PONDERLM-3: TOKEN-ADAPTIVE LOOPING WITH DIFFERENTIABLE MASKING

Anonymous authors
Paper under double-blind review

ABSTRACT

Looped language models offer an effective way to improve model performance through iterative computation, motivating a natural follow-up question: when are additional loop iterations actually needed? To address this question, we introduce *PonderLM-3*, a pretraining framework for token-adaptive pondering, where easy tokens stop pondering early while difficult tokens receive additional pondering steps. This allows inference computation to be dynamically allocated at the per-token level. Under purely self-supervised training, PonderLM-3 learns how many pondering steps to allocate to each token through a differentiable attention masking mechanism, which also maintains train–inference consistency. At inference, the learned allocation is realized through hard pruning. The framework builds on the PonderLM-2 backbone. Compared with existing loop-based baselines, PonderLM-3 defines a stronger Pareto frontier, achieving lower pretraining perplexity at equal inference FLOPs. It also achieves the highest average performance on downstream benchmarks.

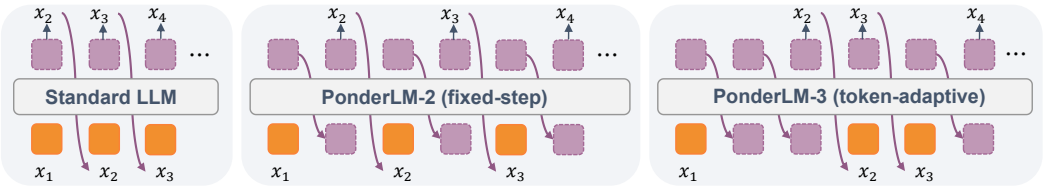


Figure 1: From fixed compute to token-adaptive allocation. Standard LMs decode with one forward pass per token. PonderLM-2 applies fixed pondering steps, turning extra compute into a uniform per-token overhead. PonderLM-3 makes refinement depth token-dependent, allocating additional internal updates only when they provide non-trivial marginal gains.

1 INTRODUCTION

Looped language models have emerged as an effective way to improve language modeling and reasoning by repeatedly applying shared parameters to refine latent representations (Giannou et al., 2023; Yang et al., 2024; Saunshi et al., 2025; Zeng et al., 2025b;a; Zhu et al., 2025). Recent loop-based models learn such iterative computation directly through self-supervised pretraining on large-scale unlabeled text, without human-annotated supervision. In this paper, we refer to the loop computation in the PonderLM family as *pondering*.

However, existing loop-based language models typically use a fixed number of computation steps per token, so inference cost grows linearly with the step budget. This fixed-step regime does not match the uneven computational needs of language generation: many tokens require only simple local continuation or copying, while a smaller fraction can strongly affect subsequent generation. As a result, fixed-step computation not only wastes compute on easy tokens and can even cause overthinking (Fu et al., 2025a) that degrades predictions, but also leaves hard tokens with insufficient computation. We therefore seek a model that decides how much additional computation each token needs and stops pondering when further computation is no longer useful.

This motivates adaptive computation, where the model decides how far to refine and when to stop during generation, making Adaptive Computation Time (ACT)–style dynamic halting a natural

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.

framework. Yet existing ACT paradigms commonly exhibit four limitations: (i) limited adaptivity, where compute-saving policies such as pruning ratios, token budgets, or halting priors are fixed by design rather than learned from data-driven token difficulty (Raposo et al., 2024; Bae et al., 2025; Chen et al., 2025; Hwang et al., 2025; Banino et al., 2021); (ii) halting policies learned or calibrated outside pretraining, where the allocation rule is obtained through SFT with step annotations (Fu et al., 2025a), RL on a frozen backbone (He & Tang, 2025), extra supervised targets (Fu et al., 2025a; Jolicoeur-Martineau, 2025), multi-stage training (Fu et al., 2025a; He & Tang, 2025), or post-hoc thresholds (Fu et al., 2025b; Schuster et al., 2022; Scalena et al., 2025); (iii) structural overhead during acceleration, such as draft-and-verify, early-exit rollback, or piecewise static policy compositions, whose extra costs can trigger more frequently on hard tokens, sometimes wasting compute where deeper processing is most needed (Leviathan et al., 2023; Elhoushi et al., 2024); and (iv) train–inference mismatch, as halting is learned under parallel training but executed sequentially during generation.

To overcome these challenges, we propose **PonderLM-3**, a data-driven, token-adaptive mechanism for pondering-style pretraining models, trained end-to-end under self-supervised language modeling. Building on the PonderLM-2 (Zeng et al., 2025a) backbone that supports efficient parallel training via Jacobi iterations, we retain training efficiency and train–inference consistency while learning per-token pondering-step allocation. To make inference-time hard stopping learnable under self-supervised pretraining, we let a lightweight router produce a *differentiable attention mask* that modulates the contributions of later pondering steps during training, providing a continuous approximation of step skipping optimized directly from the next-token objective. At inference, this learned gate is instantiated as a hard-stopping criterion, allowing the model to execute only the necessary pondering steps. This approach achieves tangible reductions in inference latency while maintaining generation fidelity, offering a more granular and controllable compute-quality trade-off.

2 Related Work

2.1 Looped Language Models

Looped language models enable multiple rounds of internal computation by repeatedly applying shared parameters, rather than relying on a single forward pass for each prediction. Existing approaches can be broadly categorized into *vertical* and *horizontal loops*. Vertical loops repeat computation along model depth, whereas horizontal loops perform additional iterations by inserting or propagating latent states along the sequence.

Vertical loops in Transformers dates back to the Universal Transformer (Dehghani et al., 2018), with subsequent work exploring looped computation for iterative algorithms and latent reasoning (Giannou et al., 2023; Saunshi et al., 2025; Zeng et al., 2025b). More recently, Huginn (Geiping et al., 2025) and Ouro (Zhu et al., 2025) have scaled this direction to substantially larger pretraining regimes,while other studies investigate training stability under deeper loops (Fu et al., 2026; Li et al., 2026).

Horizontal loops instead propagate latent states across sequential positions or computation steps. Coconut (Hao et al., 2024), PonderLM-2 (Zeng et al., 2025a), and PCCoT (Wu et al., 2025) feed continuous latent states into subsequent computation, with PonderLM-2 and PCCoT using Jacobi iterations to parallelize the otherwise sequential dependency. More recently, T2MLR (Cai et al., 2026) feeds a deeper representation from the previous token into an earlier layer of the current token.

2.2 Adaptive Computation Time

Adaptive Computation Time (ACT) (Graves, 2016) introduced dynamic halting for recurrent computation, allowing the number of internal updates to vary across inputs. Existing adaptive methods differ mainly in how computation allocation is controlled or learned. Fixed-budget and prior-based methods constrain token-wise computation through predefined continuation ratios or structural priors: MoR (Bae et al., 2025), MoD (Raposo et al., 2024), and ITT (Chen et al., 2025) retain a prescribed fraction of tokens, H-Net (Hwang et al., 2025) enforces a target compression ratio through a specialized ratio-style loss, and PonderNet (Banino et al., 2021) regularizes halting toward a geometric prior. Another line learns stopping policies through additional supervision, reinforcement

Table 1: Comparison of adaptive-compute methods.

