DEPTH-ADAPTIVE INFERENCE OF LOOPED LANGUAGE MODELS VIA CONTINUOUS DEPTH BATCHING

Anonymous authors
Paper under double-blind review

ABSTRACT

A main promise of looped language models is *depth-adaptive inference*. By looping a block of shared layers a variable number of times, the model can use less compute for "easy" tokens and more for "hard" ones. However, tokens with different numbers of loops cannot share a uniform forward pass and therefore cannot be handled by standard batching systems such as vLLM. The practical value of depth-adaptive inference thus hinges on whether batching can be made efficient. We introduce the first efficient method for depth-adaptive looped LMs via *continuous depth batching (CDB)*, which forms new batches *between* loop steps. Our method dynamically schedules looped and non-looped parts of the architecture, manages looped KV-caching, and predicts which tokens will exit the loop in advance so it can prepare batches asynchronously. Experiments on Ouro 1.4B and Huginn 3.5B show that fully looped architectures are best suited to depth-adaptive inference, as large non-looped layers outside the recurrent core (e.g., token embedding, LM head, and unshared transformer blocks) slow down and complicate scheduling. Overall, CDB realizes up to 99% of the estimated maximum speedup available, leaving further gains primarily dependent on model architecture and exit behavior.

1 INTRODUCTION

Efficient batching, i.e., processing multiple sequences in one forward pass, is central to large language model (LLM) inference. Online APIs must handle requests from many users at once, while local workloads like post-training with reinforcement learning and benchmarking require generating rollouts over many samples. Batching is especially important for autoregressive decoding, which generates responses one token at a time. Without batching, each decode step loads the model weights from memory to process a single token. Weight loading often takes much longer than the computation, leaving compute units underutilized; the forward pass is *memory-bandwidth bound* (Anthony et al., 2024). By processing multiple sequences, batching reuses the loaded weights and turns small matrix multiplications into larger, more efficient ones. With a large enough batch, the forward pass can even become *compute bound*, saturating the compute units while weight-loading time becomes negligible.

Hardware constraints and inference systems also increasingly influence LLM architecture design (Anthony et al., 2024; Bian et al., 2026). For example, layer dimensions are chosen based on accelerator utilization, modern attention layers reduce KV-cache memory footprint, and native multi-token prediction enables self-speculative decoding (DeepSeek-AI et al., 2025; Gloeckle et al., 2024).

Looped language models, however, have received little attention from the inference perspective, despite offering several potential advantages. A looped LM repeatedly applies a block of shared layers, called the *recurrent core*, to refine the latent representation before predicting a token. Sharing layers reduces parameter memory relative to an equally deep standard transformer and also makes it possible to reuse KV states across depth (Geiping et al., 2025). The key advantage we focus on is *depth-adaptive decoding*, which allows tokens to exit the recurrent core after different depths (Fig. 1, left) (Dehghani et al., 2019; Graves, 2017). The model spends fewer loops and less compute on "easy" tokens, and more loops on "hard" ones, improving the compute–accuracy trade-off (Zhu et al., 2025).

However, the compute savings from depth-adaptive decoding do not automatically translate into faster inference. The central challenge is that tokens with different numbers of loops cannot share a uniform forward pass and therefore cannot be handled by standard batching. A promising solution is *continuous depth batching (CDB)* (Bae et al., 2025a), a depth-wise scheduler that forms new batches

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.

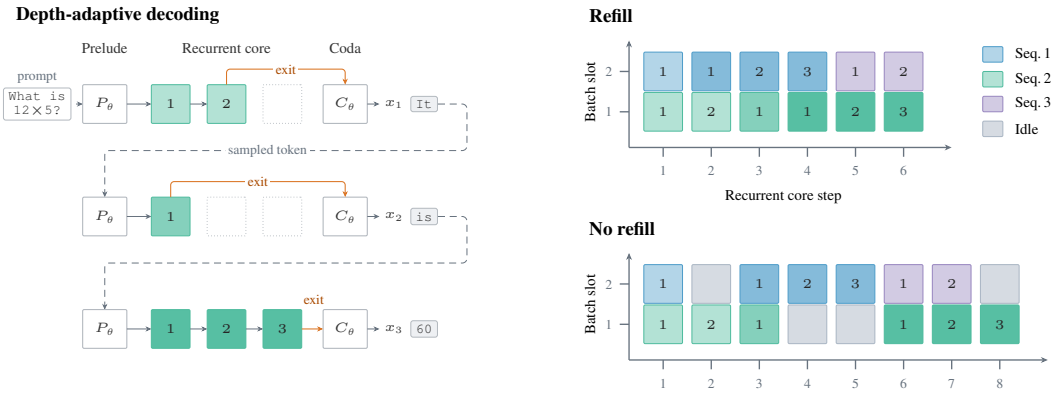

Figure 1: *Left:* the looped LM decodes three tokens, each running the prelude and coda once and the recurrent core a different number of times. *Right:* three sequences under CDB with two concurrent batch slots. Refill fills slots freed by an exit with a new token, keeping a larger batch size and finishing in 6 forward passes. No refill reduces the batch size on token exits, needing 8 forward passes.

between loop steps *within* the forward pass. CDB removes tokens from the loop as they exit and refills the freed slots with new tokens to keep batch size large (Fig. 1, right). However, this makes scheduling much more complex and a naive implementation can incur more overhead than early exit saves, leaving standard fixed-depth batching faster. In this work, we ask: *Can depth-adaptive looped LMs be served efficiently? Which architectural properties determine their speedup?*

For this, we introduce the first end-to-end CDB implementation, addressing several open challenges. Looped LMs contain boundary stages outside the shared core that run only once and must be scheduled alongside changing loop batches: the *prelude* processes each token before it enters the loop, and the *coda* predicts the next token after it exits (Fig. 1, left). New tokens for refilling freed batch slots are also not directly available: the next token of an exit needs to pass through coda and prelude and a new request must first complete prompt prefill and prelude. Furthermore, exit decisions become available only after a loop step finishes, so the scheduler cannot prepare the next recurrent batch in advance, leaving the GPU idle. To address these challenges, we interleave prefill, prelude, and coda batches with loop steps, allowing new and exiting sequences to provide replacement tokens for refill, and use a lookahead gate to overlap batch preparation with GPU execution. We implement the original refill scheduler and a simpler *no-refill* mode that only removes tokens (Fig. 1, right).

To analyze CDB, we derive a prefill-corrected speedup estimate and a hardware-aware latency model that predicts when refill helps. We evaluate CDB in offline throughput and online serving benchmarks on two prominent looped architectures. Our results show that refill is important, especially in the memory-bound regime and for fully looped models such as Ouro (Zhu et al., 2025), whose boundary stages are minimal. On Ouro, CDB realizes up to 99% of the estimated maximum speedup available from early exit. For Huginn (Geiping et al., 2025), costly transformer boundary stages increase the cost of refill and no-refill performs better. Here, CDB realizes 83 to 96% of the available speedup.

Contributions.

1. We enable efficient depth-adaptive inference for looped LMs with the first end-to-end implementation of CDB.
2. We compare theoretical compute savings with wall-clock speed-up by deriving a FLOP-based upper bound and a hardware-aware model of recurrent-step latency.
3. We show how boundary-stage cost, KV cache design, and loop count determine inference efficiency, providing concrete guidance for inference-aware looped-LM architecture design.

2 Problem Setup

The looped-LM architecture is split into a *prelude* P_θ, a shared *recurrent core* R_θ, and a *coda* C_θ. For token t, the recurrent core runs $r_t \leq r_{\max}$ times, while the prelude and coda each run once.

We denote the number of transformer layers in these stages as prelude-core-coda. For example, a 2-4-2 model executes $2{+}4r_t{+}2$ layers, giving an effective depth of 20 at $r_t{=}4$ loops. A fully looped 0-4-0 model has no transformer layers outside the core; its boundary stages (i.e., prelude and coda) only contain the embedding layer and LM head. Within this architecture, depth-adaptive decoding lets r_t vary per token: after each loop, an exit criterion determines whether the token stops looping. Crucially, each token leaves the core at an *input-dependent* depth, decided *at the end of a loop step*.

This per-token variability creates a problem for batched inference. Standard *continuous batching (CB)* schedules only at token boundaries: at each tick, active sequences generate one token, completed sequences are removed, and new sequences are admitted (Yu et al., 2022). Because CB treats the complete forward pass as one scheduling unit, every token in a batch must execute the same operations. With adaptive depth, however, an exited token must run the coda while the remaining tokens may require another loop step. To support this, the CDB scheduler must operate *within* the forward pass by tracking each token's loop progress individually and routing tokens between stages as they exit.

