fit the prefix router. We evaluate the location rule on a 4,096-file Python test set and the routing rule on a separate 4,096-file Python test set, with a second Python set used for replication and JavaScript/FineWeb used for transfer. Main quality experiments use $C = 4096$, $H = 64$, FP32 eager attention, paired model runs, and repository-clustered intervals for code (document resampling for web text). Prefill timing includes the controller and decoder.

Figure 2 makes the intervention concrete before we compare action families statistically. The visible line already shows that a local view of the mesh can be ambiguous: H and N can look identical in the excerpt while differing elsewhere in the prefix and producing different continuation likelihoods. This motivates evaluating location and allocation at the level of the full observed prefix rather than treating visually salient boundaries as sufficient evidence of value.

4 Empirical hierarchy of refinement actions

4.1 Unrestricted boundary insertion fails

Fully densifying the observed prefix lowers future likelihood relative to the native mesh across the 147M runs, tested prefix lengths, and code/web domains (Table 4). The sign is negative even under repository-clustered intervals for code. Small frozen-input and conditional LoRA adapters, as well as a longer middle/upper decoder continuation, reduced the damage but did not make the dense intervention beat the untouched native model. The failure motivates restricting test-time refinement to boundaries already selected by the model's native gate. A matched-update variable-rate continuation experiment also trails the untouched native-rate model in its best full-sequence diagnostic (Appendix).

4.2 Within-support location depends on domain

At $C = 1024$, equal-count geometry beats recent-32 on FineWeb by a paired 0.00235 BPB (development-set document interval $[0.00080, 0.00385]$), while full native remains better than either. The sign reverses on Python code: geometry minus recency is -0.01532 BPB (repository interval $[-0.02211, -0.00879]$). At $C = 4096$ the geometric advantage is also absent on FineWeb (-0.00177, interval $[-0.00504, 0.00149]$). Thus first-layer query geometry does not supply a stable location rule across domain and context length; model-level and gate-probability controls are in the appendix.

[2] https://huggingface.co/datasets/codeparrot/codeparrot-clean

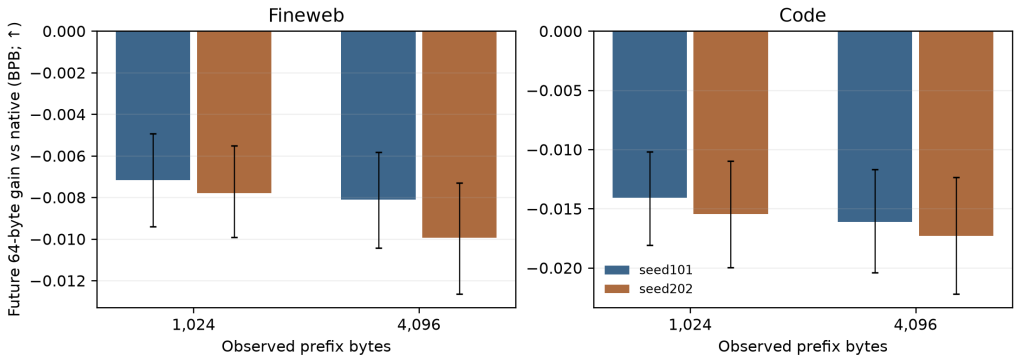

Figure 3: Dense-prefix effects on next-64-byte BPB on the model-selection data. Negative gain means the native mesh predicts better. FineWeb intervals resample documents; code intervals resample repositories.

4.3 Attention allocation on code

On separate code development data at $C = 4096$, the fixed attention/recent 24+8 rule improves continuation BPB relative to recent-32 by 0.00293 across the two backbones, with a repository-clustered interval $[0.00095, 0.00493]$ (Table 5). Pure attention-32 instead underperforms recency. This development-set pattern motivated testing whether a small part of a recency budget should be allocated to high-attention native positions rather than replacing recency outright. The hybrid still trails full native by about 0.045 BPB here; the web-text pilot has mixed directions. Together with the smaller-backbone variation, this suggests that fixed location heuristics are sensitive to domain and model state rather than forming a stable refinement rule.

4.4 Learned action choice and measured cost

On FineWeb, the STOP controller does not reliably beat a learned single-action geometry baseline at matched action count. On code, the equal-count 24+8-versus-recent-32 router gains only 0.00042 BPB over fixed 24+8 on separate development documents, interval $[-0.00080, 0.00167]$; lexical features and an observed-suffix validation rule do not yield a reliable gain. Additional controller comparisons are reported in the appendix.

For budgeted native escalation, a ten-feature predictor does not reliably beat count-only allocation. Adding the 25 observed-code features raises its direct development-set gain by 0.00638 BPB at nearly equal spend. On a separate code development partition, the fixed-threshold route beats an offline count ranking by 0.00660 BPB, descriptive interval $[0.00369, 0.00961]$. This is development evidence; Section 6 evaluates the fixed router on disjoint test data.

Exploratory ablation associates repetition cues with routing gain, yet a matched repeated-suffix witness placement rule gives $+0.00087$ BPB over 24+8, with repository interval $[-0.00262, 0.00419]$. Predicting *when* native resolution helps does not locate *where* to add individual gates; additional diagnostics are in the appendix.

Warmed single-request FP32 prefill averages about 10.4 requests/s for the route, between fixed 24+8 (10.9) and full native (9.6), with feature extraction included. An offline count allocation at no greater latent spend is faster (about 10.8 requests/s); ordinary mixed batches also lose some benefit to padding. The timing gap reflects both route overhead and padding in mixed-support batches; Appendix H analyzes these effects in more detail.

5 Where: location signals are short-range

The test comparison uses 24+8 with recent-32 at exactly the same latent count. Across two independent 147M backbones and 4,096 Python files (2,440 repositories), the next-64-byte gain is $+0.00130$ BPB with repository interval $[-0.00001, 0.00258]$ (Table 1). The interval crosses zero, substantially

weaker than the development result. Full native support predicts better than either reduced mesh, and the alternative attention and geometry selectors likewise provide no reliable H=64 improvement.

FineWeb shows the same pattern: $+0.00048$ BPB with document interval $[-0.00010, 0.00106]$. The horizon analysis is more informative about why: on code the gain falls from $+0.02157$ BPB over the first eight future bytes to $+0.00130$ over 64, indicating that the endpoint-query signal is strongest locally and fades over the continuation. Three smaller backbones also disagree on the sign of the rate effect. Appendix E reports the full horizon and checkpoint breakdown. These results make fixed *where* selection the fragile part of test-time refinement.

Table 1: Equal-count location test with two 147M backbones, $C = 4096, H = 64$. Positive gain means lower BPB for 24+8 than recent-32. Intervals pair model runs and resample repositories or documents.