Method	Training Paradigm	Adaptive Mechanism	Constraints / Priors
PonderNet	Pretraining	Learned stopping distribution	Geometric prior
MoR / MoD / ITT	Pretraining	Router-based token selection	Fixed continuation ratio
Think-at-Hard	SFT (LoRA)	Supervised step targets	External step labels
Learning to Ponder	RL (GRPO)	Learned controller	Frozen backbone; reward-shaped budget
CALM / DAT	Post-hoc	Confidence-based stopping	Threshold calibration
LayerSkip	Pretraining	Draft–verify	Static exit policy
AdaPonderLM	Pretraining	Iteration-specific gating with token-wise K/V replacement	No fixed pruning ratio
PonderLM-3 (ours)	Pretraining	Router-predicted differentiable attention mask	No fixed continuation ratio or external step labels

learning, or multi-stage training (Fu et al., 2025a; He & Tang, 2025; Jolicoeur-Martineau, 2025), while training-free approaches determine stopping thresholds through post-hoc calibration (Fu et al., 2025b; Schuster et al., 2022; Scalena et al., 2025). Draft-and-verify methods accelerate generation through cheaper proposal computation followed by verification, as in speculative decoding and LayerSkip (Leviathan et al., 2023; Elhoushi et al., 2024). AdaPonderLM (Song et al., 2026) additionally learns iteration-specific gates for token-level early exiting and reuses previous-iteration K/V states after halting. Table 1 summarizes these methods along their training paradigms, adaptivity sources, and constraints.

3 METHOD

3.1 BACKGROUND: JACOBI-ALIGNED LATENT-THOUGHT BACKBONE

We build on the pretraining framework of PonderLM-2 (Zeng et al., 2025a), which inserts K internal latent pondering steps between adjacent observed tokens, forming a sequence of hidden states $\{h_t^{(k)}\}_{k=0}^{K-1}$ at each position. At inference time, the model repeatedly feeds the current position's last-layer hidden state back as the input embedding for subsequent pondering steps, yielding a sequence of additional computation steps at the same token position. At training time, PonderLM-2 adopts a Jacobi-aligned approximation that rewrites the recurrent updates as parallel iterations: in each iteration, a single Transformer forward pass updates all latent positions simultaneously, and a small number of iterations is performed to approach the fixed point that matches inference-time behavior.

3.2 TOKEN-ADAPTIVE PONDERING MODULES

PonderLM-3 learns a token-wise pondering budget through four components.

(1) Step distribution. For each token position t, a lightweight router takes the step-0 hidden state $h_t^{(0)}$, i.e., the representation before additional pondering, and predicts a distribution over the number of pondering steps:

$$s_{t,k} = \Pr(\text{token } t \text{ uses } k \text{ pondering steps}), \qquad k = 0, \ldots, K.$$

(2) Mask score. We convert the step distribution into a continuation score using its tail probability:

$$w_{t,k} = \sum_{j=k}^{K} s_{t,j},$$

where $w_{t,k}$ represents the probability of reaching step k.

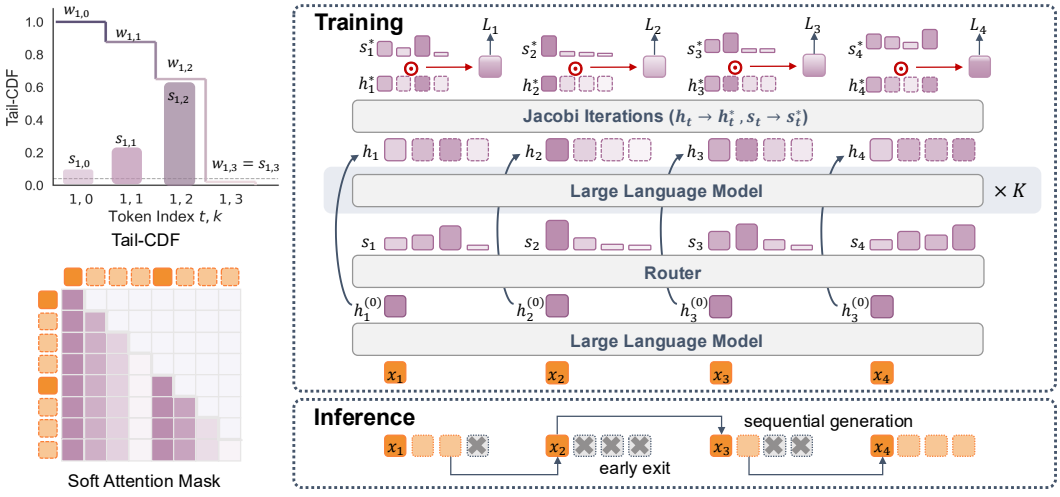


Figure 2: Token-adaptive pondering in PonderLM-3. **(1)** A lightweight router predicts the step distribution $s_{t,k}$ from $h_t^{(0)}$, and obtains a monotone mask score $w_{t,k}$ via the tail-CDF of s; $w_{t,k} \in [0,1]$ serves as a soft attention mask (color intensity indicates w). **(2)** the same LLM performs K pondering steps; at each step, the current hidden state is inserted as a latent token at its corresponding position to form an interleaved sequence, and $w_{t,k}$ is applied in attention as a soft mask. Then we use Jacobi iterations to approximate the sequential inference dynamics, and the output uses the $s_{t,k}$-weighted fused hidden state. **(3)** Inference: token-by-token decoding uses the same attention masking and fusion mechanism, and early exits once $w_{t,k} < \tau$, skipping remaining pondering steps.

(3) Differentiable attention mask. During training, w controls the contribution of later-step latent states through a differentiable attention mask. Specifically, we add $\log w$ to the attention logits:

$$\mathrm{Attn}(Q,K,V) = \mathrm{Softmax}\left(\frac{QK^\top}{\sqrt{d}} + M + \log w\right) V,$$

where M is the standard causal and padding mask. As $w_{t,k}$ approaches zero, the corresponding latent state receives vanishing attention. Our FlashAttention-2-compatible implementation is described in Appendix A.

(4) Hidden-state integration. Finally, we integrate representations from different pondering steps using the predicted step distribution:

$$\hat{h}_t = \sum_{k=0}^{K} s_{t,k} h_t^{(k)},$$

and use $\hat{h}_t$ for next-token prediction. Together, the soft attention mask and hidden-state integration provide a differentiable training procedure, while the same continuation score enables hard token-wise stopping at inference.

3.3 Inference

At inference time, PonderLM-3 performs standard autoregressive decoding. For each token, the model first computes the step-0 hidden state and obtains the mask scores for all pondering steps, $\{w_{t,k}\}_{k=0}^{K}$. Pondering steps are then executed sequentially, with token-wise hard stopping once the mask score falls below a fixed truncation threshold τ[1].The stopping step $\hat{K}_t$ is defined as the last step whose mask score remains above τ, and all subsequent pondering steps are skipped.

[1]We use a fixed truncation $\tau = 10^{-4}$. Under bfloat16 precision, contributions below this magnitude are numerically negligible, so truncating them is effectively equivalent to hard stopping. τ is not tuned as a halting prior and it is not used as a compute budget hyperparameter.

For the executed steps, we retain the same differentiable attention mask to softly down-weight later-step latent states, and use the same weighted hidden-state integration truncated at $\hat{K}_t$. In this way, hard stopping removes computation whose contribution is numerically negligible, allowing different tokens to execute different numbers of pondering steps and directly reducing inference FLOPs.

3.4 TRAINING

PonderLM-3 is trained end-to-end with next-token prediction. We use parallel Jacobi iterations to align training with sequential pondering at inference, together with a minimum-ponder penalty that encourages earlier stopping.

3.4.1 PARALLEL TRAINING VIA JACOBI ITERATIONS

To align parallel training with sequential inference, we construct an interleaved latent sequence and apply the Transformer iteratively in Jacobi iterations. Let $e(x_t)$ denote the token embedding of the observed token x_t, and let $h_t^{(k)}$ be the step-k latent state associated with token position t. We form an interleaved sequence (for a given iteration r) as

$$S^{(r)} \;=\; \big[\, e(x_1),\, h_1^{(0)},\, \ldots,\, h_1^{(K-1)},\, e(x_2),\, h_2^{(0)},\, \ldots,\, h_2^{(K-1)},\, \ldots \,\big].$$

We then apply the same LLM with shared parameters to update all latent states in parallel:

$$S^{(r+1)} = \mathrm{Transformer}\Big(S^{(r)}\Big),$$

which produces an updated set of latent states $\{h_t^{(k)}\}$ for all token positions and steps simultaneously.