Overall, CDB must address three main challenges: (1) *stage heterogeneity*—a CB step is one uniform forward pass, while CDB must schedule prefill, prelude, recurrent core, and coda separately, each at different batch sizes and frequencies; (2) *missing KV states*—when a token exits at r_t, it writes no KV states for deeper loop steps, leaving later tokens without a state to attend to if they loop deeper; (3) *critical-path exits*—the exit decision is available only *after* step r but determines which tokens enter step $r+1$, so the GPU idles after each loop step until the scheduler has processed the exit.

3 Related Work

Depth-adaptive looped transformers. We focus on methods that decide for each token *individually* when it leaves the loop. Here, learned variants attach a small exit head to the recurrent core (Banino et al., 2021; Frey et al., 2026; Fu et al., 2026; Li et al., 2026; Song et al., 2026; Tan et al., 2023; Zhu et al., 2025), while training-free variants compare latent states or output logits between consecutive loop steps and exit once they converge (Geiping et al., 2025; Kohli et al., 2026; Lee et al., 2026). In contrast, in Mixture-of-Depths-style methods, tokens exit in fixed-sized *groups* (Bae et al., 2025b; Mohtashami et al., 2025; Raposo et al., 2024), while in sequence-level methods, *all* tokens exit at the same depth (Jeddi et al., 2026; Prairie et al., 2026). We study per-token adaptive depth from a *systems* perspective, quantifying how its FLOP savings translate into throughput and latency gains. Importantly, CDB does not change the model's predictions and exit decisions.

Depth-adaptive standard transformers. Early exit has also been studied for standard transformers, where intermediate layers can predict the next token and stop computation early (Elbayad et al., 2020; Elhoushi et al., 2024; Schuster et al., 2022; Yoo et al., 2026). However, efficient batched inference remains difficult because their unshared layers prevent refill across depths and leave missing KV states that must be recomputed or approximated when needed (Bae et al., 2023; Chen et al., 2024; Corro et al., 2023; Liu et al., 2025; Shen & Zou, 2026). We study CDB only for looped LMs, as their shared recurrent core naturally allows cross-depth refill and KV sharing without recomputation.

Continuous depth batching. Bae et al. (2025a) introduced CDB for looped LMs theoretically and estimated its throughput speedup through simulation. Bae et al. (2025b) later evaluated CDB in ablation experiments, excluding prefill, scheduling, and KV-cache updates. In contrast, we include all of these costs and provide the first end-to-end implementation and evaluation of CDB.

4 Method: Continuous Depth Batching

4.1 Stage-wise Scheduling

We conceptually split each request's path from prompt to completed response into four queues: inactive requests waiting for prefill, decode tokens awaiting the prelude, recurrent items ready for their next loop step, and exited tokens awaiting the coda. These queues correspond to the four stages of looped inference, which can be batched and run separately. In principle, the scheduler can execute them in any order, allowing prefill, prelude, recurrent, and coda batches to be interleaved.

Based on this queuing system, we implement two execution modes that differ in how the stages are ordered. In *no-refill* mode (Algorithm 1), stages follow a fixed order: prelude, then recurrent steps

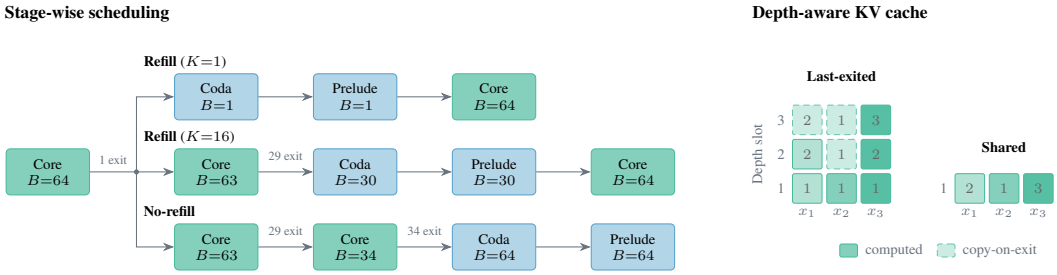

Figure 2: *Left:* three schedules after a single token exits the loop (core); B denotes the stages' batch size. Refill interleaves coda and prelude with recurrent-core steps, while the minimum coda batch size K determines how many exits accumulate before coda runs. No-refill lets the recurrent batch shrink and runs coda after all tokens have exited. *Right:* depth-aware KV cache designs for three tokens that exit at depths $(r_1, r_2, r_3) = (2, 1, 3)$. Each cell shows the loop step whose state is stored.

until *all* tokens have exited, then coda. Exited tokens are removed from the recurrent batch, reducing batch size and thus compute on subsequent loop steps. In *refill* mode (Algorithm 2), the scheduler orders the stages dynamically so that freed slots can be filled by new tokens, keeping batch size high.

For refill, we use a priority rule to determine the dynamic order. We let prelude run directly after prefill or coda, as there is no routing decision between these stages. Among the remaining stages, coda has highest priority because it predicts next tokens for refill and finishing requests. Prefill comes next, since admitting prompts early allows for larger decode batches. Otherwise, we run the recurrent core. Without an additional constraint, this priority system would run the coda after every individual exit. For models with costly coda layers, repeatedly running them on only a few tokens can outweigh the benefit of refill. We therefore introduce a minimum coda batch size K: coda takes priority only after at least K exits have accumulated. See implementation details in Appendix A.

Fig. 2 (left) illustrates what happens after one token exits a recurrent core batch of 64. With refill (K=1), coda and prelude run immediately for the single exited token, so the next recurrent batch refills to B=64. With refill (K=16), the recurrent batch first continues with 63 tokens, allowing more exits to accumulate before coda and prelude run on 30 tokens together. No-refill waits until the entire recurrent batch has exited and processes all 64 tokens through the coda together. A larger K therefore makes coda and prelude batches more efficient, but delays refill and temporarily reduces the recurrent batch size, making it less efficient. Complete scheduling traces are provided in Appendix A.4.

4.2 Depth-aware KV Cache

A depth-indexed KV cache gives each shared layer a separate slot for every loop step, resulting in $r_{\max}$ slots per token. When a token exits early ($r_t < r_{\max}$), its remaining slots stay empty, but later tokens reaching these loop steps need a state to attend to. We consider two ways to provide these missing states (Fig. 2, right), both compatible with modern attention kernels like paged attention (Kwon et al., 2023) and FlashAttention (Dao et al., 2022). A *last-exited cache* copies a token's final KV state into skipped slots upon exit. This generally preserves accuracy and avoids computing the missing states (Zhu et al., 2025), but retains $r_{\max}$ slots and adds copying overhead. A *shared cache* instead gives each recurrent layer a single KV slot per token, reducing recurrent-core KV memory by a factor of $r_{\max}$. Each loop step overwrites this slot, so later tokens always attend to the most recent state of each earlier token, regardless of its exit depth. This avoids empty slots and copying, but may require specialized training to preserve accuracy (Geiping et al., 2025; Vendrell et al., 2026; Zhu et al., 2025).

4.3 Asynchronous Scheduling

CDB introduces frequent CPU interactions between short GPU stages, because CDB schedules every loop and boundary stage separately. After each stage, the scheduler reads back results, selects the next stage, assembles its batch, and transfers the required inputs to the GPU. Doing this synchronously results in much GPU idle time. We therefore prepare batches and transfer data asynchronously while

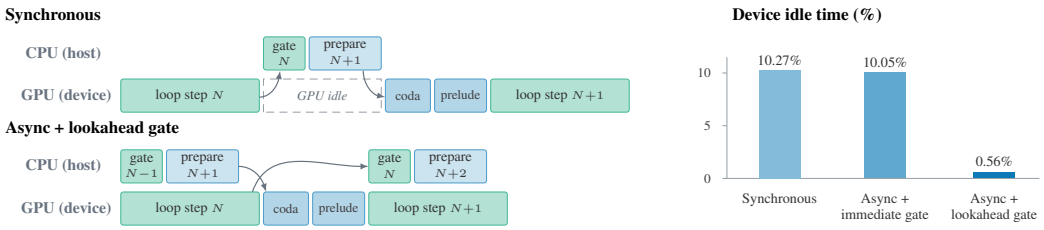

Figure 3: *Left:* synchronous scheduling leaves the GPU idle during exit processing and batch preparation. Asynchronous lookahead overlaps this work with the GPU step. *Right:* asynchronous preparation alone lowers idle time minimally, adding the lookahead gate reduces it to nearly zero.

the GPU processes the current batch (Fig. 3, left). We also use acceleration techniques like static, preallocated tensors, CUDA graphs, and batches padded to the next power of two (Appendix A.5).