Corpus	Gain (BPB)	95% interval
Python code	$+0.00130$	$[-0.00001, 0.00258]$
FineWeb text	$+0.00048$	$[-0.00010, 0.00106]$

6 WHEN: BUDGETED NATIVE ESCALATION

We train the budgeted router on repository-disjoint code development data and evaluate it on a separate 4,096-file Python test set containing 3,942 repositories. This study asks a different question from the location experiment: whether prefix structure can allocate a fixed native-resolution budget across requests.

Our main routing comparison uses the online route and a stronger *offline* count-only allocator that ranks the full test workload by native gate count. At each run's *realized* added-latent spend, the route improves next-64-byte BPB by 0.00553 across the two backbones (20,000-draw repository interval $[0.00362, 0.00748]$; Table 2). Both runs agree. The threshold targets 25% on calibration but spends 22.59% here; rematching the comparator prevents that shift from inflating the contrast. Relative to fixed 24+8, the route recovers 36.9% of the possible full-native quality gain with 22.6% of extra latent states, versus 25.8% recovered by offline count. Full native still predicts better by 0.03159 BPB. The 50% operating point is reported as a secondary budget setting.

Table 2: Routing results at the 25% calibration target, $C = 4096, H = 64$, paired across two 147M backbones. Gain is online route minus the stronger offline count-only allocation at each run's realized added- latent spend (reported mean). Lexical uses all fixed prefix features; Repeat uses the fixed repetition subset. Python-v2 and Python-v3 are disjoint Python test sets. Intervals resample repositories for code and documents for FineWeb.

Corpus	Route	Docs	Spend	Gain BPB [95% interval]
Python-v2	Lexical	4096	22.59%	$+0.00553$ $[0.00362, 0.00748]$
Python-v3	Lexical	4096	22.50%	$+0.00716$ $[0.00518, 0.00913]$
JavaScript-v1	Lexical	1024	22.40%	$+0.00380$ $[0.00009, 0.00759]$
JavaScript-v2	Repeat	1024	22.86%	$+0.00363$ $[0.00007, 0.00730]$
FineWeb-v2	Lexical	1024	15.72%	-0.00001 $[-0.00153, 0.00151]$

The same policy also improves BPB on the second Python test set, which contains 3,944 repository groups (Table 2). On the same teacher-forced target, next-byte top-1 accuracy rises by 0.097 percentage points (repository interval $[0.052, 0.142]$ percentage points); structural-byte accuracy rises by 0.227 points, while the first-byte interval spans zero.

Figure 4 asks whether the likelihood effects are visible in a discrete code decision. On the unadapted backbones, the route-versus-count contrast is small relative to its interval. After code adaptation, however, the native-versus-24+8 contrast is substantially larger and positive. Together with the BPB results, this suggests that latent resolution can affect local code decisions, while the benefit of the lightweight router is clearest in likelihood-based allocation. The probe is intentionally narrow: it measures constrained choice rather than end-to-end program generation.

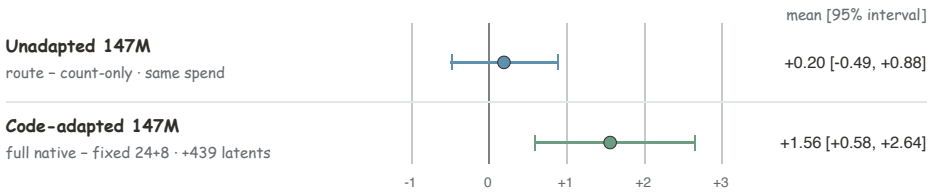

Figure 4: A downstream decision probe on code. For the unadapted 147M backbones, routing rather than count-only allocation at the same spend changes constrained-choice accuracy by $+0.20$ percentage points, with 95% interval $[-0.49, +0.88]$. For the code-adapted 147M model, full native resolution rather than fixed 24+8 changes accuracy by $+1.56$ points, with interval $[+0.58, +2.64]$. The two rows use different model/action contrasts and are therefore interpreted separately.

JavaScript transfer without refitting gives two small positive signals, although both interval lower bounds lie near zero; because the second shard uses a different fixed feature subset, we report the two results separately. FineWeb-v2 is near zero, consistent with the earlier web-text result. The routing advantage therefore appears tied to source-code structure rather than to a domain-general tokenizer effect.

Exploratory model-size check. We transferred the same fixed predictor and a shared threshold without refitting to three independently trained 346M backbones under a 6B-byte training exposure budget. On 1,792 paired development code files, the route beats offline count allocation at matched *realized* latent spend by 0.00696 BPB on average (repository interval $[0.00486, 0.00909]$); all three directions are positive. The nominal 25% calibration target, however, spends 22.8–52.1% across these backbones, showing substantial threshold drift with scale. At the secondary 50% target, one run reverses sign. The main scaling effect is therefore threshold drift: the learned ranking transfers better than the original budget calibration.

Why *when* is easier than *where*. Sections 5 and 6 expose a granularity mismatch. A location rule asks a first-layer statistic to rank individual omitted states, even though each restoration perturbs attention normalization and then passes through many nonlinear layers. The observed horizon decay is consistent with such endpoint-query signals being locally useful but weak for longer-range credit assignment. The routing problem instead treats full native support as one aggregate action and predicts Δ/c without assigning credit to individual gates. This aggregation reduces the credit-assignment burden: gate-level effects can cancel or interact after attention renormalization, while their combined value may still vary systematically with prefix structure. Empirically, the coarser decision is stable across the two Python test sets. The feature ablation points in the same direction: repetition cues help rank prefixes for escalation, yet the corresponding repeated-suffix placement rule does not reliably identify individual gates. This suggests a design principle for adaptive tokenizers: first learn *prefix-level compute allocation*; only then attempt finer within-prefix placement when the positional signal is stable at the evaluation horizon.

7 Related work

Compression is not the same as useful resolution. Subword methods such as byte-pair encoding and SentencePiece replace a long sequence of characters with a shorter sequence of learned units (Sennrich et al., 2016; Kudo & Richardson, 2018). This makes sequence length a convenient measure of cost, but it does not make the shortest segmentation the best representation. In a controlled comparison of tokenizers, Schmidt et al. (2024) find that minimizing token count does not, by itself, maximize downstream performance. The distinction matters even more for a learned internal tokenizer: a boundary determines which byte states receive global computation, so removing it changes both cost and the information available to the decoder. Our question is therefore not whether a pre-

fix can be represented with fewer latents. It is whether the value of restoring native latents varies predictably across prefixes after accounting for their cost.