Crucially, the router is re-applied after each iteration to the updated step-0 states, yielding updated step distributions s_t (and thus updated mask scores). As the iterations proceed, both the latent trajectory and the router outputs are repeatedly refined,

$$h_t \to h_t^\star, \qquad s_t \to s_t^\star,$$

and after a small number of iterations the process approaches a fixed point. This fixed-point approximation matches the behavior of running pondering steps sequentially at inference, while remaining efficiently parallelizable during training. Appendix B explains the convergence of the Jacobi trajectory and provides empirical evidence that only a few iterations are needed to approach the sequential-inference solution.

3.4.2 AUXILIARY LOSS: MINIMUM-PONDER PENALTY LOSS

We optimize the sum of the language modeling loss and an auxiliary penalty:

$$\mathcal{L} \;=\; \mathcal{L}_{\mathrm{CE}} \;+\; \mathcal{L}_{\mathrm{aux}},$$

where $\mathcal{L}_{\mathrm{CE}}$ is the standard next-token cross-entropy, and $\mathcal{L}_{\mathrm{aux}}$ encourages earlier stopping when additional pondering steps yield diminishing returns.

To obtain a step-dependent signal of whether early steps already suffice, we define a partial integrated state at truncation depth i:

$$h_t^{(\le i)} \;=\; \sum_{j=0}^{i} s_{t,j}\, h_t^{(j)},$$

and compute the token-level per-step cross-entropy

$$\ell_{t,i} \;=\; \mathrm{CE}\big(\mathrm{LM}(h_t^{(\le i)}),\, x_{t+1}\big).$$

We map the token-level cross-entropy to a batch-averaged penalty ratio

$$\rho_i \;=\; \mathbb{E}_{t\in\mathcal{T}}\left[1 - \sigma\big(10\,(\ell_{t,i} - 0.5)\big)\right],$$

so that smaller ce_i yields larger ρ_i, more strongly encouraging the model to allocate less probability mass to late steps.

In practice, we instantiate this preference via a "bottom-mass" penalty on the mask score values at each step. For each step $k \geq 1$, define

$$\Delta\rho_k \;=\; \max(\rho_k - \rho_{k-1},\, 0),$$

and penalize the mean of the smallest $\Delta\rho_k$ fraction of the flattened values $w_{\cdot,k}$. The auxiliary loss is

$$\mathcal{L}_{\text{aux}} \;=\; \lambda \sum_{k=1}^{K} \text{mean}\Big(\,\text{Bottom}_{\Delta\rho_k}\,(w_{\cdot,k})\Big),$$

where $\text{Bottom}_p(\cdot)$ denotes the smallest p fraction of elements in a set, and λ controls the compute–quality trade-off.

4 Experiments

To evaluate token-adaptive pondering, we ask whether our approach can turn inference compute from a uniform per-token load into an allocatable resource while preserving generation quality. We answer this through six targeted experiments: (1) Pareto curves of perplexity versus inference compute under matched pretraining; (2) wall-clock inference latency and throughput; (3) downstream task performance reported alongside compute; (4) continual pretraining on a pretrained 3B model; (5) token-difficulty buckets to measure the marginal benefit of additional steps; (6) ablations over step budget and auxiliary-loss design.

4.1 Pareto Efficiency: Performance vs. Inference Compute

We compare PonderLM-3 against representative baselines that introduce additional computation during autoregressive generation, including PonderLM-1 (Zeng et al., 2025b), PonderLM-2 (Zeng et al., 2025a), LoopedLM (Giannou et al., 2023), and MoR (Bae et al., 2025). Figure 3 compares perplexity against normalized inference compute, a compute proxy relative to one vanilla forward.

Settings. All methods use a 70M-scale LLaMA-style architecture and are trained from scratch on the same 15B-token subset of The Pile (Gao et al., 2020) with the same training hyperparameters. For each method and each maximum number of additional computation steps $K \in \{2, 3, 4\}$, we train a separate model from scratch and evaluate it with the same K. To estimate normalized inference compute, for PonderLM-3, each pondering step is weighted by the fraction of tokens that execute it; for MoR, non-recurrent layers are counted once and each recurrent pass is weighted by the fraction of tokens processed.[2]

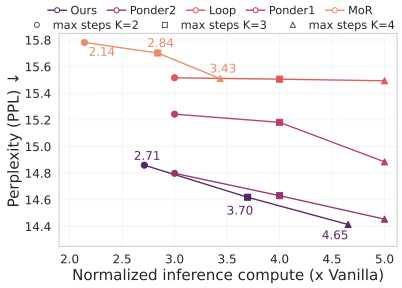


Figure 3: Performance–compute Pareto curve. The x-axis shows normalized inference compute relative to one vanilla forward. PonderLM-3 achieves a better compute–quality trade-off than the baselines.

Results. PonderLM-3 consistently achieves lower perplexity at comparable inference compute, forming a better Pareto frontier across the tested range. Its token-adaptive pruning reduces the amount of recurrent computation executed while preserving or improving model quality.

[2]For MoR, normalized compute is $C = (D_{\text{fix}} + D_{\text{rec}} \sum_r q_r)/(D_{\text{fix}} + D_{\text{rec}})$, where q_r is the fraction of tokens processed at recursion r.

Table 2: Wall-clock inference latency and throughput under the same serving setup.

Method	Prefill (s)	Decode (s)	Total (s)	Throughput (tok/s)
Loop baseline (4 loops)	0.080	17.364	17.459	14.686
PonderLM-1 (3 steps)	0.084	17.850	17.949	14.286
PonderLM-2 (3 steps)	0.186	17.705	17.892	14.402
PonderLM-3 (*max* 3 steps)	0.265	**14.064**	**14.330**	**18.132**

4.2 Wall-Clock Inference Efficiency

The Pareto analysis above uses executed additional computation steps as an inference-compute proxy. We further examine whether these savings translate into practical wall-clock acceleration under autoregressive decoding.

Settings. We evaluate all 410M models under the same serving setup on a single NVIDIA A100 GPU, using bfloat16 precision, FlashAttention-2, greedy decoding, an identical prompt pipeline, 256-token generation, and batch size 1. We further evaluate batched inference with continuous batching for batch sizes larger than one; the implementation and results are provided in Appendix C.

Results. PonderLM-3 achieves the highest end-to-end throughput among the tested variants, confirming that adaptive pondering translates into practical wall-clock acceleration.

The wall-clock speedup can be larger than what the pruning ratio alone suggests. In PonderLM-3, a pruned latent position is not appended to the horizontal KV cache, so subsequent tokens attend over a shorter sequence. Since cumulative autoregressive attention grows approximately quadratically with the horizontal sequence length, reducing latent positions yields additional attention savings beyond the directly pruned computation.

4.3 Downstream Performance with Adaptive Computation

Having established the practical acceleration of adaptive pondering in the previous section, we next examine whether this adaptive execution preserves downstream performance.

Settings. Following the evaluation protocol in PonderLM-2, we test on LAMBADA (Paperno et al., 2016), SciQ (Welbl et al., 2017), HellaSwag (Zellers et al., 2019), PIQA(Bisk et al., 2020), WinoGrande(Sakaguchi et al., 2021), ARC-Easy / ARC-Challenge (Clark et al., 2018), and RACE(Lai et al., 2017). We include PonderLM-2, PonderLM-1, LoopedLM and MoR (Bae et al., 2025) as baselines under the same downstream evaluation protocol. We use a 410M-parameter Pythia-arch model for each baseline, pretrained on a 25B-token subset of The Pile.