However, there is a problem with preparing the next batch in advance: the gate output at loop step r determines which tokens leave the batch, but is only available once step r finishes on the GPU. The CPU must read back the result and build the batch before the GPU can start step $r{+}1$. End-of-sequence (EOS) tokens pose an analogous problem on the token level, which is usually handled by assuming no EOS and catching it one step late, causing one extra generated token. We propose a similar idea with a *lookahead gate*. The exit signal at step r determines whether a token participates in step $r{+}2$ instead of $r{+}1$, giving the CPU a full loop step to prepare the next batch. Unlike the EOS approach, this does not waste compute, since we decide the exit earlier instead of catching it later. In Appendix E.2, we show that the exit decision can be made one step earlier without reducing accuracy. Overall, our CDB implementation reduces GPU idle time to nearly zero (Fig. 3, right).

5 Efficiency Limits of Adaptive Depth

We next study what drives efficiency gains for adaptive depth from the hardware perspective. We find that early exits reduce recurrent-core FLOPs, but fewer FLOPs do not always reduce wall-clock time, especially in the memory bound regime. This is where the refill mode of CDB is important.

FLOP-based speedup bound. Let F_r be the FLOPs of one recurrent-core application for a single token, and F_0 the loop-independent FLOPs of the prelude and coda (i.e., the boundary stages). Fixed-depth decoding requires $F_0 + r_{\max} F_r$ and adaptive-depth decoding $F_0 + r_t F_r$. For T generated tokens with mean exit depth $\bar{d} = \frac{1}{T}\sum_{t=1}^{T} r_t$, the FLOP-based upper bound on decode speedup is

$$S_{\text{dec}} = \frac{F_0 + r_{\max} F_r}{F_0 + \bar{d} F_r} \leq \frac{r_{\max}}{\bar{d}}. \tag{1}$$

For fully looped models whose boundary stages contain only the token embedding and LM head, $F_0 \ll F_r$ and the bound is close to $r_{\max}/\bar{d}$. Non-looped transformer layers increase the boundary-stage cost F_0, reducing the savings of exits. A larger $r_{\max}$ also makes each skipped loop save a smaller fraction of compute. Thus, model architecture has a substantial impact on the *available* speedup.

The speedup from adaptive-depth is often estimated from decode FLOPs alone, although prefill is unaffected and can add substantial latency. Since prefill latency depends on the workload and batching configuration, we measure its contribution post hoc. Let α denote the measured fraction of time spent on prefill. Adjusting the decode-only bound gives the end-to-end speedup estimate

$$S_{\text{e2e}} = \left(\alpha + (1-\alpha)/S_{\text{dec}}\right)^{-1}. \tag{2}$$

Recurrent batch size and latency. CDB refill aims to improve hardware utilization by keeping recurrent batches large, but its benefit depends on how latency scales with batch size. A recurrent step loads core weights once, while KV cache reads and matrix multiplications scale with batch size, so larger batches reduce the weight loading share. With N_b batch tokens and N_c context tokens (KV entries), we model the total recurrent step latency as $t(N_c, N_b) = \delta + \gamma\, N_b + \alpha\, N_c$, where δ is the fixed weight-reload cost, γN_b the core computation cost, and αN_c the KV cache streaming cost.

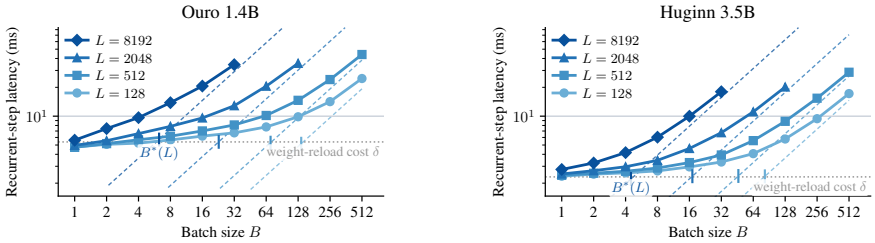

Figure 4: Recurrent step latency versus batch size at context lengths L for Ouro (left) and Huginn (right). Dotted lines show weight-reload cost δ, blue ticks mark the saturation batch size $B^*(L)$, and blue dashed lines show linear scaling in the compute bound regime. Below $B^*(L)$, shrinking a batch saves little time; above it, smaller sequential batches take roughly as long as one large batch.

Fig. 4 shows the recurrent step latency for the two models evaluated below, Ouro 1.4B and Huginn 3.5B. At small batches, weight loading dominates and latency remains nearly constant, making the step *memory bound* (Williams et al., 2009). At larger batches, batch-dependent computation and KV-cache streaming dominate, so latency grows approximately linearly with batch size. The *saturation batch size* B^* marks the transition between these regimes. B^* decreases with context length because longer contexts increase the KV-cache streaming cost. See Appendix B for details.

Thus, *refill works best in the memory-bound regime*. Here, early exits save FLOPs, but shrinking the batch provides little reduction in latency because weight loading still dominates. Refill fills the freed slots with new tokens at little additional latency. In the compute-bound regime, latency decreases as the batch shrinks, so no-refill already benefits from early exits and refill provides less improvement.

6 Experiments

We evaluate CDB in two settings. *Offline throughput* measures how fast the engine processes a fixed workload where all requests are known upfront, as in benchmarking or RL. This sustained load lets us test how much of the estimated maximum speedup CDB can realize and whether the recurrent-step latency model predicts well when refill helps. *Online serving* replays requests with dynamic arrivals, simulating user queries that arrive at different times, adding queueing effects and varying load.

6.1 Setup

We construct three request workloads with different prompt and output lengths: Alpaca (short instructions) (Taori et al., 2023) , ShareGPT (chat requests with similar average prompt and output lengths) (ShareGPT Team, 2023), and ArXiv-Summarization (long, prefill-heavy prompts) (Cohan et al., 2018). Varying prompt and output lengths tests CDB across different prefill costs, decode durations, and context lengths, all of which affect end-to-end speedup. We also vary each model's exit threshold to change how often tokens leave the recurrent core and open slots for refill. For each workload and exit threshold, we set the requests' output length to that of their reference responses and replay exit decisions recorded in advance from the model's exit policy (Appendix C).

We evaluate CDB on the two main looped LM architecture families, i.e., fully looped models and models with transformer layers in their prelude and coda. Ouro 1.4B (Zhu et al., 2025) is fully looped (0-24-0, $r_{\max}$=4): all its transformer layers are looping, with only the token embedding and LM head outside the recurrent core. Huginn 3.5B (Geiping et al., 2025) has a 2-4-2 prelude–core–coda split ($r_{\max}$=16), so 50% of its transformer layers are outside the core. This contrast tests how boundary stages affect the efficiency of depth-adaptive inference and which architecture design is better suited for it. Unless stated otherwise, we set the minimum coda batch size to K=1 for Ouro and K=$W/4$ for Huginn, where W is the maximum number of active sequences (e.g., K=16 at W=64).

We compare CDB using no-refill and refill against standard token-level CB, which runs $r_{\max}$ loops for every token. All schedulers use the same codebase, kernels, and admission policy, build on Hugging Face Transformers (Wolf et al., 2020), vLLM (Kwon et al., 2023), and SGLang (Zheng et al., 2024). All experiments run on a single 80 GB H100 GPU. See Appendix A for further details.