Byte models and learned compression. Byte-level models avoid committing to a fixed subword vocabulary, while inheriting the computational burden of long input sequences. Xue et al. (2022) model bytes directly; Clark et al. (2022) downsample character states, and Tay et al. (2022) learns soft local blocks. Hierarchical decoders concentrate expensive computation on byte patches or coarser levels (Yu et al., 2023; Nawrot et al., 2022). SpaceByte places larger transformer blocks after selected bytes, including likely word boundaries (Slagle, 2024), whereas MambaByte explores a state-space alternative (Wang et al., 2024). Adaptive byte and chunk models go further by learning boundaries or rates together with the backbone (Pagnoni et al., 2025; Hwang et al., 2025; Owodunni et al., 2025; Deng et al., 2026; Dauncey & Wattenhofer, 2026); training across chunk sizes can also provide an inference-time rate control (Prakash et al., 2026). These works establish that resolution is an architectural choice with consequences for quality and cost. They do not answer the post-training question studied here: once a particular decoder has learned to use a causally sampled native support N, can a frozen model benefit from choosing how much of that support to restore for each prefix?

Changing a segmentation after training. Subword regularization and BPE dropout train models on alternative segmentations so that variation is part of their learning distribution (Kudo, 2018; Provilkov et al., 2020). Other approaches retrofit dynamic tokenization, adapt a domain vocabulary, or search over equivalent retokenizations of the same input string (Feher et al., 2025; Pilana Liyanage & Yvon, 2026; Jain et al., 2026). Prompt compression instead removes material from the external context before the model processes it (Jiang et al., 2023). All of these interventions can change the external token sequence or the text presented to a model. ActiveMesh holds the observed bytes and model weights fixed and changes which *internal* byte states enter the middle network. Restricting actions to N avoids introducing gates at locations the frozen gate did not select. It does not guarantee compatibility: the decoder was trained on its sampled meshes, not on every subset constructed from them. For this reason, the effect of an internal mesh intervention must be measured against continuation likelihood rather than inferred from its token count or its apparent segmentation.

Where to retain computation. There is substantial precedent for selecting positions rather than processing every position equally. Adaptive attention span learns how much context an attention head should use (Sukhbaatar et al., 2019); DynamicViT trains input-dependent token pruning, while Token Merging can combine similar tokens in an existing vision transformer (Rao et al., 2021; Bolya et al., 2023). In language-model inference, H2O balances recently observed tokens with attention heavy hitters, SnapKV uses a prompt observation window to select cached positions, and TokenSelect makes query-dependent KV choices (Zhang et al., 2023; Li et al., 2024; Wu et al., 2025). These methods motivate recency/attention controls, but their principal resource is attention context or KV memory. Our action instead changes the resolution of a frozen byte model's *latent sequence*. Moreover, high attention mass is not a guarantee of high restoration value: the selected state's value vector, interactions in later layers, and the continuation target can all alter its effect on likelihood. This motivates equal-count location controls and separates a plausible selection heuristic from evidence that it improves prediction.

When to spend additional resolution. Adaptive computation time and PonderNet learn example-dependent iteration depth (Graves, 2016; Banino et al., 2021); CALM exits from decoder layers based on confidence, and Mixture-of-Depths allocates layer capacity across positions (Schuster et al., 2022; Raposo et al., 2024). RouteLLM learns when to invoke a stronger rather than a weaker language model (Ong et al., 2025). ActiveMesh considers a different action space: within *one frozen backbone*, it chooses whether a prefix warrants restoring the remainder of its native latent support. This creates a specific counterfactual quantity—the paired continuation-likelihood difference between a fixed partial mesh and native support—and a specific budget question: does a prefix-only estimate of gain per added latent outperform allocating the same realized latent budget by gate count alone? Source code offers plausible observable cues, since identifiers and formatting carry structure that generic subword statistics need not respect (Wang et al., 2021; Li et al., 2026). These cues provide the prefix-level signal used by the router. Added-latent count serves as the architectural cost coordinate, with timing evaluated separately.

8 LIMITATIONS

Our experiments use a constructed reduced mesh and short teacher-forced continuations, so they characterize resolution allocation rather than a complete production serving system. The strongest routing results are on Python source; transfer to JavaScript is smaller and FineWeb shows little signal. The 346M experiments also show that routing scores transfer more readily than budget calibration. Evaluating generated code, cache-aware serving, and larger models are natural next steps.

9 CONCLUSION

Learned tokenization exposes a new inference-time control variable: latent resolution. Our results show that the most reliable way to use this variable is not to micromanage individual boundaries, but to allocate resolution at the prefix level. Boundary placement based on first-layer geometry or attention is dominated by short-range effects, whereas a lightweight prefix router consistently identifies code examples that benefit from native resolution and improves the quality–latent tradeoff on two Python test sets. This granularity separation suggests a practical design path for adaptive tokenizers: learn robust prefix-level compute allocation first, then add finer boundary placement only when a stable positional signal is available.

REPRODUCIBILITY STATEMENT

The appendix gives the model, data, optimization, and evaluation details needed to reproduce the experiments. The anonymous supplement contains the experiment code, data-preparation utilities, per-document outputs, and router weights. Public corpora must be retrieved separately, and the backbone models must be retrained at the stated compute budget.

AI USE STATEMENT

A generative AI assistant was used for language editing. The authors defined the research questions, mathematical derivations, experimental design, implementation, data curation, and scientific figures, and they verified all claims, numerical results, citations, and final text.

REFERENCES

Andrea Banino, Jan Balaguer, and Charles Blundell. PonderNet: Learning to ponder, 2021. URL https://arxiv.org/abs/2107.05407.

Daniel Bolya, Cheng-Yang Fu, Xiaoliang Dai, Peizhao Zhang, Christoph Feichtenhofer, and Judy Hoffman. Token merging: Your ViT but faster. In *International Conference on Learning Representations*, 2023. URL https://openreview.net/pdf?id=JroZRaRw7Eu.

Jonathan H. Clark, Dan Garrette, Iulia Turc, and John Wieting. CANINE: Pre-training an efficient tokenization-free encoder for language representation. *Transactions of the Association for Computational Linguistics*, 10:73–91, 2022. doi: 10.1162/tacl_a_00448. URL https://aclanthology.org/2022.tacl-1.5/.

Sam Dauncey and Roger Wattenhofer. You can learn tokenization end-to-end with reinforcement learning, 2026. URL https://arxiv.org/abs/2602.13940.

Chunyuan Deng, Sanket Lokegaonkar, Colin Lockard, Besnik Fetahu, Nasser Zalmout, and Xian Li. ByteFlow: Language modeling through adaptive byte compression without a tokenizer. In *International Conference on Learning Representations*, 2026. URL https://arxiv.org/abs/2603.03583.