Results. Overall, PonderLM-3 achieves the strongest average performance under both zero-shot and few-shot evaluation. It matches or outperforms the fixed-step baselines on most benchmarks, showing that adaptive pondering preserves strong downstream performance. Together with the wall-clock results above, these results suggest that the practical speedup from adaptive pruning can be achieved without sacrificing downstream performance

4.4 Continual Pretraining

Settings. We evaluate continual pretraining on Llama 3B using a two-stage procedure: we first convert the vanilla model to PonderLM-2 on 5B SlimPajama tokens, and then convert PonderLM-2 to PonderLM-3 on 5B FineWeb tokens. All models follow the same data schedule and matched training hyperparameters within each stage. We evaluate generation performance on four benchmarks covering mathematical and scientific reasoning, code generation, and diverse challenging reasoning: GSM8K (Cobbe et al., 2021), GPQA-Diamond (Rein et al., 2023), HumanEval (Chen et al., 2021), and BBH (Suzgun et al., 2023). We use a maximum of one additional pondering step in this setting.

Table 3: Downstream accuracy under zero-shot and five-shot evaluation.

Model	Lambada OpenAI	ARC -E	Lambada Standard	ARC -C	Wino Grande	PIQA	Hella Swag	SciQ	RACE	**Avg.**
0-shot										
LoopedLM (4 loops)	45.6	43.2	35.9	24.7	51.7	66.6	38.3	71.7	30.9	45.4
MoR (*max* 3 steps)	38.6	42.8	24.2	24.1	51.4	64.0	33.6	74.1	29.6	42.5
PonderLM-1 (3 steps)	46.9	43.7	35.2	24.5	**55.2**	66.5	38.6	72.2	32.0	46.1
PonderLM-2 (3 steps)	**53.1**	**48.0**	42.1	25.5	53.8	66.5	42.9	75.1	33.4	48.9
PonderLM-3 (*max* 3 steps)	**53.1**	**48.0**	**43.1**	**26.0**	55.1	**66.9**	**43.1**	**76.4**	**33.6**	**49.5**
5-shot										
LoopedLM (4 loops)	41.1	49.6	33.1	25.3	53.6	67.6	39.2	85.6	31.2	47.4
MoR (*max* 3 steps)	38.5	46.6	35.1	22.7	51.9	66.6	38.7	86.2	**34.1**	46.7
PonderLM-1 (3 steps)	41.2	50.2	32.2	25.3	53.0	66.5	38.8	85.3	32.4	47.2
PonderLM-2 (3 steps)	45.3	52.2	39.1	**27.9**	53.4	67.0	43.3	**92.1**	**34.1**	50.5
PonderLM-3 (*max* 3 steps)	**46.6**	**55.2**	**41.0**	27.7	**55.1**	**67.8**	**43.6**	91.0	33.5	**51.3**

Results. As shown in Table 4, PonderLM-3 consistently improves over PonderLM-2 across all four benchmarks. Its adaptive computation naturally yields different pruning ratios across tasks, and consequently different speedups.

Model	GSM8K	GPQA	HumanEval	BBH
Vanilla	17.51	24.75	9.15	29.17
PonderLM-2	20.17	24.75	13.41	40.17
PonderLM-3	**21.38**	**25.25**	**18.29**	**47.45**
Speedup	1.306×	1.161×	1.229×	1.124×

Table 4: Continual pretraining on Llama 3B. We report task accuracy and end-to-end inference speedup of PonderLM-3 over PonderLM-2.Speedup is computed as the ratio of generated-token throughput (tokens/s).

4.5 Where Extra Compute Helps

We examine whether PonderLM-3 allocates extra computation to tokens that benefit most from it. We define a token's *intrinsic difficulty* as its step-0 token-level prediction loss ℓ_t, before any additional computation.

Settings. We use a 410M-parameter PonderLM-3 with at most $K = 3$ additional computation steps. Statistics are computed on a held-out Pile validation subset of roughly 10^7 tokens.

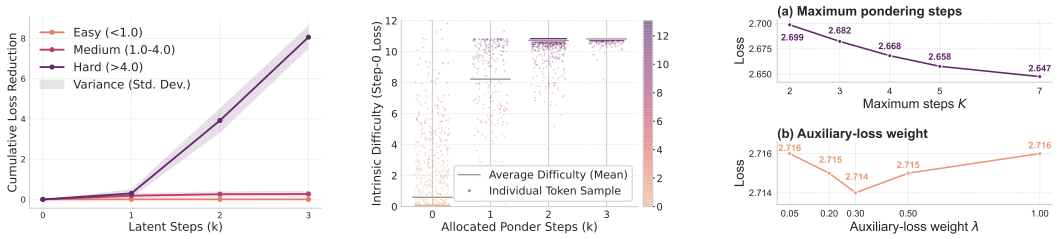
Difficulty-aware utility and allocation. We partition tokens by ℓ_t into easy (< 1), medium (1–4), and hard (> 4) buckets. Let $CE_t^{(k)}$ denote the loss of token t after k additional computation steps. For each bucket B, we measure the average cumulative loss reduction

$$R_B(k) = \frac{1}{|B|} \sum_{t \in B} \left(CE_t^{(0)} - CE_t^{(k)} \right).$$

Figure 4(a) shows that hard tokens benefit substantially more from additional computation, while easy tokens quickly saturate. Figure 4(b) shows that tokens allocated more pondering steps also have higher intrinsic difficulty. Together, PonderLM-3 allocates more computation to tokens that benefit more from it.

4.6 Ablations and Design Choices

We ablate two design choices that directly affect the quality–compute trade-off of PonderLM-3: the maximum number of additional computation steps K and the auxiliary-loss weight λ.



(a) (b) (c)

Figure 4: Mechanism and design analysis. (a) Cumulative loss reduction across token-difficulty buckets. (b) Intrinsic difficulty versus allocated pondering steps. (c) Ablations of the maximum pondering steps K and auxiliary-loss weight λ.

Maximum pondering steps K. We first study how the maximum allowed pondering depth affects pretraining performance. We train 70M-parameter LLaMA-style models on a 15B-token subset of The Pile and sweep $K \in \{2, 3, 4, 5, 7\}$. Figure 4(c) shows that increasing K consistently lowers pretraining cross-entropy, indicating that additional pondering steps provide useful extra capacity for refining token representations. In practice, however, larger K also increases training and evaluation cost. Considering both resource budget and diminishing returns at higher depths, we use $K = 3$ as the default setting in all main results.

Auxiliary-loss weight λ. Figure 4(c) shows that pretraining loss varies only mildly across a broad range of λ after 5B tokens of pretraining on the Pile, suggesting that our method does not rely on narrowly tuned auxiliary-loss weighting. We therefore use a fixed default value of $\lambda = 0.2$ in the main experiments. Additional ablations over α and β in the mapping $\rho_i = 1 - \sigma(\alpha(\mathrm{ce}_i - \beta))$ are provided in Appendix D; the results remain stable, suggesting that this mapping is not sensitive to these parameters.

5 Conclusion and Limitations

In this paper, we introduce PonderLM-3, a token-adaptive pondering framework that learns to allocate different amounts of computation to different tokens. Under matched inference compute, PonderLM-3 achieves a more favorable perplexity–compute trade-off than fixed-step pondering and adaptive baselines, while allocating more computation to harder tokens and pruning easy tokens earlier. We further show that the same mechanism can be applied through continual pretraining to a pretrained 3B model, extending token-adaptive pondering beyond from-scratch training. Overall, our results suggest that adaptively allocating computation across tokens provides a practical new dimension for scaling language-model inference.

Although we validate continual pretraining on a pretrained 3B model, most controlled from-scratch pretraining and mechanism analyses are still conducted at smaller scales. In addition, the current formulation adapts per-token computation only within a globally predefined maximum pondering depth K.

AI use statement

In this work, we used generative AI tools to assist with experimental implementation, debugging, result interpretation, literature search and summarization, code editing, and improving the readability of the manuscript. We did not use generative AI to generate synthetic datasets or formulate or prove mathematical claims. All AI-assisted outputs were reviewed and verified by the authors: code was tested through the corresponding experiments, reported results were obtained from our experimental runs, and relevant literature and citations were checked against the original sources. The authors made all final decisions on the methodology, experiments, analysis, and manuscript content, and take responsibility for the final content of this work.