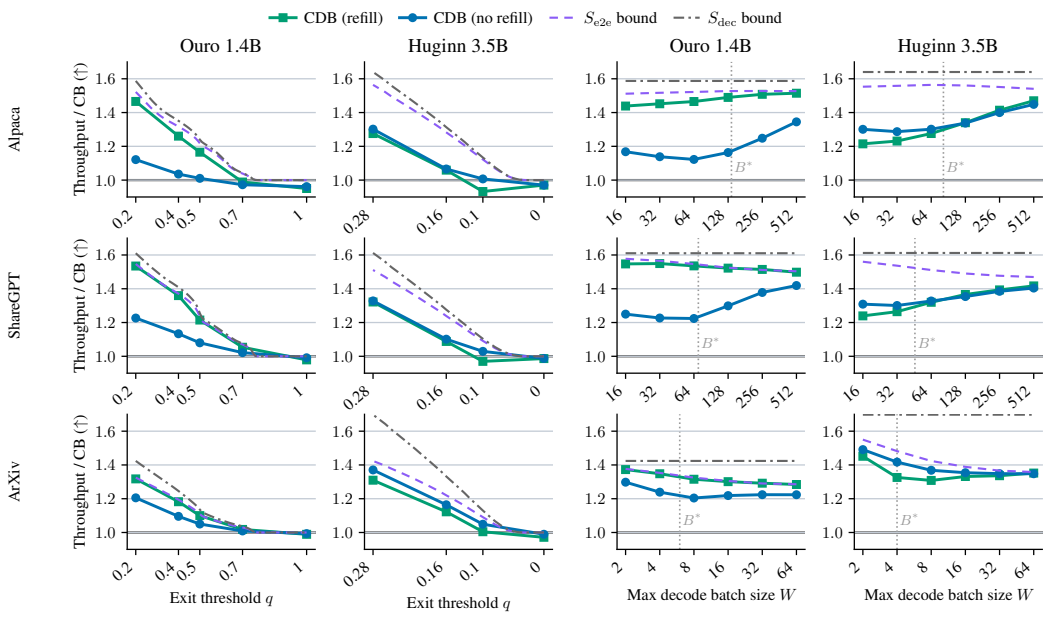

Figure 5: Offline throughput normalized to CB (full depth) for Alpaca (top), ShareGPT (middle), and ArXiv (bottom). *Left:* exit threshold sweep for a fixed maximum batch size of 64 (8 for ArXiv). *Right:* maximum batch size sweep at q=0.2 (Ouro) and q=0.28 (Huginn); dotted lines mark B^*.

6.2 Offline Throughput

We submit all requests at once and report generated tokens per second relative to full-depth CB. Fig. 5 (left) sweeps the exit threshold at a fixed maximum decode batch size of W=64 (W=8 for ArXiv), near the saturation batch size B^*. The right panels test different batch sizes by sweeping W for a fixed exit threshold. The dashed curves show the estimated upper bounds on speedup S_{e2e} and S_{dec}.

How close does CDB get to the maximum speedup? At the selected thresholds, refill achieves 1.32 to 1.53× the throughput of CB on Ouro and realizes 96 to 99% of S_{e2e}, nearly matching the estimated maximum for each setting. On Huginn, no-refill performs best, reaching 1.30 to 1.37× CB and 83 to 96% of S_{e2e}. At full depth, CDB has no early exits and therefore no opportunity to outperform CB. Even with its staged forward pass and additional scheduler interactions, our implementation retains 95 to 99% of CB throughput, showing that the additional scheduling introduces little overhead.

When is refill most effective? For Ouro, refill remains within 5% of S_{e2e} throughout all three batch-size sweeps and consistently outperforms no-refill. Its advantage is largest around or below B^*, where decoding is memory bound. The advantage narrows as larger batches become compute bound, matching the latency model in Section 5. In contrast, for Huginn, no-refill performs better at smaller widths, and the two modes converge as W increases.

How much does prefill reduce the available speedup? Decode FLOPs alone (S_{dec}) overestimate end-to-end speedup because depth-adaptive decoding does not reduce prefill latency. Across workloads, prefill accounts for 3.7–36% of GPU time. Its share is largest on long-context ArXiv and generally increases with decode batch size as decoding becomes more efficient. Consequently, S_{e2e} remains close to S_{dec} on Alpaca and ShareGPT and falls substantially below it on ArXiv. Decode still accounts for most latency, supporting our focus on decode-only early exit.

6.3 Online Serving

We replay requests with arrival times sampled from a Poisson process at rate λ for a 10-minute window and use a maximum decode batch size of W=64 (W=8 for ArXiv). We report normalized latency (end-to-end time divided by output length), which controls for varying response lengths (Kwon et al., 2023). Once the arrival rate exceeds the serving capacity, requests accumulate in the queue and

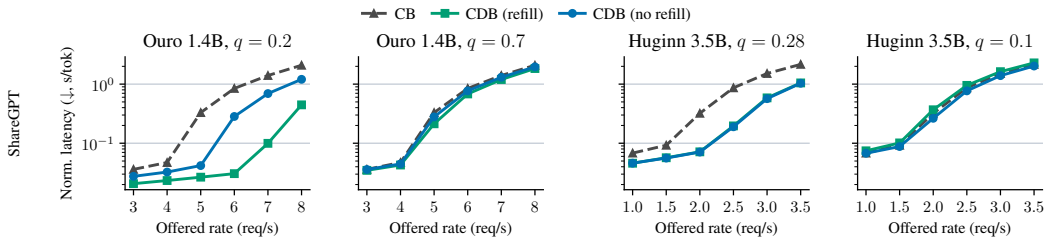

Figure 6: Normalized latency against request rate at two exit thresholds per model on ShareGPT. The first and third columns contain thresholds with many exits, and the other columns thresholds with few exits that leave little room for speedup. CB is threshold-independent as it always runs full depth.

normalized latency rises sharply. Fig. 6 shows the ShareGPT results at two exit thresholds per model. The complete results across workloads and exit thresholds are in Appendix D.

Do offline throughput gains translate to online serving? CDB sustains higher request rates before too many requests overload the system and cause normalized latency to rise, translating the offline throughput gains into greater serving capacity. For Ouro, refill provides the largest capacity gain, while no-refill adds only about half as much capacity over CB. For Huginn, refill and no-refill reach the same serving capacity, consistent with the small difference between them in offline throughput. At thresholds with fewer early exits, the curves approach CB and the capacity advantage disappears.

6.4 Ablations

CDB must decide when to interleave coda batches with recurrent-core steps. Running the coda immediately allows generated tokens to directly refill free core slots. Waiting for more exits improves coda efficiency at the cost of smaller recurrent batches. We vary the minimum coda batch size K to measure this trade-off across models, batch sizes, and number of boundary-stage layers (Fig. 7).

What is the best K? For Huginn, batching the coda improves throughput until the cost of delayed refill (i.e., recurrent-core batch size shrinks) outweighs the gain in coda efficiency (i.e., coda batch size grows). Across all decode batch sizes, $K{=}W/4$ best balances these effects. Ouro has cheap boundary stages, so keeping a large core batch is more important than coda efficiency, so immediate coda execution ($K{=}1$) performs best. At the largest decode batch sizes, the choice of K matters less because the stages become compute bound, reducing the importance of both coda batching and refill.

What if Huginn had smaller boundary stages? Removing transformer layers from Huginn's boundary stages makes immediate coda execution more competitive and improves throughput. For a fully looped 0-4-0 split, $K{=}1$ is near optimal, mirroring the behavior of Ouro. The gap to the maximum speedup also closes, confirming that boundary-stage cost is a central efficiency constraint.

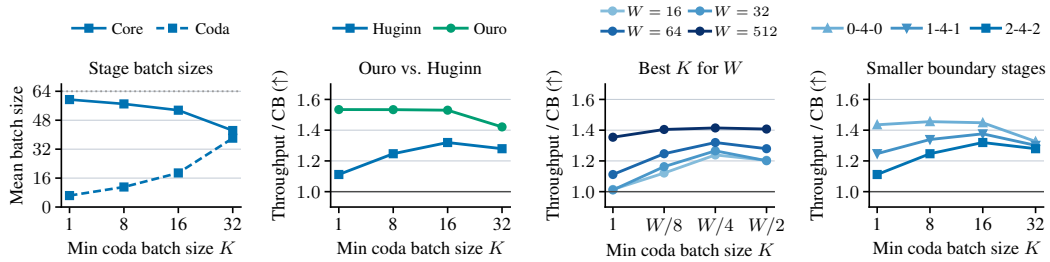

Figure 7: The effects of boundary-stage batching on ShareGPT. Blue colors consistently represent Huginn. *Left:* effect of K on mean batch size of core and coda on Huginn. *Center left:* effect of K on throughput for Ouro and Huginn. *Center right:* best K relative to maximum decode batch size W for Huginn. *Right:* effect of K on Huginn with different prelude-core-coda layer splits.