Darius Feher, Ivan Vulić, and Benjamin Minixhofer. Retrofitting large language models with dynamic tokenization. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics*, pp. 29866–29883, 2025. doi: 10.18653/v1/2025.acl-long.1444. URL https://aclanthology.org/2025.acl-long.1444/.

Alex Graves. Adaptive computation time for recurrent neural networks, 2016. URL https://arxiv.org/abs/1603.08983.

Sukjun Hwang, Brandon Wang, and Albert Gu. Dynamic chunking for end-to-end hierarchical sequence modeling, 2025. URL https://arxiv.org/abs/2507.07955.

Kanishk Jain, Matthew Day, and Tankut Can. Emergent retokenization symmetry in large language models: Phenomenology and applications, 2026. URL https://arxiv.org/abs/2606.15521.

Huiqiang Jiang, Qianhui Wu, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. LLMLingua: Compressing prompts for accelerated inference of large language models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 13358–13376, 2023. doi: 10.18653/v1/2023.emnlp-main.825. URL https://aclanthology.org/2023.emnlp-main.825/.

Taku Kudo. Subword regularization: Improving neural network translation models with multiple subword candidates. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics*, 2018. URL https://arxiv.org/abs/1804.10959.

Taku Kudo and John Richardson. SentencePiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pp. 66–71, 2018. doi: 10.18653/v1/D18-2012. URL https://aclanthology.org/D18-2012/.

Yinxi Li, Yuntian Deng, and Pengyu Nie. TokDrift: When LLM speaks in subwords but code speaks in grammar. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics*, pp. 47605–47629, 2026. doi: 10.18653/v1/2026.acl-long.2199. URL https://aclanthology.org/2026.acl-long.2199/.

Yuhong Li, Yingbing Huang, Bowen Yang, Bharat Venkitesh, Acyr Locatelli, Hanchen Ye, Tianle Cai, Patrick Lewis, and Deming Chen. SnapKV: LLM knows what you are looking for before generation. In *Advances in Neural Information Processing Systems*, volume 37, 2024. doi: 10.52202/079017-0722. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/28ab418242603e0f7323e54185d19bde-Abstract-Conference.html.

Piotr Nawrot, Szymon Tworkowski, Michał Tyrolski, Łukasz Kaiser, Yuhuai Wu, Christian Szegedy, and Henryk Michalewski. Hierarchical transformers are more efficient language models, 2022. URL https://arxiv.org/abs/2110.13711.

Isaac Ong, Amjad Almahairi, Vincent Wu, Wei-Lin Chiang, Tianhao Wu, Joseph E. Gonzalez, M. Kadous, and Ion Stoica. RouteLLM: Learning to route LLMs from preference data. In *International Conference on Learning Representations*, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/5503a7c69d48a2f86fc00b3dc09de686-Abstract-Conference.html.

Abraham Toluwase Owodunni, Orevaoghene Ahia, and Sachin Kumar. FLEXITOKENS: Flexible tokenization for evolving language models, 2025. URL https://arxiv.org/abs/2507.12720.

Artidoro Pagnoni, Ramakanth Pasunuru, Pedro Rodriguez, John Nguyen, Benjamin Muller, Margaret Li, Chunting Zhou, Lili Yu, Jason E Weston, Luke Zettlemoyer, et al. Byte latent transformer: Patches scale better than tokens. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 9238–9258, 2025.

Guilherme Penedo, Hynek Kydlíček, Loubna Ben Allal, Anton Lozhkov, Margaret Mitchell, Colin Raffel, Leandro von Werra, and Thomas Wolf. The fineweb datasets: Decanting the web for the finest text data at scale, 2024. URL https://arxiv.org/abs/2406.17557.

Vijini Pilana Liyanage and François Yvon. AdaptBPE: From general purpose to specialized tokenizers. In *Proceedings of the 19th Conference of the European Chapter of the Association for Computational Linguistics*, pp. 2607–2620, 2026. doi: 10.18653/v1/2026.eacl-long.119. URL https://aclanthology.org/2026.eacl-long.119/.

Jatin Prakash, Aahlad Puli, and Rajesh Ranganath. Controllably efficient language models, 2026. URL https://arxiv.org/abs/2511.05313.

Ivan Provilkov, Dmitrii Emelianenko, and Elena Voita. BPE-dropout: Simple and effective subword regularization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pp. 1882–1892, 2020. doi: 10.18653/v1/2020.acl-main.170. URL https://aclanthology.org/2020.acl-main.170/.

Yongming Rao, Wenliang Zhao, Benlin Liu, Jiwen Lu, Jie Zhou, and Cho-Jui Hsieh. DynamicViT: Efficient vision transformers with dynamic token sparsification. In *Advances in Neural Information Processing Systems*, volume 34, 2021. URL https://proceedings.nips.cc/paper_files/paper/2021/hash/747d3443e319a22747fbb873e8b2f9f2-Abstract.html.

David Raposo, Sam Ritter, Blake Richards, Timothy Lillicrap, Peter Conway Humphreys, and Adam Santoro. Mixture-of-depths: Dynamically allocating compute in transformer-based language models, 2024. URL https://arxiv.org/abs/2404.02258.

Craig W. Schmidt, Varshini Reddy, Haoran Zhang, Alec Alameddine, Omri Uzan, Yuval Pinter, and Chris Tanner. Tokenization is more than compression. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 678–702, 2024. doi: 10.18653/v1/2024.emnlp-main.40. URL https://aclanthology.org/2024.emnlp-main.40/.

Tal Schuster, Adam Fisch, Jai Gupta, Mostafa Dehghani, Dara Bahri, Vinh Q. Tran, Yi Tay, and Donald Metzler. Confident adaptive language modeling. In *Advances in Neural Information Processing Systems*, 2022. URL https://arxiv.org/abs/2207.07061.

Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, pp. 1715–1725, 2016. doi: 10.18653/v1/P16-1162. URL https://aclanthology.org/P16-1162/.

Kevin Slagle. SpaceByte: Towards deleting tokenization from large language modeling. *arXiv preprint arXiv:2404.14408*, 2024. URL https://arxiv.org/abs/2404.14408.

Sainbayar Sukhbaatar, Edouard Grave, Piotr Bojanowski, and Armand Joulin. Adaptive attention span in transformers. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pp. 331–335, 2019. doi: 10.18653/v1/P19-1032. URL https://aclanthology.org/P19-1032/.