Ethics statement

This work does not involve human-subject studies or the collection of new personal or sensitive data, and we do not release a new dataset. Our experiments use existing language-model pretraining corpora and standard evaluation benchmarks. PonderLM-3 is a general-purpose language-modeling method for adaptive inference computation and, like other methods that improve model capability or efficiency, may be used in systems with both beneficial and harmful applications. Models trained with our approach may also inherit biases, inaccuracies, or other undesirable behaviors from their training data and pretrained backbones; addressing these issues is outside the scope of this work. The authors take responsibility for the research design, analysis, and conclusions presented in this paper.

Reproducibility statement

We provide the model architecture, token-adaptive pondering mechanism, training objective, and inference procedure in the main paper. The experimental section specifies the model scales, training data and budgets, baselines, and evaluation protocols, with additional implementation and training details provided in the appendix. We report the key hyperparameters and experimental settings to facilitate reproduction of our results. We plan to release the source code and configurations upon publication.

References

Sangmin Bae, Yujin Kim, Reza Bayat, Sungnyun Kim, Jiyoun Ha, Tal Schuster, Adam Fisch, Hrayr Harutyunyan, Ziwei Ji, Aaron Courville, et al. Mixture-of-recursions: Learning dynamic recursive depths for adaptive token-level computation. *arXiv preprint arXiv:2507.10524*, 2025.

Andrea Banino, Jan Balaguer, and Charles Blundell. Pondernet: Learning to ponder. *arXiv preprint arXiv:2107.05407*, 2021.

Yonatan Bisk, Rowan Zellers, Jianfeng Gao, Yejin Choi, et al. Piqa: Reasoning about physical commonsense in natural language. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pp. 7432–7439, 2020.

Ziyang Cai, Xingyu Zhu, Yihe Dong, Yinghui He, and Sanjeev Arora. Tˆ 2mlr: Transformer with temporal middle-layer recurrence. *arXiv preprint arXiv:2607.15178*, 2026.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Yilong Chen, Junyuan Shang, Zhenyu Zhang, Yanxi Xie, Jiawei Sheng, Tingwen Liu, Shuohuan Wang, Yu Sun, Hua Wu, and Haifeng Wang. Inner thinking transformer: Leveraging dynamic depth scaling to foster adaptive internal thinking. *arXiv preprint arXiv:2502.13842*, 2025.

Peter Clark, Isaac Cowhey, Oren Etzioni, Tushar Khot, Ashish Sabharwal, Carissa Schoenick, and Oyvind Tafjord. Think you have solved question answering? try arc, the ai2 reasoning challenge. *arXiv preprint arXiv:1803.05457*, 2018.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

Mostafa Dehghani, Stephan Gouws, Oriol Vinyals, Jakob Uszkoreit, and Łukasz Kaiser. Universal transformers. *arXiv preprint arXiv:1807.03819*, 2018.

Mostafa Elhoushi, Akshat Shrivastava, Diana Liskovich, Basil Hosmer, Bram Wasti, Liangzhen Lai, Anas Mahmoud, Bilge Acun, Saurabh Agarwal, Ahmed Roman, et al. Layerskip: Enabling early exit inference and self-speculative decoding. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 12622–12642, 2024.

Rao Fu, Zixuan Yang, Jiankun Zhang, Jing Ma, Hechang Chen, Yu Li, and Yi Chang. Simply stabilizing the loop via fully looped transformer. *arXiv preprint arXiv:2605.18797*, 2026.

Tianyu Fu, Yichen You, Zekai Chen, Guohao Dai, Huazhong Yang, and Yu Wang. Think-at-hard: Selective latent iterations to improve reasoning language models. *arXiv preprint arXiv:2511.08577*, 2025a.

Yichao Fu, Xuewei Wang, Yuandong Tian, and Jiawei Zhao. Deep think with confidence. *arXiv preprint arXiv:2508.15260*, 2025b.

Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, et al. The pile: An 800gb dataset of diverse text for language modeling. *arXiv preprint arXiv:2101.00027*, 2020.

Jonas Geiping, Sean McLeish, Neel Jain, John Kirchenbauer, Siddharth Singh, Brian R Bartoldson, Bhavya Kailkhura, Abhinav Bhatele, and Tom Goldstein. Scaling up test-time compute with latent reasoning: A recurrent depth approach. *arXiv preprint arXiv:2502.05171*, 2025.

Angeliki Giannou, Shashank Rajput, Jy-yong Sohn, Kangwook Lee, Jason D Lee, and Dimitris Papailiopoulos. Looped transformers as programmable computers. In *International Conference on Machine Learning*, pp. 11398–11442. PMLR, 2023.

Alex Graves. Adaptive computation time for recurrent neural networks. *arXiv preprint arXiv:1603.08983*, 2016.

Shibo Hao, Sainbayar Sukhbaatar, DiJia Su, Xian Li, Zhiting Hu, Jason Weston, and Yuandong Tian. Training large language models to reason in a continuous latent space. *arXiv preprint arXiv:2412.06769*, 2024.

Yixin He and Lumingyuan Tang. Learning to ponder: Adaptive reasoning in latent space. *arXiv preprint arXiv:2509.24238*, 2025.

Sukjun Hwang, Brandon Wang, and Albert Gu. Dynamic chunking for end-to-end hierarchical sequence modeling. *arXiv preprint arXiv:2507.07955*, 2025.

Alexia Jolicoeur-Martineau. Less is more: Recursive reasoning with tiny networks. *arXiv preprint arXiv:2510.04871*, 2025.

Guokun Lai, Qizhe Xie, Hanxiao Liu, Yiming Yang, and Eduard Hovy. Race: Large-scale reading comprehension dataset from examinations. *arXiv preprint arXiv:1704.04683*, 2017.

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

Shuzhen Li, Yifan Zhang, Jiacheng Guo, Quanquan Gu, and Mengdi Wang. Deeploop: Depth scaling for looped transformers. *arXiv preprint arXiv:2607.13491*, 2026.

Denis Paperno, Germán Kruszewski, Angeliki Lazaridou, Quan Ngoc Pham, Raffaella Bernardi, Sandro Pezzelle, Marco Baroni, Gemma Boleda, and Raquel Fernández. The lambada dataset: Word prediction requiring a broad discourse context. *arXiv preprint arXiv:1606.06031*, 2016.

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. *arXiv preprint arXiv:2404.02258*, 2024.

David Rein, Betty Li Hou, Asa Cooper Stickland, Jackson Petty, Richard Yuanzhe Pang, Julien Dirani, Julian Michael, and Samuel R Bowman. Gpqa: A graduate-level google-proof q&a benchmark. *arXiv preprint arXiv:2311.12022*, 2023.

Keisuke Sakaguchi, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. Winogrande: An adversarial winograd schema challenge at scale. *Communications of the ACM*, 64(9):99–106, 2021.

Nikunj Saunshi, Nishanth Dikkala, Zhiyuan Li, Sanjiv Kumar, and Sashank J Reddi. Reasoning with latent thoughts: On the power of looped transformers. *arXiv preprint arXiv:2502.17416*, 2025.

Daniel Scalena, Leonidas Zotos, Elisabetta Fersini, Malvina Nissim, and Ahmet Üstün. Eager: Entropy-aware generation for adaptive inference-time scaling. *arXiv preprint arXiv:2510.11170*, 2025.

Tal Schuster, Adam Fisch, Jai Gupta, Mostafa Dehghani, Dara Bahri, Vinh Tran, Yi Tay, and Donald Metzler. Confident adaptive language modeling. *Advances in Neural Information Processing Systems*, 35:17456–17472, 2022.