7 DISCUSSION

Does depth-adaptive decoding result in faster inference? CDB realizes up to 99% of the estimated maximum speedup available from the models' early-exit behavior. However, several conditions must hold to make inference meaningfully faster. First, more aggressive exit thresholds and missing KV states can reduce accuracy, so the model must be trained to provide enough early exits at the desired accuracy to create compute savings that CDB can use (Appendix E). Second, recurrent computation must account for a sufficient part of total inference time. For example, expensive prefill limits the end-to-end speedup, as our prefill-corrected bound shows. Third, the inference system must turn skipped recurrent computation into lower wall-clock time. In the memory-bound regime, shrinking the recurrent batch saves FLOPs but little time, making refill important. Additionally, updating batches between loop steps frequently stalls the GPU, so an asynchronous system is needed. Overall, our CDB implementation provides the inference method needed, while further inference speedups come from future models that allow more frequent exits at high accuracy.

Co-designing looped LMs and inference systems. Our results offer concrete guidance for designing looped models that benefit even more from depth-adaptive inference. Refilling batches between loops can substantially increase inference speed by replacing tokens as they exit. However, refill needs tokens waiting in the recurrent queue, and tokens only get there after running the boundary stages. If those stages are costly, they cannot run as often, leaving fewer tokens ready for refill. This suggests keeping *non-looped stages small* for efficient inference. However, adding layers outside the recurrent core can improve accuracy (Bae et al., 2025b; Geiping et al., 2025), creating a trade-off between accuracy and inference efficiency. Furthermore, *more loops* offer finer-grained control over exit depths, but each skipped loop then saves a smaller fraction of total recurrent compute (Section 5). Higher recurrence counts can also reduce accuracy at matched training compute (Schwethelm et al., 2026). The KV cache layout creates another co-design requirement. Early exits leave KV slots for skipped loops unwritten, requiring a policy for missing states. Reusing the last available state avoids recomputation, and looped models can often tolerate this without additional training, though support for KV sharing is not guaranteed. A *fully shared cache* further reduces the memory footprint and implementation complexity of CDB (Appendix E.1). Understanding when these layouts work, and designing models that reliably support them, is an important direction for future work.

Designing exit gates. Existing gates make exit decisions from the latest recurrent state, while asynchronous batch preparation requires knowing exits one loop in advance. Our lookahead gate provides this early decision without reducing accuracy, but constrains the minimum exit depth to $r_{\min}=2$ (Appendix E.2). For Ouro and Huginn, excluding exits after only one loop is consistent with their exit behavior, so the constraint does not reduce their realistic speedup. However, the constraint could reduce the available speedup of models that produce useful outputs after a single loop. One solution is to predict the complete exit depth before the token enters the recurrent core. However, making this decision reliable remains difficult (Bae et al., 2025b). A more promising alternative is to predict only the first loop-exit before the recurrent core and then use lookahead decisions for all later loops. This would allow $r_{\min}=1$ without requiring the complete exit depth to be known beforehand.

Limitations. We evaluate only two looped architectures on a single H100 GPU. Future looped models may introduce components such as mixture-of-experts that change the costs and scheduling requirements of recurrent and boundary stages. Our experiments also do not include production serving features such as multi-GPU execution, preemption, prefix caching, or speculative decoding. These techniques may shift the balance between prefill and decoding or introduce new interactions with depth-adaptive scheduling, so the relative benefit of CDB may differ in a broader serving system.

8 CONCLUSION

We introduced the first efficient inference method for depth-adaptive looped LMs. Our implementation of continuous depth batching (CDB) decomposes decode steps into separate stage queues, solving the batching problem posed by variable-depth forward passes. Our method is near-optimal and realizes most of the speedup available from early exits, showing that depth-adaptive decoding can be served efficiently. We also give new impulses for looped architecture design to improve the compute–accuracy trade-off, e.g., more accurate early exits and less expensive boundary stages. Overall, co-designing looped architectures and serving systems can lead to substantially faster inference.

AI USE STATEMENT

In this work, we used generative AI tools to help develop theoretical models and conceptual frameworks, formulate mathematical claims, propose or refine hypotheses, design and provide feedback on research methodology and experiments, implement methods, and interpret results. We have not used generative AI tools to generate synthetic datasets, assist with translation, clean or reformat datasets, or support qualitative and thematic data analysis. Providing critical ingredients for proving mathematical claims and assisting with the writing of proofs are not applicable to this work, which contains no formal proofs. Additionally, we used generative AI tools to create and modify scientific figures and other research artifacts, create and edit software code, draft and edit parts of the paper, search for and analyze relevant literature, brainstorm and identify research gaps, and provide feedback on the paper's structure and readability.

We have reviewed all AI-assisted work. LLM-generated code was reviewed and tested for correctness by two authors. All reported measurements were checked against the underlying experiment outputs, and all figures and tables were checked against their source data and generation scripts. References suggested by LLMs were manually verified against the original sources. AI-assisted research ideas were evaluated against the existing literature to check their originality and ensure appropriate attribution. The authors made all methodological and experimental decisions and critically evaluated the mathematical arguments and interpretation of results. The authors used LLM suggestions selectively and wrote the final manuscript themselves. We take responsibility for the final content of this work, including text, claims, code, and artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

For review, the supplementary material contains all implementation and experiment code required to reproduce the reported results, including the CDB variants, the CB baseline, and Bash scripts for reproducing every experiment and ablation reported in this paper. All evaluated model checkpoints and datasets are publicly available and cited in the experimental setup and Appendix C. Appendix A documents the implementation, system configuration, batching and admission parameters, scheduling algorithms, and execution details. Appendices B and C further document the measurement protocols and workload construction. We will release the code publicly upon publication.

REFERENCES

Amey Agrawal, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S. Gulavani, and Ramachandran Ramjee. SARATHI: Efficient LLM inference by piggybacking decodes with chunked prefills, 2023. URL https://arxiv.org/abs/2308.16369.

Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S. Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming throughput-latency tradeoff in LLM inference with Sarathi-Serve. In *Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation*, OSDI'24, USA, 2024. USENIX Association. ISBN 978-1-939133-40-3.

Quentin Anthony, Jacob Hatef, Deepak Narayanan, Stella Biderman, Stas Bekman, Junqi Yin, Aamir Shafi, Hari Subramoni, and Dhabaleswar Panda. The case for co-designing model architectures with hardware. In *Proceedings of the 53rd International Conference on Parallel Processing*, ICPP '24, pp. 84–96, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400717932. doi: 10.1145/3673038.3673136. URL https://doi.org/10.1145/3673038.3673136.

Sangmin Bae, Jongwoo Ko, Hwanjun Song, and Se-Young Yun. Fast and robust early-exiting framework for autoregressive language models with synchronized parallel decoding. In Houda Bouamor, Juan Pino, and Kalika Bali (eds.), *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 5910–5924, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.emnlp-main.362. URL https://aclanthology.org/2023.emnlp-main.362/.

Sangmin Bae, Adam Fisch, Hrayr Harutyunyan, Ziwei Ji, Seungyeon Kim, and Tal Schuster. Relaxed recursive transformers: Effective parameter sharing with layer-wise LoRA. In *The Thirteenth*

International Conference on Learning Representations, 2025a. URL https://openreview.net/forum?id=WwpYSOkkCt.

Sangmin Bae, Yujin Kim, Reza Bayat, Sungnyun Kim, Jiyoun Ha, Tal Schuster, Adam Fisch, Hrayr Harutyunyan, Ziwei Ji, Aaron Courville, and Se-Young Yun. Mixture-of-Recursions: Learning dynamic recursive depths for adaptive token-level computation. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025b. URL https://openreview.net/forum?id=QuqsEIVWIG.

Andrea Banino, Jan Balaguer, and Charles Blundell. PonderNet: Learning to ponder. In *8th ICML Workshop on Automated Machine Learning (AutoML)*, 2021. URL https://openreview.net/forum?id=1EuxRTe0WN.

Song Bian, Tao Yu, Shivaram Venkataraman, and Youngsuk Park. Scaling laws meet model architecture: Toward inference-efficient LLMs. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=0TmVqOpBbK.

Yanxi Chen, Xuchen Pan, Yaliang Li, Bolin Ding, and Jingren Zhou. EE-LLM: large-scale training and inference of early-exit large language models with 3D parallelism. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org, 2024.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems, 2021. URL https://arxiv.org/abs/2110.14168.

Arman Cohan, Franck Dernoncourt, Doo Soon Kim, Trung Bui, Seokhwan Kim, Walter Chang, and Nazli Goharian. A discourse-aware attention model for abstractive summarization of long documents. In Marilyn Walker, Heng Ji, and Amanda Stent (eds.), *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*, pp. 615–621, New Orleans, Louisiana, June 2018. Association for Computational Linguistics. doi: 10.18653/v1/N18-2097. URL https://aclanthology.org/N18-2097/.