Yi Tay, Vinh Q. Tran, Sebastian Ruder, Jai Gupta, Hyung Won Chung, Dara Bahri, Zhen Qin, Simon Baumgartner, Cong Yu, and Donald Metzler. Charformer: Fast character transformers via gradient-based subword tokenization. In *International Conference on Learning Representations*, 2022. URL https://arxiv.org/abs/2106.12672.

Junxiong Wang, Tushaar Gangavarapu, Jing Nathan Yan, and Alexander M. Rush. MambaByte: Token-free selective state space model. In *Conference on Language Modeling*, 2024. URL https://arxiv.org/abs/2401.13660.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven C. H. Hoi. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pp. 8696–8708, 2021. doi: 10.18653/v1/2021.emnlp-main.685. URL https://aclanthology.org/2021.emnlp-main.685/.

Wei Wu, Zhuoshi Pan, Kun Fu, Chao Wang, Liyi Chen, Yunchu Bai, Tianfu Wang, Zheng Wang, and Hui Xiong. TokenSelect: Efficient long-context inference and length extrapolation for LLMs via dynamic token-level KV cache selection. In *Proceedings of EMNLP*, pp. 21264–21281, 2025. doi: 10.18653/v1/2025.emnlp-main.1079. URL https://aclanthology.org/2025.emnlp-main.1079/.

Linting Xue, Aditya Barua, Noah Constant, Rami Al-Rfou, Sharan Narang, Mihir Kale, Adam Roberts, and Colin Raffel. ByT5: Towards a token-free future with pre-trained byte-to-byte models. *Transactions of the Association for Computational Linguistics*, 10:291–306, 2022. doi: 10.1162/tacl_a_00461. URL https://aclanthology.org/2022.tacl-1.17/.

Lili Yu, Dániel Simig, Colin Flaherty, Armen Aghajanyan, Luke Zettlemoyer, and Mike Lewis. MEGABYTE: Predicting million-byte sequences with multiscale transformers, 2023. URL https://arxiv.org/abs/2305.07185.

Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, Zhangyang Wang, and Beidi Chen. H2O: Heavy-hitter oracle for efficient generative inference of large language models. In *Advances in Neural Information Processing Systems*, volume 36, 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/6ceefa7b15572587b78ecfcebb2827f8-Abstract.html.

A DATA, TRAINING, AND EVALUATION DETAILS

Each retained source document contributes its first 4,160 UTF-8 bytes as one fixed window. The first 4,096 bytes are the observed prefix and the final 64 are used only for continuation scoring. Foundation training uses the first 4,096 bytes. Documents shorter than 4,160 bytes, exact full-document and first-1,024-byte duplicates, and documents or repositories appearing in the archived predecessor experiments were excluded before split assignment. Semantic duplicates may remain. FineWeb is partitioned by document, while code is partitioned by repository so that a repository does not cross controller splits. The partition sizes are shown in Table 3. We use fixed public FineWeb and CodeParrot-clean source shards. Each retained code file contributes one fixed window; this is therefore a controlled source-code sample rather than a representative sample of all GitHub code.

Table 3: Window counts. Parentheses give distinct repository counts for the CodeParrot-clean Python shard. Test rows are disjoint from model and policy selection data.

Split	FineWeb	Python code
Foundation train	262,209	—
Foundation validation	1,024	—
Controller train	14,707	32,461 (19,042)
Controller validation	2,048	2,048 (1,407)
Test	4,096	4,096 (2,440)

Both 147M seeds use 4 down, 12 middle, and 4 up layers; the 73M model uses 2, 6, and 2. All have embedding width 768, 12 heads, a 64-byte sliding window in the outer layers, and an expected native downsampling rate of 0.2. All models use an effective batch of 128 windows, AdamW, gradient clipping at 1, BF16 autocast with FP32 master weights, and a byte-budgeted warmup followed by cosine decay. The 147M models use peak learning rate 1.5×10^{-3} over 6.0B byte exposures; the 73M model uses 2.4×10^{-3} over 3.4B. Training windows are sampled with replacement from the same fixed FineWeb partition. The implementation follows the published late plus 0.1 early loss rather than the released wrapper's 0.9 late plus 0.1 early weighting, excludes forced endpoints from gate and rate losses, and keeps external byte-coordinate rotary positions. These implementation differences are included in the released configuration. The two foundation-validation-selected 147M checkpoints occur at updates 11,000 and 11,200 (validation BPB 1.22888 and 1.23093). The training objective weights late, early, score-function, and gate-rate terms by 1, 0.1, 0.01, and 0.01, respectively. FineWeb partitions are document-level and code partitions are repository-level. Exact full-document and first-1,024-byte duplicates are removed before splitting. The experiment environment uses PyTorch `2.14.0.dev20260623+cu132`, Transformers `4.51.3`, FlashAttention `2.8.3.post1`, NumPy `2.5.0`, and scikit-learn `1.9.0` on RTX PRO 6000 Blackwell GPUs. Observed single-GPU wall time for foundation training was 11.56, 10.70, and 3.76 hours for 147M seed 101, 147M seed 202, and 73M seed 101, respectively; policy development and diagnostics are additional compute.

B ADDITIONAL DEVELOPMENT RESULTS

These tables collect the development-set results summarized in the main text; the corresponding test results appear in Section 5.

Variable-rate continuation diagnostic. On 256 FineWeb controller-validation documents with 2,048-byte full-sequence scoring, one 147M seed continued for 8,192 matched updates reaches 1.24442 BPB at expected gate rate 0.5, versus 1.29029 for a native-only decoder continued for the same updates. The untouched backbone at its native rate scores 1.23293; the best variable-resolution checkpoint reaches 1.24405 at rate 0.5. These full-sequence numbers are separate from the prefix-only next-64-byte densification test.

Feature and witness diagnostics. On code development at a 25% calibration target, cheap plus repetition features gain 0.00717 BPB over the strong offline count allocation (repository interval $[0.00421, 0.01020]$), close to cheap plus all lexical features at 0.00713; cheap plus syntax yields

0.00282, and cheap alone 0.00150 with an interval crossing zero. All feature groups have weak or negative no-fit FineWeb contrasts. A distinct equal-count within-support rule that places 32 gates near earlier repeated identifier/suffix witnesses has 73.63% witness coverage but improves only 0.00087 BPB over fixed 24+8 (repository interval $[-0.00262, 0.00419]$) on code development. The witness rule therefore does not show a reliable location advantage.