Kristian Schwethelm, Daniel Rueckert, and Georgios Kaissis. Depth-adaptive inference of looped language models via continuous depth batching. *arXiv preprint arXiv:2608.09444*, 2026.

Shixiang Song, He Li, Zitong Wang, Boyi Zeng, Feichen Song, Yixuan Wang, Zhiqin John Xu, Ziwei He, and Zhouhan Lin. Adaponderlm: Gated pondering language models with token-wise adaptive depth. *arXiv preprint arXiv:2603.01914*, 2026.

Mirac Suzgun, Nathan Scales, Nathanael Schärli, Sebastian Gehrmann, Yi Tay, Hyung Won Chung, Aakanksha Chowdhery, Quoc Le, Ed H Chi, Denny Zhou, et al. Challenging big-bench tasks and whether chain-of-thought can solve them. In *Findings of the Association for Computational Linguistics: ACL 2023*, pp. 13003–13051, 2023.

Johannes Welbl, Nelson F Liu, and Matt Gardner. Crowdsourcing multiple choice science questions. *arXiv preprint arXiv:1707.06209*, 2017.

Haoyi Wu, Zhihao Teng, and Kewei Tu. Parallel continuous chain-of-thought with jacobi iteration. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 914–926, 2025.

Liu Yang, Kangwook Lee, Robert Nowak, and Dimitris Papailiopoulos. Looped transformers are better at learning learning algorithms. In *International conference on learning representations*, volume 2024, pp. 42195–42214, 2024.

Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. Hellaswag: Can a machine really finish your sentence? *arXiv preprint arXiv:1905.07830*, 2019.

Boyi Zeng, He Li, Shixiang Song, Yixuan Wang, Ziwei He, Xinbing Wang, and Zhouhan Lin. Pretraining llm with latent thoughts in continuous space. *arXiv e-prints*, pp. arXiv–2509, 2025a.

Boyi Zeng, Shixiang Song, Siyuan Huang, Yixuan Wang, He Li, Ziwei He, Xinbing Wang, Zhiyu Li, and Zhouhan Lin. Pretraining language models to ponder in continuous space. *arXiv preprint arXiv:2505.20674*, 2025b.

Rui-Jie Zhu, Zixuan Wang, Kai Hua, Tianyu Zhang, Ziniu Li, Haoran Que, Boyi Wei, Zixin Wen, Fan Yin, He Xing, et al. Scaling latent reasoning via looped language models. *arXiv preprint arXiv:2510.25741*, 2025.

A FLASHATTENTION-2 COMPATIBILITY FOR DIFFERENTIABLE ATTENTION MASK

FlashAttention-2 (FA2) kernels typically support boolean attention masks (causal and padding) but do not natively expose an interface for adding a dense, real-valued bias term to the attention logits.

	x_1	$h_1^{(0)}$	$h_1^{(1)}$	x_2	$h_2^{(0)}$	$h_2^{(1)}$	x_3	$h_3^{(0)}$	$h_3^{(1)}$
x_1	1								
$h_1^{(0)}$	1	$w_{1,1}$							
$h_1^{(1)}$	1	$w_{1,1}$	$w_{1,2}$						
x_2	1	$w_{1,1}$	$w_{1,2}$	1					
$h_2^{(0)}$	1	$w_{1,1}$	$w_{1,2}$	1	$w_{2,1}$				
$h_2^{(1)}$	1	$w_{1,1}$	$w_{1,2}$	1	$w_{2,1}$	$w_{2,2}$			
x_3	1	$w_{1,1}$	$w_{1,2}$	1	$w_{2,1}$	$w_{2,2}$	1		
$h_3^{(0)}$	1	$w_{1,1}$	$w_{1,2}$	1	$w_{2,1}$	$w_{2,2}$	1	$w_{3,1}$	
$h_3^{(1)}$	1	$w_{1,1}$	$w_{1,2}$	1	$w_{2,1}$	$w_{2,2}$	1	$w_{3,1}$	$w_{3,2}$

Figure 5: Soft attention mask over an interleaved sequence. Rows correspond to *queries*, and columns correspond to *keys*. Entries show the multiplicative mask values applied to attention weights. Blank entries are masked out by causality or padding.

Soft attention mask pattern. Figure 5 visualizes the differentiable attention mask used in our interleaved sequence of observed tokens and latent states. In our formulation, the attention logits contain two masking components: (i) the standard boolean mask M (causal and padding), and (ii) an additional dense bias matrix G induced by the mask score w. While M is directly supported by FA2, the dense, real-valued bias G is not. We therefore express G in an equivalent vector-bias form (VecBias), allowing the bias to be incorporated while remaining compatible with FA2.

Target attention form. The original attention is

$$\mathrm{Softmax}\left(\frac{QK^\top}{\sqrt{d}} + M\right)V,$$

and we aim to compute

$$\mathrm{Softmax}\left(\frac{QK^\top}{\sqrt{d}} + M + G\right)V.$$

Here

$$G = \log(\text{ponder mask score}) = \log w,$$

and G takes the following broadcasted matrix form:

$$G = \begin{bmatrix} \log w_1 & \log w_2 & \cdots & \log w_n \\ \log w_1 & \log w_2 & \cdots & \log w_n \\ \vdots & \vdots & \ddots & \vdots \\ \log w_1 & \log w_2 & \cdots & \log w_n \end{bmatrix} \in \mathbb{R}^{n\times n}.$$

That is, the additive bias is constant across query rows and depends only on the key position, matching the soft attention pattern in Figure 5. Therefore, G can be represented compactly by a per-key bias vector

$$g = [\log w_1 \quad \log w_2 \quad \cdots \quad \log w_n]^\top \in \mathbb{R}^n.$$

FA2-compatible VecBias implementation. Standard FlashAttention-2 does not natively expose an interface for adding such a real-valued per-key bias to the attention logits. We therefore implement a lightweight VecBias extension to FA2 that fuses the bias vector g directly into the attention-score computation before the softmax.

For query position i and key position j, the resulting attention logit follows the target form

$$\ell_{ij} = \frac{q_i^\top k_j}{\sqrt{d}} + M_{ij} + g_j = \frac{q_i^\top k_j}{\sqrt{d}} + M_{ij} + \log w_j.$$

The vector bias is shared across query positions and attention heads, consistent with the broadcasted structure of G. Thus, the implementation only stores a per-key bias vector rather than materializing the full $n \times n$ matrix G. During the tiled FA2 computation, the corresponding scalar bias is added to each attention score before softmax normalization.

Our VecBias extension supports both the forward and backward passes, allowing the differentiable attention mask to be trained end-to-end while retaining the memory-efficient computation of FlashAttention-2.

B Convergence of Jacobi-Style Parallel Iterations

This appendix clarifies why the Jacobi-style parallel iterations used during training rapidly approach the sequential refinement process used at inference time. We organize the discussion into three parts:

1. an intuitive example showing how Jacobi updates use the previous iteration's latent estimates to approximate the causal dependencies of sequential inference;
2. a fixed-point view of the training trajectory over the joint state of latent representations and router-produced mask scores, explaining why its converged solution aligns with the sequential-inference solution;
3. empirical evidence that this convergence is fast in practice, with the RMSE between Jacobi-iteration states and sequential-inference states decaying rapidly within a few iterations.

Sequential refinement and Jacobi intuition. We first illustrate the Jacobi-style update with a simple three-token example. Consider three observed tokens A, B, C with latent states a, b, c. In sequential inference, later positions can condition on the already refined latent states of earlier positions. Abstractly, the sequential solution satisfies

$$a^* = F_A(A), \qquad b^* = F_B(A, a^*, B), \qquad c^* = F_C(A, a^*, B, b^*, C).$$

Thus, the refined state b^* depends on the refined latent state a^* of token A, and c^* depends on both a^* and b^*.