Luciano Del Corro, Allie Del Giorno, Sahaj Agarwal, Bin Yu, Ahmed Awadallah, and Subhabrata Mukherjee. SkipDecode: Autoregressive skip decoding with batching and caching for efficient LLM inference, 2023. URL https://arxiv.org/abs/2307.02628.

Tri Dao, Daniel Y Fu, Stefano Ermon, Atri Rudra, and Christopher Re. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho (eds.), *Advances in Neural Information Processing Systems*, 2022. URL https://openreview.net/forum?id=H4DqfPSibmx.

DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Haowei Zhang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Li, Hui Qu, J. L. Cai, Jian Liang, Jianzhong Guo, Jiaqi Ni, Jiashi Li, Jiawei Wang, Jin Chen, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, Junxiao Song, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Lei Xu, Leyi Xia, Liang Zhao, Litong Wang, Liyue Zhang, Meng Li, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Ning Tian, Panpan Huang, Peiyi Wang, Peng Zhang, Qiancheng Wang, Qihao Zhu, Qinyu Chen, Qiushi Du, R. J. Chen, R. L. Jin, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, Runxin Xu, Ruoyu Zhang, Ruyi Chen, S. S. Li, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaoqing Wu, Shengfeng Ye, Shengfeng Ye, Shirong Ma, Shiyu Wang, Shuang Zhou, Shuiping Yu, Shunfeng Zhou, Shuting Pan, T. Wang, Tao Yun, Tian Pei, Tianyu Sun, W. L. Xiao, Wangding Zeng, Wanjia Zhao, Wei An, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, X. Q. Li, Xiangyue Jin, Xianzu Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaojin Shen, Xiaokang Chen, Xiaokang Zhang, Xiaosha Chen, Xiaotao Nie, Xiaowen Sun, Xiaoxiang Wang, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xingkai Yu, Xinnan Song, Xinxia Shan, Xinyi Zhou, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, Y. K. Li, Y. Q. Wang, Y. X. Wei, Y. X. Zhu, Yang Zhang, Yanhong Xu, Yanhong

Xu, Yanping Huang, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Li, Yaohui Wang, Yi Yu, Yi Zheng, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Ying Tang, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yu Wu, Yuan Ou, Yuchen Zhu, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yukun Zha, Yunfan Xiong, Yunxian Ma, Yuting Yan, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Z. F. Wu, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhen Huang, Zhen Zhang, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhibin Gou, Zhicheng Ma, Zhigang Yan, Zhihong Shao, Zhipeng Xu, Zhiyu Wu, Zhongyu Zhang, Zhuoshu Li, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Ziyi Gao, and Zizheng Pan. DeepSeek-V3 technical report, 2025. URL https://arxiv.org/abs/2412.19437.

Mostafa Dehghani, Stephan Gouws, Oriol Vinyals, Jakob Uszkoreit, and Lukasz Kaiser. Universal Transformers. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=HyzdRiR9Y7.

Maha Elbayad, Jiatao Gu, Edouard Grave, and Michael Auli. Depth-Adaptive Transformer. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=SJg7KhVKPH.

Mostafa Elhoushi, Akshat Shrivastava, Diana Liskovich, Basil Hosmer, Bram Wasti, Liangzhen Lai, Anas Mahmoud, Bilge Acun, Saurabh Agarwal, Ahmed Roman, Ahmed Aly, Beidi Chen, and Carole-Jean Wu. LayerSkip: Enabling early exit inference and self-speculative decoding. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 12622–12642, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.681. URL https://aclanthology.org/2024.acl-long.681/.

Markus Frey, Behzad Shomali, Ali Hamza Bashir, David Berghaus, Joachim Koehler, and Mehdi Ali. Adaptive loops and memory in transformers: Think harder or know more? In *Workshop on Latent & Implicit Thinking – Going Beyond CoT Reasoning*, 2026. URL https://openreview.net/forum?id=F87X9c107e.

Tianyu Fu, Yichen You, Zekai Chen, Guohao Dai, Huazhong Yang, and Yu Wang. Think-at-Hard: Dynamic looped transformers for improved reasoning. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=eQaJSRZiGn.

Jonas Geiping, Sean Michael 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. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025. URL https://openreview.net/forum?id=S3GhJooWIC.

Fabian Gloeckle, Badr Youbi Idrissi, Baptiste Roziere, David Lopez-Paz, and Gabriel Synnaeve. Better & faster large language models via multi-token prediction. In Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (eds.), *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 15706–15734. PMLR, 21–27 Jul 2024. URL https://proceedings.mlr.press/v235/gloeckle24a.html.

Alex Graves. Adaptive computation time for recurrent neural networks, 2017. URL https://arxiv.org/abs/1603.08983.

Ahmadreza Jeddi, Marco Ciccone, and Babak Taati. LoopFormer: Elastic-depth looped transformers for latent reasoning via shortcut modulation. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=RzYXb5YWBs.

Harsh Kohli, Srinivasan Parthasarathy, Huan Sun, and Yuekun Yao. Loop, think, & generalize: Implicit reasoning in recurrent-depth transformers. In *Third Conference on Language Modeling*, 2026. URL https://openreview.net/forum?id=8fz7WRThKL.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems*

Principles, SOSP '23, pp. 611–626, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400702297. doi: 10.1145/3600006.3613165. URL https://doi.org/10.1145/3600006.3613165.

Ryan Lee, Jacob Biloki, Edward J. Hu, and Jonathan May. Sparse layers are critical to scaling looped language models, 2026. URL https://arxiv.org/abs/2605.09165.

He Li, Feichen Song, Boyi Zeng, Shixiang Song, Zhiqin John Xu, Ziwei He, and Zhouhan Lin. PonderLM-3: Adaptive token-wise pondering with differentiable masking, 2026. URL https://arxiv.org/abs/2603.02023.

Xuting Liu, Daniel Alexander, Siva Kesava Reddy Kakarla, Behnaz Arzani, and Vincent Liu. Dynamic rebatching for efficient early-exit inference with DREX, 2025. URL https://arxiv.org/abs/2512.15705.

Amirkeivan Mohtashami, Matteo Pagliardini, and Martin Jaggi. CoTFormer: A chain of thought driven architecture with budget-adaptive computation cost at inference. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=7igPXQFupX.

Dhruv Nathawani, Shuoyang Ding, Vitaly Lavrukhin, Igor Gitman, Somshubra Majumdar, Evelina Bakhturina, Boris Ginsburg, and Jane Polak Scowcroft. Nemotron-Post-Training-Dataset-v2, August 2025. URL https://huggingface.co/datasets/nvidia/Nemotron-Post-Training-Dataset-v2.

NVIDIA. NVIDIA Nemotron Nano 2: An accurate and efficient hybrid Mamba-Transformer reasoning model, 2025. URL https://arxiv.org/abs/2508.14444.

Hayden Prairie, Zachary Novack, Taylor Berg-Kirkpatrick, and Daniel Y Fu. Parcae: Scaling laws for stable looped language models. In *Third Conference on Language Modeling*, 2026. URL https://openreview.net/forum?id=pIBqtrqFeP.

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.

Tal Schuster, Adam Fisch, Jai Gupta, Mostafa Dehghani, Dara Bahri, Vinh Q. Tran, Yi Tay, and Donald Metzler. Confident adaptive language modeling. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS '22, Red Hook, NY, USA, 2022. Curran Associates Inc. ISBN 9781713871088.

Kristian Schwethelm, Daniel Rueckert, and Georgios Kaissis. How much is one recurrence worth? Iso-Depth scaling laws for looped language models, 2026. URL https://arxiv.org/abs/2604.21106.

Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. FlashAttention-3: Fast and accurate attention with asynchrony and low-precision. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://openreview.net/forum?id=tVConYid20.

ShareGPT Team. ShareGPT, 2023. URL https://sharegpt.com.

Yingtao Shen and An Zou. River-LLM: Large language model seamless exit based on KV share, 2026. URL https://arxiv.org/abs/2604.18396.

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, 2026. URL https://arxiv.org/abs/2603.01914.

Lintang Sutawika, Hailey Schoelkopf, Leo Gao, Baber Abbasi, Stella Biderman, Jonathan Tow, ben fattori, Charles Lovering, farzanehnakhaee70, Jason Phang, Anish Thite, Fazz, Aflah, Niklas, Thomas Wang, sdtblck, nopperl, gakada, researcher2, tttyuntian, Julen Etxaniz, James A. Michaelov, Chris, Chessing234, Hanwool Albert Lee, Janna, Leonid Sinev, Khalid, Kiersten Stokes, and Zdeněk Kasner. EleutherAI/lm-evaluation-harness: v0.4.12, May 2026. URL https://doi.org/10.5281/zenodo.20122284.