Additional action-choice baselines. On 1,024 FineWeb development documents per backbone, a 26-feature STOP controller gains 0.03087 BPB over the coarse mesh at an equal 32-boundary budget versus 0.03028 for always choosing geometry; their paired advantage interval includes zero. At an approximately 16-boundary average budget it gains 0.01786 versus 0.01519 for random STOP on geometry, but at an exact per-backbone 50% action budget it gains 0.01702 versus 0.01652 for geometry-only learned STOP, again with an interval crossing zero. The code equal-count full-feature router's internal holdout gain over fixed 24+8 was 0.00122 BPB, interval $[0.00028, 0.00223]$, but fell to 0.00042 on separate code controller-validation documents, interval $[-0.00080, 0.00167]$. An additional lexical equal-count router also does not show a reliable advantage over fixed 24+8. A no-fit observed-suffix validation rule lost 0.00090 BPB to the pilot-selected fixed action on the 147M code development set. A cheap native-escalation predictor beats matched random allocation but not the native-count heuristic. The lexical budget route adds code-structure features and is evaluated on a separate reserve.

Table 4: Mean next-64-byte BPB gain from densifying the observed prefix relative to the native sampled mesh. Negative means worse. Each cell is a separate 512-document model-selection evaluation in BF16.

Corpus	Prefix bytes	147M seed 101	147M seed 202
FineWeb	1024	-0.00718	-0.00780
FineWeb	4096	-0.00811	-0.00994
Python code	1024	-0.01410	-0.01544
Python code	4096	-0.01612	-0.01730

Table 5: CodeParrot-clean Python $C = 4096$, documents 256:2048, next-64-byte BPB gain relative to the half-native starting mesh. All 32-boundary actions have the same final latent count. These data are independent of the first-256 rule selection, but remain development-set analysis.

Action	147M seed 101	147M seed 202
Recent 32	0.03916	0.03230
Attention 32	0.03667	0.02891
Recent 24 + attention 8	**0.04206**	**0.03525**
Full native	0.08742	0.08222

Exploratory multiquery check. After the location test result, we tested whether summing attention mass over the last 1, 4, 16, or 64 retained queries repairs the horizon decay. The first 256 code controller-validation documents select one query using the H64 two-seed mean, reproducing the 24+8 rule. On the remaining 1,792 code documents, one-query minus four-query H64 gain is $+0.00046$ BPB, with a repository interval $[-0.00056, 0.00147]$; the 64-query variant trails recent-32 by 0.00231 BPB. The FineWeb first-512 H64 one-query versus four-query difference is -0.00005 BPB with an interval spanning zero. Aggregating more queries therefore does not improve the H64 location result in this diagnostic.

C RESTORATION EVALUATION DETAILS

A deterministic per-document seed keeps native gate sampling paired across mesh actions. The half-native starting mesh is formed by rounding equally spaced indices in the sorted native support and includes BOS and the final native gate before the observed endpoint. Recent-32 restores the latest omitted native gates; the proposed rule restores the latest 24 and then up to eight positions with

the largest first-middle-layer normalized attention mass. Geometry, pure attention, gate-probability, reduced, and full-native variants are evaluated on the same sampled support and continuation.

Quality uses FP32 eager attention with TF32 disabled. Code confidence intervals use 20,000 repository-bootstrap replicates, preserving all documents of a sampled repository and pairing document indices across backbone seeds. FineWeb intervals resample documents. The primary location comparison is the two-seed mean Python-code next-64-byte BPB gain of 24+8 over recent-32 at $C = 4096$; the other horizons, seed-specific results, FineWeb transfer, and 73M comparisons provide supporting analyses.

D First-layer counterfactual identity

For one attention head, the first middle-layer query q_r at the final native gate and the key and value projections (k_i, v_i) of every native candidate are computed from the same causal encoder states under every subset action. The positional rotation, score scaling, and score soft cap are included in s_i. With no pre-existing KV cache, a subset T changes only which (k_i, v_i) pairs enter this query's softmax. Hence $o_T = \sum_{i \in T} e_i v_i / Z_T$, as stated in the main text. For a missing candidate j,

$$o_{S \cup \{j\}} - o_S = \frac{Z_S o_S + e_j v_j}{Z_S + e_j} - o_S = \frac{e_j}{Z_S + e_j}(v_j - o_S), \tag{7}$$

$$\|R\|^2 - \|R - d_j\|^2 = 2\langle R, d_j \rangle - \|d_j\|^2. \tag{8}$$

Summing this last identity over heads yields the singleton geometry score. It is exact for that one query and one layer under the stated causal selection assumptions. Later middle layers, upsampling, and output logits are nonlinear functions of the whole selected mesh, so the identity does not bound or optimize continuation BPB. The code-domain reversal of geometric ranking illustrates that distinction empirically.

E Location results across horizons and model sizes

Small-backbone seed heterogeneity. On 2,048 repository-disjoint code controller-validation documents, recent-32 minus native BPB is $+0.01048$ for 73M seed 101 and -0.01744 for seed 202 (the latter's repository interval is $[-0.01946, -0.01546]$). Their paired difference is 0.02793 BPB, interval $[0.02531, 0.03055]$. Selected FineWeb foundation-validation BPBs are 1.42499 and 1.42089, with mean code native support sizes 1,459 and 1,484. A third independent 73M seed gives -0.14961 BPB on the same code documents (repository interval $[-0.15816, -0.14139]$) and -0.03270 on FineWeb development; its foundation-validation BPB is 1.38460 and mean code support size is 1,519. All three received 3.400B training-byte exposures. The seed-101 FineWeb test favors native by 0.00791 BPB. Together, these observations show substantial rate heterogeneity at 73M.

Table 6 gives the absolute next-64-byte losses behind Table 1. Each code row contains the same 4,096 test documents in the same order; code intervals preserve repository clusters and document pairing across seeds. Full native is the best-quality 147M action in both domains. On 73M seed 101, code recency wins over native, but the 24+8 location rule loses to recency; the FineWeb rate effect has the opposite sign.

Table 6: Mean next-64-byte BPB on the location test; lower is better. The 147M hybrid gain over recency in code is positive but its 95% interval includes zero.