Jacobi training approximates this sequential refinement by repeatedly updating all latent states in parallel using the previous iteration's estimates. It starts from an initial parallel estimate

$$a^{(0)}, b^{(0)}, c^{(0)} = \mathrm{LM}(A, B, C),$$

where $a^{(0)}$ is computed from the prefix A, $b^{(0)}$ from the prefix A, B, and $c^{(0)}$ from the prefix A, B, C. At this point, $b^{(0)}$ has not yet used the refined latent state of token A, and $c^{(0)}$ has not yet used the refined latent states of tokens A and B.

At the first Jacobi iteration, the model updates all latent states using the previous estimates:

$$a^{(1)} = F_A(A), \qquad b^{(1)} = F_B(A, a^{(0)}, B), \qquad c^{(1)} = F_C(A, a^{(0)}, B, b^{(0)}, C).$$

At the second iteration, the update becomes

$$a^{(2)} = F_A(A), \qquad b^{(2)} = F_B(A, a^{(1)}, B), \qquad c^{(2)} = F_C(A, a^{(1)}, B, b^{(1)}, C).$$

Therefore, as the estimates of earlier latent states become more accurate, later states are updated with increasingly accurate latent-prefix information. If $a^{(1)} \approx a^*$ and $b^{(1)} \approx b^*$, then the next update also gives $b^{(2)} \approx b^*$ and $c^{(2)} \approx c^*$. This is the basic Jacobi intuition: each iteration uses the previous iteration's latent estimates to move the parallel training trajectory closer to the sequential-inference solution.

PonderLM-3 additionally introduces a router, but the same logic applies to the joint state of hidden representations and routing signals. At each iteration r, the router is re-applied to the updated hidden states to obtain

$$s_t^{(r)} = R\left(h_t^{(r)}\right), \qquad w_t^{(r)} = \mathrm{TailCDF}\left(s_t^{(r)}\right).$$

The next hidden states are then computed using the previous iteration's hidden states and mask scores. Thus, the scores are not predicted once and used statically; instead, hidden states and router outputs are jointly refined across Jacobi iterations.

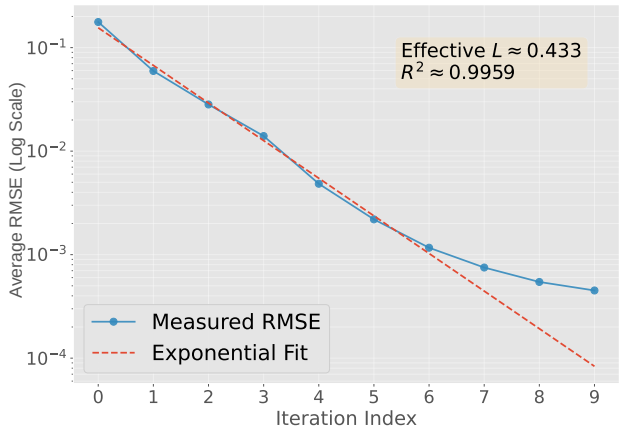


Figure 6: RMSE convergence of Jacobi-style parallel iterations.We report the average RMSE between the hidden states $H^{(r)}$ after the r-th Jacobi-style parallel iteration and the sequential-inference hidden states H_{seq}, plotted over iteration index r on a log scale.The error decreases rapidly and is well approximated by an exponential fit with effective contraction factor $L \approx 0.433$ and $R^2 \approx 0.9959$, indicating that the Jacobi trajectory approaches the sequential-inference solution within a few iterations.

Convergence of the Jacobi trajectory. We now show that the Jacobi-style training trajectory converges to the same refinement process as sequential inference. Let $S^{(r)}$ denote the interleaved latent sequence after iteration r, and let $W^{(r)}$ denote the corresponding router-produced mask scores:

$$W^{(r)} = \{w_t^{(r)}\}_{t=1}^{T}.$$

The next Jacobi update uses the previous interleaved sequence and mask scores as its causal context. We write this joint update as

$$(S^{(r+1)}, W^{(r+1)}) = \mathcal{T}(S^{(r)}, W^{(r)}),$$

where $\mathcal{T}$ denotes one parallel Jacobi-style refinement step.

The update has a causal triangular dependency structure. Although all latent positions are updated in parallel, the state at position t only depends on positions up to t, and never on future positions. Therefore, the first position has no earlier latent position to depend on, so its update is determined by the input and becomes fixed first. Once the first position is fixed, the second position sees a fixed causal prefix, so its update also becomes fixed. Repeating the same argument from left to right, once positions $1, \ldots, t-1$ are fixed, the update at position t is determined by a fixed prefix and becomes fixed as well.

Consequently, after sufficiently many Jacobi iterations, all positions stop changing and the joint state reaches a fixed point:

$$(S^{(r+1)}, W^{(r+1)}) = (S^{(r)}, W^{(r)}).$$

This establishes the convergence of the Jacobi trajectory. The remaining question is how quickly the Jacobi trajectory approaches this sequential-inference solution in practice, which we measure next.

Empirical convergence rate. We next measure how quickly the Jacobi trajectory approaches the sequential-inference solution. Let $H^{(r)}$ denote the hidden states associated with the interleaved latent sequence $S^{(r)}$, and let H_{seq} denote the hidden states produced by sequential inference. We compute

$$d_r = \text{RMSE}(H^{(r)}, H_{\text{seq}}),$$

where d_r measures the discrepancy between the hidden states after the r-th Jacobi iteration and those from sequential inference.

Figure 6 shows that d_r decreases rapidly within a few iterations. The decay is well fit by an exponential form:

$$d_r \approx CL^r,$$

or equivalently,

$$\log_{10} d_r \approx r \log_{10} L + \log_{10} C.$$

The fitted value is

$$L \approx 0.433,$$

with

$$R^2 \approx 0.9959.$$

Here, L is the effective contraction factor: each additional Jacobi iteration reduces the remaining hidden-state discrepancy to about 43.3% of the previous one. The high R^2 indicates that the exponential-decay model closely matches the measured trajectory. Since the mask scores $W^{(r)}$ are deterministically produced from the hidden states through the router, this hidden-state alignment also implies alignment of the corresponding router-produced mask scores. Therefore, in practice, the Jacobi trajectory approaches the sequential-inference solution within only a few iterations.

C Additional Wall-Clock Inference Results

We provide additional wall-clock results beyond the single-request comparison in the main text. In particular, we evaluate whether the throughput advantage of PonderLM-3 persists as the batch size increases. PonderLM-2 and PonderLM-3 are evaluated under strictly aligned inference settings; further implementation details of continuous batching are provided in Appendix C.2.

C.1 Batched Inference

Table 5 reports throughput from single-request inference to larger batch sizes. PonderLM-3 remains faster than the fixed-step PonderLM-2 across all evaluated batch sizes, showing that its adaptive-computation advantage persists under batched inference.

Table 5: Wall-clock decode throughput under batched inference. All results use 256-token generation.

B	PonderLM-2 (tok/s)	PonderLM-3 (tok/s)	Speedup
1	14.510	**18.260**	1.258×
2	28.648	**35.993**	1.256×
4	57.124	**71.701**	1.255×
8	114.370	**140.932**	1.232×
16	232.632	**277.561**	1.193×

C.2 Continuous Batching

When $B > 1$, batch elements may require different numbers of latent pondering steps at the same real-token position. Under naive synchronized batching, an element that stops early cannot advance to its next real token until deeper elements finish their remaining pondering. Following Continuous Depth Batching (Schwethelm et al., 2026), we instead let batch elements progress independently and dynamically combine their next required real or latent positions into a batched GPU forward, as illustrated in Figure 7.

For long-running serving, different generations also finish at different times. We therefore immediately refill released batch slots with waiting inputs. On heterogeneous pondering trajectories, refill increases average batch occupancy from about 87–88% to 94–95%, yielding an additional 7.6% and 10.2% throughput improvement for 1024- and 2048-token generation, respectively. This verifies that continuous batching can recover utilization otherwise lost to heterogeneous adaptive computation.