Shawn Tan, Yikang Shen, Zhenfang Chen, Aaron Courville, and Chuang Gan. Sparse Universal Transformer. In *The 2023 Conference on Empirical Methods in Natural Language Processing*, 2023. URL https://openreview.net/forum?id=yXYJPAlLqn.

Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. Stanford Alpaca: An instruction-following LLaMA model, 2023. URL https://github.com/tatsu-lab/stanford_alpaca.

Victor Conchello Vendrell, Arnau Padres Masdemont, Niccolò Grillo, Jordi Ros-Giralt, Arash Behboodi, and Fabio Valerio Massoli. Memory-efficient looped transformer: Decoupling compute from memory in looped language models, 2026. URL https://arxiv.org/abs/2605.07721.

Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, April 2009. ISSN 0001-0782. doi: 10.1145/1498765.1498785. URL https://doi.org/10.1145/1498765.1498785.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander Rush. Transformers: State-of-the-art natural language processing. In Qun Liu and David Schlangen (eds.), *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pp. 38–45, Online, October 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.emnlp-demos.6. URL https://aclanthology.org/2020.emnlp-demos.6/.

Sangmin Yoo, Srikanth Malla, Chiho Choi, Wei D. Lu, and Joon Hee Choi. ADEPT: Adaptive dynamic early-exit process for transformers, 2026. URL https://arxiv.org/abs/2601.03700.

Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for Transformer-Based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pp. 521–538, Carlsbad, CA, July 2022. USENIX Association. ISBN 978-1-939133-28-1. URL https://www.usenix.org/conference/osdi22/presentation/yu.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. SGLang: Efficient execution of structured language model programs. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://openreview.net/forum?id=VqkAKQibpq.

Rui-Jie Zhu, Zixuan Wang, Kai Hua, Tianyu Zhang, Ziniu Li, Haoran Que, Boyi Wei, Zixin Wen, Fan Yin, He Xing, Lu Li, Jiajun Shi, Kaijing Ma, Shanda Li, Taylor Kergan, Andrew Smith, Xingwei Qu, Mude Hui, Bohong Wu, Qiyang Min, Hongzhi Huang, Xun Zhou, Wei Ye, Jiaheng Liu, Jian Yang, Yunfeng Shi, Chenghua Lin, Enduo Zhao, Tianle Cai, Ge Zhang, Wenhao Huang, Yoshua Bengio, and Jason Eshraghian. Scaling latent reasoning via looped language models, 2025. URL https://arxiv.org/abs/2510.25741.

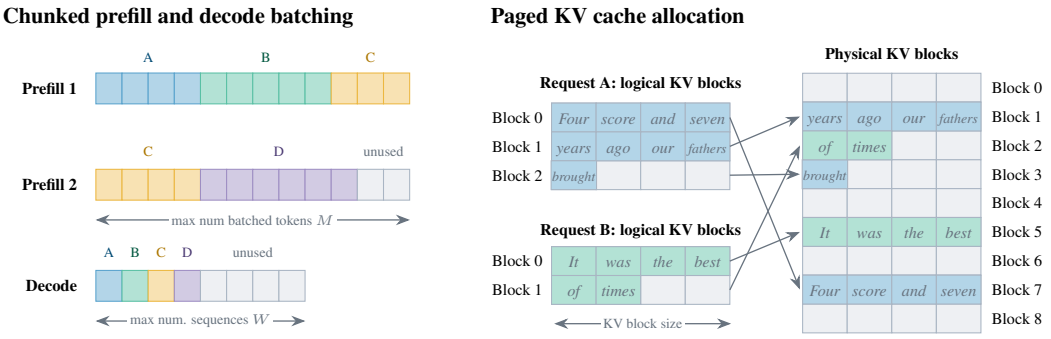

Figure 8: *Left:* prompt tokens are packed into prefill batches of at most M tokens, and each active sequence contributes one token to a decode batch of at most W tokens. Prompt C is split (chunked) across two prefill batches. *Right:* the logical KV blocks of each sequence map to non-contiguous blocks in the shared physical memory pool. Blocks are allocated as a sequence grows.

A Implementation Details

A.1 Inference System Overview

Engine design. Our CB and CDB implementations share the same basic inference design. Each sequence progresses from *prefill* to autoregressive *decode* and remains active until generation finishes or the scheduler preempts it. Following SGLang's default setting[1] (Zheng et al., 2024), the scheduler forms separate prefill and decode batches based on token, sequence, and KV cache memory limits. Another common design mixes prefill and decode tokens within the same batch ("piggybacking"), allowing ongoing generation to advance while new prompts are processed, which reduces long interruptions between decode steps (Agrawal et al., 2023; 2024). We use the phase-separated design to keep decode batches homogeneous, making the throughput effects of early exit easier to isolate. The CDB method itself does not require phase separation and could also be combined with piggybacking.

Chunked prefill. Generation begins with *prefill*, which processes the prompt and writes the resulting attention keys and values to the KV cache. Since all prompt tokens are known in advance, they can be processed in parallel. We pack the prompts of multiple requests into a prefill batch containing at most M tokens (Fig. 8, left). If a prompt does not fit into the remaining batch budget, we process the part that fits and continue the remaining tokens in the next prefill batch. This *chunked prefill* bounds the work of each prefill batch and prevents a single long prompt from occupying the GPU in one large forward pass. Once the final prompt token has been processed, its output produces the first generated token and the sequence enters decode. Decode is autoregressive: each generated token becomes the input for the next step, so every sequence advances by only one token at a time (Fig. 8, left).

Paged KV cache. During generation, each sequence continuously extends its KV cache with the keys and values of newly processed tokens, so its memory footprint grows. The caches of all active sequences share the limited GPU memory, which bounds the size of the active set and thus the achievable decode batch size. Output lengths are unknown in advance and memory requirements of sequences may differ substantially. Reserving space for the maximum possible length of every sequence would admit only a small active set and result in small decode batches. We use a *paged KV cache* (Kwon et al., 2023), which divides the cache into fixed-size blocks and assigns new blocks as a sequence grows (Fig. 8, right). This incremental allocation uses the available capacity efficiently, although the cache may still fill before generation finishes, which requires a preemption policy.

Preemption. The scheduler admits new requests only while sufficient cache capacity is available. If active sequences nevertheless exhaust the KV-cache capacity because they generate more tokens than expected, the scheduler can remove an active sequence by releasing its KV blocks and returning it to the waiting set. We implement two preemption policies. *Offloading* moves the KV state to CPU memory and restores it when the sequence resumes, preserving the values produced during decode. *Recomputation* discards the KV state and later reconstructs it by processing the prompt and

[1] SGLang v0.5.18.

Table 1: Inference configuration and parameters used in the main experiments.

Parameter	Setting
max num sequences W	64; $\{2, 4, 8, 16, 32, 64, 128, 256, 512\}$ in the batch-size sweep
max num batched tokens M	8192
min free slots S	$\max(1, \lfloor W/8 \rfloor)$
max context length	16,384 tokens
KV-cache capacity	20,480 blocks (327,680 token positions)
KV block size	16 tokens
KV-cache headroom ρ_{KV}	0%
preemption policy	disabled
KV-cache layout	shared cache
minimum coda batch size K (CDB refill)	1 for Ouro; $\lfloor W/4 \rfloor$ for Huginn
Numerical precision	`bfloat16`
Attention implementation	Paged FlashAttention-3
CUDA graphs	Prefill and decode

previously generated tokens again through prefill. Depth-adaptive decoding applies early-exit only during decode, while prefill always runs at full depth. Tokens that originally exited early are therefore processed again at full depth, so their reconstructed KV values may differ from those produced during decode. Although this repeats model computation, prefill processes the known tokens in parallel and is much faster per token than autoregressive decode, making cache reconstruction relatively efficient. Offloading preserves the exact KV state but incurs CPU memory transfers. We leave a detailed preemption analysis to future work. We disable preemption in the main experiments so that these costs and state changes do not affect the CB and CDB comparison.

A.2 Batching and Admission Parameters

Table 1 summarizes the configuration used in our main experiments, and the following paragraphs explain the parameters that control batching and admission.