Checkpoint / corpus	Recent-32	24+8	Native
147M seed 101 / code	2.19709	2.19622	2.14962
147M seed 202 / code	2.26458	2.26285	2.21401
147M seed 101 / FineWeb	1.22663	1.22618	1.21338
147M seed 202 / FineWeb	1.22862	1.22811	1.21586
73M seed 101 / code	2.98722	2.99827	2.99797
73M seed 101 / FineWeb	1.41998	1.41982	1.41207

Table 7 reports the 147M horizon readouts. These are correlated measurements on the same continuations and are used to characterize the time scale of the location effect. The disjoint-block calculation

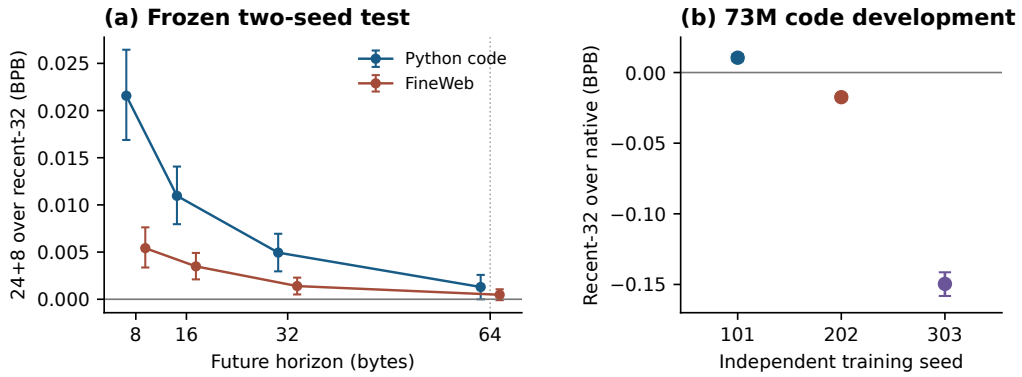

Figure 5: Two sources of instability in location-based refinement. Left: the 147M horizon contrasts for 24+8 versus recent-32, whose H64 interval crosses zero. Right: independent 73M instances disagree on recent-32 versus native on the same development code documents. Bars are 95% repository-bootstrap intervals for code and document-bootstrap intervals for FineWeb.

is a post-hoc linear transformation of the same per-document cumulative losses: with D_H the mean gain over the first H bytes, the mean gain on bytes $a+1{:}b$ is $(bD_b - aD_a)/(b-a)$. On code, the paired block gains for bytes 1–8, 9–16, 17–32, and 33–64 are respectively $+0.02157$, $+0.00036$, -0.00108, and -0.00234 BPB. The last block's repository interval is $[-0.00385, -0.00088]$; this block interval is a post-hoc diagnostic of the observed decay.

Table 7: Two-seed 24+8 minus recent-32 mean BPB gain by cumulative continuation horizon. Intervals are repository-bootstrap for code and document-bootstrap for FineWeb.

Future bytes	Python code: gain [95% interval]	FineWeb: gain [95% interval]
8	$+0.02157$ $[0.01688, 0.02645]$	$+0.00542$ $[0.00337, 0.00762]$
16	$+0.01096$ $[0.00796, 0.01406]$	$+0.00349$ $[0.00211, 0.00491]$
32	$+0.00494$ $[0.00296, 0.00694]$	$+0.00140$ $[0.00051, 0.00229]$
64	$+0.00130$ $[-0.00001, 0.00258]$	$+0.00048$ $[-0.00010, 0.00106]$

As a post-hoc check of document weighting within code repositories, we give every repository equal weight instead of every document. The H64 paired mean becomes $+0.00064$ BPB with a repository-bootstrap interval $[-0.00087, 0.00212]$ across 2,440 repositories. The equal-repository estimate is similarly close to zero.

F Budgeted routing: implementation and additional results

The routing model uses paired labels from 8,192 code controller-training documents per 147M seed, grouped by repository: 11,504 two-seed rows for fitting, 1,540 for model choice and calibration, and 3,340 for an internal holdout. The selected model is a standardized ridge regressor with penalty 100 on 35 features. The fixed per-seed 25% thresholds for predicted Δ/c are 8.75570×10^{-5} and 8.72259×10^{-5}; no external scores enter them. The regression target and threshold correspond to the population expected-budget rule derived above, not an exact finite-batch knapsack. The direct development-set lexical-versus-ten-feature ablation gains 0.00638 BPB at 25% and 0.00603 at 50%, with descriptive repository intervals $[0.00323, 0.00965]$ and $[0.00318, 0.00897]$ at virtually identical latent spend. These are post-selection development-set comparisons.

The routing test set is a separate Python shard of CodeParrot-clean. Its 4,096 retained documents come from 3,942 repositories and contribute one first-4,160-byte window per file. Repository-, file-, and prefix-level exact deduplication separates it from the development index. The same evaluator is used for development and test data.

Table 8: External mean next-64-byte BPB and actual incremental latent spend. Lower BPB is better. Count-only is allowed to sort the whole test workload by gate count at the route's actual per-seed spend; the route uses a fixed calibration threshold for each request.

Target	Seed	24+8	Count-only	Route	Native	Extra latents/doc
25%	101	2.20726	2.19461	2.18874	2.15868	168.65
25%	202	2.29086	2.27768	2.27248	2.23936	164.02
50%	101	2.20726	2.18226	2.17623	2.15868	348.23
50%	202	2.29086	2.26316	2.25901	2.23936	341.73

The route's realized extra-latent fractions were 22.59%/22.59% at the 25% target and 46.64%/47.07% at the 50% target for seeds 101/202. At 25%, 23.34%/23.44% of requests chose native; at 50%, 48.49%/49.07% did. The repository bootstrap resamples all files within a sampled repository and pairs the two model seeds on each file. As a post-hoc sensitivity analysis, equal weighting of the 3,942 repositories gives $+0.00548$ BPB, close to the file-weighted $+0.00553$; the largest repository has only three files. Leaving out any one repository yields gains from $+0.00535$ to $+0.00565$ BPB. In 2,000 paired repository resamples that *re-rank* the offline count comparator and rematch its budget each time, the descriptive 95% interval is $[0.00400, 0.00719]$ BPB. Only 38.0% of repositories have a nonzero paired policy contrast, and 54.6% of those favor the route. These sensitivity checks leave the point estimate essentially unchanged.

In a FineWeb development transfer without refitting, the two code-calibrated 25% thresholds spend 16.45% and 16.73% of available extra latents on 1,024 documents per seed. The route improves over fixed 24+8 by 0.00231 BPB, but its paired gain over offline count at actual spend is -0.00079 BPB, with a 20,000-draw document-bootstrap interval $[-0.00296, 0.00112]$; the individual seed contrasts are -0.00052 and -0.00105. At the secondary 50% target, actual spend is 43.43%/43.85% and the route-versus-count contrast is -0.00007 BPB, interval $[-0.00198, 0.00174]$. These are cross-domain development diagnostics; the code-shard confirmation is unchanged.