C.3 Implementation Fairness and Optimization Alignment

To ensure that the measured wall-clock gains arise from adaptive pruning rather than asymmetric systems optimization, we align all serving components shared by PonderLM-2 and PonderLM-3.

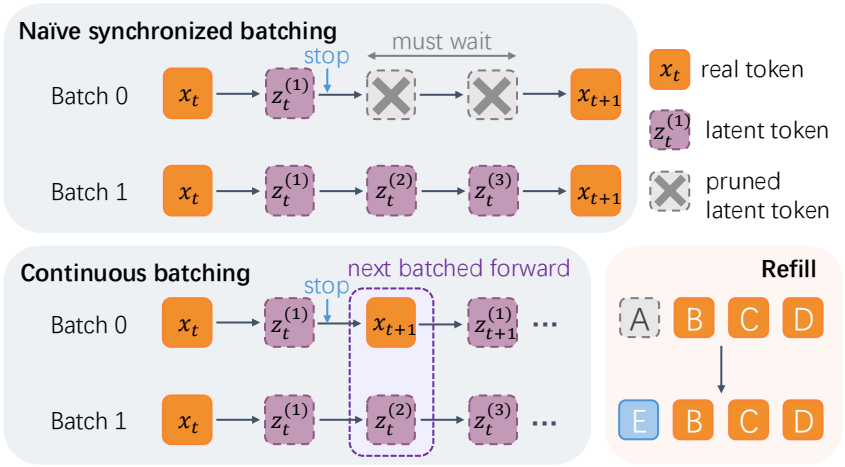


Figure 7: Continuous batching for adaptive pondering. **(1)** Under naïve synchronized batching, Batch 0 stops after $z_t^{(1)}$ but must wait for Batch 1 to finish its remaining pondering steps. **(2)** With continuous batching, each batch element progresses independently, so the next batched forward can jointly process Batch 0's x_{t+1} and Batch 1's $z_t^{(2)}$. **(3)** When a generation finishes, its released slot can be immediately filled by a waiting request (refill) to maintain batch occupancy.

Both models use the same Transformer backbone, BF16 precision, preallocated KV caches with in-place append, FlashAttention-2 KV-cache decoding, batched language-model heads and argmax, as well as identical warmup and timing protocols. Optimizations specific to PonderLM-3 are restricted to operations introduced by adaptive pondering itself, such as routing, gate bias, adaptive scheduling, and hidden-state fusion.

Equal-work control. We first force PonderLM-3 to execute the same four horizontal positions per real token as PonderLM-2, while retaining all additional routing, gating, and fusion operations required by PonderLM-3. As shown in Table 6, PonderLM-3 is consistently no faster than PonderLM-2 when the executed computation is matched. This rules out a faster shared serving implementation as the source of the observed acceleration.

Table 6: Equal-work control. PonderLM-3 is forced to execute the same four horizontal positions per real token as PonderLM-2. Values below $1\times$ indicate that PonderLM-3 is slower under matched computation.

Batch size	1	2	4	8	16	32
PonderLM-3 / PonderLM-2	0.968×	0.983×	0.969×	0.937×	0.928×	0.909×

Adaptive pruning control. We next isolate the effect of adaptive pruning within the same PonderLM-3 implementation. For the 256-token workload, PonderLM-3 executes an average of 3.106 horizontal positions per real token, compared with four positions under full computation. This corresponds to a theoretical speedup of $4/3.106 = 1.288\times$. As shown in Table 7, the measured speedup closely follows this theoretical value across batch sizes.

Table 7: Effect of adaptive pruning within the same PonderLM-3 implementation. The theoretical speedup is computed from the reduction in executed horizontal positions.

Batch size	1	2	4	8	16	32
Measured speedup	1.307×	1.278×	1.294×	1.294×	1.303×	1.257×
Theoretical speedup	1.288×	1.288×	1.288×	1.288×	1.288×	1.288×

Together, these controls show that the wall-clock advantage of PonderLM-3 does not come from more aggressive optimization of shared serving components. Under matched computation, PonderLM-3 is slightly slower because of the additional overhead introduced by adaptive pondering; once adaptive pruning is enabled, the resulting speedup closely matches the reduction in executed horizontal positions.

D Additional Auxiliary-Loss Ablations

This appendix provides additional ablations for the scale and shift parameters in the mapping from the per-step CE proxy to the penalty ratio, evaluated after 5B tokens of pretraining on the Pile. Recall that the penalty ratio is computed as

$$\rho_i = 1 - \sigma\big(\alpha(\mathrm{ce}_i - \beta)\big),$$

where α controls the sharpness of this mapping and β controls its shift. These parameters do not change the form of the auxiliary objective; they only determine how the per-step CE proxy is mapped to the penalty ratio.

Table 8: Ablation over the scale α and shift β in the mapping from the per-step CE proxy to the penalty ratio. We report pretraining cross-entropy loss under the same ablation setting as Section 4.6.

β	$\alpha = 5$	$\alpha = 10$	$\alpha = 20$
0.3	2.711	2.714	2.715
0.5	2.712	2.710	2.713
0.6	2.714	2.717	2.718
0.8	2.711	2.713	2.715

As shown in Table 8, the loss varies only mildly across a broad range of α and β. This indicates that the auxiliary loss is not sensitive to the particular choice of this mapping. In our main experiments, we use $(\alpha, \beta) = (10, 0.5)$ as a simple default setting.

E Analysis of Pondering Step Distribution

To better understand the learned halting behavior, we analyze how pondering depth varies across different linguistic contexts. This analysis focuses on empirical patterns in the resulting step distribution rather than identifying the specific causal signals used by the halting policy.

Patterns in Pondering Depth

Tables 9 and 10 show that the learned pondering depth is highly non-uniform across tokens and linguistic positions. We observe two clear patterns.

Token-dependent computation. As shown in Table 9, pondering depth differs substantially across selected frequent function tokens. The token "a" almost always reaches the maximum depth (Avg Step ≈ 3.0), whereas "the" and "of" are pruned substantially more often. This indicates that the learned policy does not distribute computation uniformly across token types. However, this comparison alone does not distinguish effects associated with token identity from those arising from the surrounding context.

Position-dependent computation within compound words. Table 10 shows a clear difference between non-terminal and terminal tokens within compound words. Among non-terminal tokens, approximately 28% halt before the maximum depth, whereas all terminal tokens in this sample reach Category 3. Thus, pondering depth is strongly associated with the token's position within a compound word, with terminal positions receiving consistently deeper computation in this analysis.

Table 9: Pondering step statistics for selected frequent tokens, illustrating token-dependent variation in computation depth.

Token	Count	Avg Step	Pruned %
a	4,203	2.999	0.07%
the	14,727	2.257	74.28%
of	6,840	2.337	66.27%

Table 10: Halting distribution for non-terminal and terminal tokens within compound words. Terminal positions exhibit consistently deeper computation in this sample.

Category	Non-terminal	Terminal
Category 0	6.24%	0.00%
Category 1	8.13%	0.00%
Category 2	13.96%	0.00%
Category 3	71.67%	100.00%
Total Count	5,980	2,164

Interpretation of Learned Halting Patterns

Together, Tables 9 and 10 show that the learned halting policy develops structured, non-uniform computation patterns. Pondering depth varies substantially across token identities and also across positions within compound words. In particular, terminal compound-word tokens consistently receive deeper computation than non-terminal tokens in our sample.

These observations are consistent with the goal of token-adaptive pondering: instead of applying a uniform computation depth to every token, the learned policy allocates different amounts of computation across linguistic contexts. At the same time, these statistics are correlational. The token-level comparison in Table 9 does not separate token identity from contextual effects, and the compound-word analysis establishes an association with token position rather than the specific mechanism driving the halting decision. We therefore view these results as descriptive evidence of structured adaptive computation rather than as a mechanistic identification of predictive uncertainty.