The independently trained 147M seed 303 supplies a backbone-transfer check without refitting. Its 6.000B-exposure foundation run selects the update-11,000 checkpoint by FineWeb validation BPB 1.21579. Applying the fixed predictor with the arithmetic mean of the seed-101/202 thresholds to 1,792 code development documents gives a 25%-target route-over-offline-count gain of $+0.00667$ BPB at 23.41% actual incremental spend, repository interval $[0.00191, 0.01153]$. At the 50% target, the contrast is $+0.00050$, interval $[-0.00421, 0.00520]$. Its equal-count 24+8-versus-recent-32 gain is $+0.00124$, interval $[-0.00193, 0.00444]$. The same fixed predictor on the *reused* 4,096-file external shard gains $+0.00789$ BPB over matched offline count at 22.48% spend for the 25% target, interval $[0.00492, 0.01092]$; the secondary 50% target gains $+0.00833$ at 46.48% spend, interval $[0.00521, 0.01153]$. We report seed 303 as a separate backbone-transfer analysis rather than including it in the two-seed aggregate.

Additional code and language-transfer sets. The third Python shard contains 3,944 repositories with no exact overlap with the earlier code partitions, and we evaluate the same two-seed controller without refitting. At the 25% calibration point the realized extra-latent fractions are 22.52%/22.48% for seeds 101/202; their route-minus-offline-count BPB gains are $+0.00686$/$+0.00747$. The route still trails native by 0.02904 BPB in the paired mean. On the same teacher-forced target, top-1 correctness rises by 0.000973 (repository interval $[0.000525, 0.001419]$), while first-byte correctness changes by 0.000122 (interval $[-0.001829, 0.002069]$). A fixed structural-byte subset of 44,108 target bytes gains 0.002267 accuracy (interval $[0.000820, 0.003729]$). Generation-level evaluation remains for future work.

The no-fit JavaScript transfer uses a fixed Python-trained router on 1,024 files from 986 repositories. The realized extra-latent fractions are 22.09%/22.71% and the per-seed BPB gains over offline count are $+0.00493$/$+0.00268$. Sensitivity analyses give a $+0.00352$ equal-repository gain; 1%-trimmed and winsorized file means are $+0.00248$ and $+0.00330$. Re-ranking and budget-rematching the count baseline inside 2,000 paired repository resamples gives a descriptive interval $[0.00081, 0.00728]$. The effect is small enough that a broader language-general conclusion would require larger transfer studies.

Second language-transfer tests. The JavaScript-v2 test set contains 1,024 files from 1,007 repositories and uses a repeat-only Python-trained regressor. Its 25% calibration target spends 22.26%/23.47% of available extra latents for seeds 101/202 and improves paired H64 BPB over offline count by $+0.00363$ (20,000-draw repository interval $[0.00007, 0.00730]$). The earlier full-feature regressor on the same shard gives $+0.00405$ BPB against its *own* cost-matched count comparator; the two routes have different realized spends and cannot be compared directly. The repeat-only 50% point gives $+0.00371$ BPB, interval $[0.00022, 0.00732]$. The FineWeb-v2 test set has 1,024 documents. At 15.91%/15.52% actual spend, the unchanged full-feature code router gives -0.00001 BPB over offline count, paired document interval $[-0.00153, 0.00151]$. Its secondary 50% point gives -0.00098, interval $[-0.00280, 0.00086]$. The fixed thresholds underspend on this domain, but all comparisons rematch actual spend. The small JavaScript effects and near-zero FineWeb results indicate limited transfer beyond the source-code setting.

G Exploratory transfer to 346M models

Three independently initialized 346M Byte U-Nets were trained on the same FineWeb partition with a matched 6B-byte sampling budget. FineWeb validation selects checkpoints at 5.66B, 5.98B, and 5.87B byte exposures. We apply the 147M predictor and the mean of its two calibration thresholds without refitting to the previously defined code development documents 256–2047 (1,268 repositories). For all fixed-size restoration rules we use $K' = \min(32, |N \setminus S|)$, so the action naturally restores all remaining native positions when fewer than 32 are available.

Table 9: Exploratory 346M no-fit transfer, three paired backbones on 1,792 code development files. The comparator is re-ranked by gate count at each run's *actual* extra-latent spend. Intervals resample repositories and condition on these three checkpoints. The 25% and 50% labels refer to the original 147M calibration settings; actual 346M spend is shown explicitly.

Target	Actual spend range	Gain BPB [95% interval]	Positive runs
25%	22.8–52.1%	$+0.00696$ $[0.00486, 0.00909]$	3/3
50%	47.0–86.1%	$+0.00192$ $[0.00049, 0.00345]$	2/3

At the 25% target, individual paired gains over count are $+0.01671$, $+0.00054$, and $+0.00362$ BPB; at 50% they are $+0.00584$, -0.00194, and $+0.00187$. Full native still improves over the route by 0.01702 BPB at the lower target. The notable scaling behavior is the large spread in realized budget, showing that the router's ranking signal transfers more cleanly than its calibration threshold.

H Implementation details and timing

The prefix-only and full-sequence implementations agree on the evaluated endpoint probabilities, and the vectorized mesh planner matches the reference selection on the checked examples. Prompt-prefill timing covers the encoder, planner, and decoder for the observed prefix. Code timing was measured on physical GPUs 0 and 1 for the 147M backbone and GPU 3 for the 73M model. For the later 25%-target route, warmed FP32 single-request timing uses 16 warmup and 96 measured development prompts per seed. Ordinary mixed batch-8 FP32 timing uses two warmup and twelve measured batches per seed, with the same prefix encoder and a batch decoder containing both native and reduced supports. For seeds 101/202 the mixed route reaches 9.43/9.51 documents/s, fixed 24+8 reaches 11.11/11.19, and full native reaches 8.95/9.09. The route averages 987/963 latent states per document, versus 1,550/1,511 for native, but its mean *maximum* support per mixed batch is 1,721/1,696. This padding effect explains why fewer average states do not translate proportionally into batch throughput. A layout experiment on the same 96-document workload per seed shows that splitting each batch into native and 24+8 sub-batches reduces mean batch time from 849/840 to 756/748 ms for seeds 101/202. Throughput rises from 9.42/9.53 to 10.58/10.70 documents/s; all twelve measured batches per seed favor grouping. The reduced-support subgroup's mean maximum latent count is 982/955, versus 1,721/1,696 for the native subgroup. This confirms that support-size heterogeneity, rather than average latent count alone, is a major source of padding overhead. Using the same single-request component times, an offline count-only allocation across the 96 requests that spends no more incremental latent states than the route reaches 10.71/10.80 requests/s for seeds

101/202, versus 10.38/10.46 for the lexical route. Its binary allocation leaves 241/436 extra latent states unspent on the two 96-request samples. At the same latent budget, this count-based allocation is slightly faster in the current implementation. Together with the split-batch result above, this shows that realized throughput depends strongly on support-size variance and batching strategy, not only on mean latent count.