Maximum batch sizes. GPU memory limits how many sequences and tokens can be batched together. Model parameters occupy a fixed part of this memory, while the remaining capacity is primarily shared between the KV cache and intermediate activations. We therefore use a sequence limit W (*max num sequences*) and a token limit M (*max num batched tokens*). The KV cache grows with the number and length of active sequences, so W limits how many sequences may retain KV state on the GPU. Since each active sequence contributes only one token during decode, W also defines the maximum decode batch size (Fig. 8, left). In Section 6.2, we vary W (and thus the decode batch size) to evaluate CDB in memory- and compute-bound regimes. During prefill, M limits the total number of prompt tokens batched together. Since $M > W$ in our experiments, the token limit does not further restrict decode batches. A larger M allows more prompt tokens to be processed together, which requires more activation memory and may interrupt decode for longer.

Prefill admission. The active set contains at most W sequences. When sequences finish, they free slots that can be filled by new sequences. Because prefill and decode run in separate batches, filling these slots requires a prefill batch, which temporarily pauses decode. Refilling each slot immediately can produce many prefill batches containing only one or a few prompts. However, prefill uses the GPU efficiently once a batch contains enough tokens to become compute-bound. A sufficiently long prompt can reach this regime by itself, so waiting to combine it with other prompts gives little throughput benefit. Short prompts may not contain enough tokens, making it more efficient to wait for several free slots to combine more prompts. For example, the short Alpaca prompts generally require more sequences to form a compute-bound prefill batch than the longer ShareGPT prompts. The *min free slots* parameter S makes the scheduler wait until a chosen number of slots are free before interrupting decode for prefill. A low value can cause frequent and inefficient prefill interruptions, while a high value makes decode less efficient as the decode batch shrinks with less active sequences.

KV-cache headroom. The memory footprint of the KV cache depends on the prompt and generated lengths of each active sequence. The maximum number of sequences W therefore provides only coarse control over memory use. The *KV-cache headroom* ρ_{KV} specifies how much of the cache should remain free so that active sequences can add new KV entries as they generate new tokens.

When the free capacity falls below this value, the scheduler stops admitting new sequences and continues decoding the active set. For example, two sequences with very long prompts may already leave less free cache than the configured headroom, even if $W{=}64$. The scheduler then keeps the active set at two until one finishes and releases its KV state. The KV-cache headroom therefore adapts the active-set size to its actual memory use, while W provides a fixed upper bound.

A.3 SCHEDULING ALGORITHMS

Algorithms 1 and 2 show pseudocode for CDB's no-refill and refill mode, respectively. In both algorithms, CANPREFILL$(W, S, \rho_{\mathrm{KV}})$ denotes the prefill admission rule described in Appendix A.2. GATE represents the early-exit gate, which includes checking if a token has reached $r_{\max}$. The algorithms focus on batch formation and stage ordering and omit asynchronous execution.

Algorithm 1 *No-refill* CDB follows the same phase-separated prefill and decode batching as CB. The main difference is that during decode, early exits shrink the recurrent batch without replacement. Exited tokens wait until the entire cohort runs the coda together.

Require: prompt queue Q_{pre}, decode queue Q_{dec}, max num batched tokens M, max num sequences W, min free slots S, KV-cache headroom ρ_{KV}, maximum recurrent depth $r_{\max}$

loop
  while CANPREFILL$(W, S, \rho_{\mathrm{KV}})$ **do**
    $B_{\mathrm{pre}} \leftarrow$ pack up to M prompt tokens from Q_{pre}
    $x_{\mathrm{new}} \leftarrow$ first generated tokens from FULLDEPTHFORWARD(B_{pre})
    push x_{new} onto Q_{dec}
  end while
  $B_{\mathrm{dec}} \leftarrow Q_{\mathrm{dec}}$
  $H \leftarrow$ PRELUDE(B_{dec})
  $E \leftarrow \emptyset$
  for $r = 1, \ldots, r_{\max}$ **do**
    $H \leftarrow$ CORE(H)
    $H_{\mathrm{exit}} \leftarrow$ GATE(H, r)
    $E \leftarrow E \cup H_{\mathrm{exit}}$; $H \leftarrow H \setminus H_{\mathrm{exit}}$ ▷ Shrink recurrent batch
    if $H = \emptyset$ **then**
      break ▷ All tokens have exited
    end if
  end for
  $x_{\mathrm{next}} \leftarrow$ CODA(E)
  $Q_{\mathrm{dec}} \leftarrow$ tokens in x_{next} belonging to unfinished sequences
end loop

Algorithm 2 *Refill* CDB schedules prefill, recurrent core, and coda independently using the priority rules described in Section 4.1. Prelude always runs directly after prefill or coda. Tokens that continue looping return to Q_R, while exited tokens go to Q_C. When $K{=}1$, coda runs immediately after every loop exit, allowing the successor tokens to refill Q_R before the next recurrent core batch.

Require: prompt queue Q_{pre}, recurrent queue Q_R, coda queue Q_C, max num batched tokens M, max num sequences W, min free slots S, KV-cache headroom ρ_{KV}, minimum coda batch size K, maximum recurrent depth $r_{\max}$

loop
  if $|Q_C| \geq K$ **then**
    $E \leftarrow Q_C$; $\quad Q_C \leftarrow \emptyset$
    $x_{\text{next}} \leftarrow \textsc{Coda}(E)$
    $H_{\text{new}} \leftarrow \textsc{Prelude}(x_{\text{next}})$ for unfinished sequences
    push H_{new} onto Q_R
  else if $\textsc{CanPrefill}(W, S, \rho_{\text{KV}})$ **then**
    $B_{\text{pre}} \leftarrow$ pack up to M prompt tokens from Q_{pre}
    $x_{\text{new}} \leftarrow$ first generated tokens from $\textsc{FullDepthForward}(B_{\text{pre}})$
    push $\textsc{Prelude}(x_{\text{new}})$ onto Q_R for unfinished sequences
  else if $Q_R \neq \emptyset$ **then**
    $H \leftarrow Q_R$; $\quad Q_R \leftarrow \emptyset$
    $H \leftarrow \textsc{Core}(H)$
    $H_{\text{exit}} \leftarrow \textsc{Gate}(H, r_{\max})$
    push H_{exit} onto Q_C
    push $H \setminus H_{\text{exit}}$ onto Q_R
  end if
end loop

A.4 Scheduling Examples

Tables 2 and 3 show concrete traces of the scheduling algorithms above. Each row represents one scheduled stage's batch, where B gives the number of sequences processed by the stage. For decode stages (prelude, coda, loop) B corresponds to the batch size, while for prefill the detail column reports the number of batched prompt tokens and thus the prefill batch size. The queue columns report the queue sizes *after* the stage. $\mathrm{r}_i{:}n$ in the detail column of loop stages denotes that n tokens execute their i-th loop step. Green boxes highlight the recurrent-core batch sizes.

The traces illustrate how the two modes order the model stages. With refill, each prefill or coda batch is followed by a prelude, which places the resulting decode tokens directly into Q_R. No-refill instead waits for all prefill steps to finish before starting the decode phase. With $K{=}1$, coda and prelude run whenever tokens exit, so successor tokens from unfinished sequences can fill the freed recurrent slots before the next loop batch. Consequently, the recurrent batch remains at its maximum size $W{=}64$ throughout the shown refill traces. Refill can combine tokens at different loop steps into one forward pass because every step loads the same recurrent-core weights. For example, stage 14 of the Ouro refill trace combines tokens at their first, second, and fourth loop steps. Also the shared KV-cache layout makes this easy as every loop step writes and reads to the same KV slot. No-refill waits until every token has exited, so its recurrent batch only shrinks. For $K{>}1$, exits accumulate in Q_C and the recurrent batch temporarily shrinks. A lower K runs coda more frequently and refills the freed slots sooner, trading off a higher recurrent batch size with more coda executions.

Table 2: The first 20 stages of refill CDB on ShareGPT with $W=64$ and $K=1$, using Ouro (left) and Huginn (right).

OURO: REFILL ($K=1$)

	Stage	B	Detail	Q_R	Q_C	Q_P
0	init			0	0	256
1	prefill	21	8192 prompt tokens	0	0	235
2	prelude	20	after prefill	20	0	235
3	prefill	33	8192 prompt tokens	20	0	203
4	prelude	32	after prefill	52	0	203
5	prefill	12	1998 prompt tokens	52	0	192
6	prelude	12	after prefill	64	0	192
7	loop	64	r_1:64	64	0	192
8	loop	64	r_2:64; 1 exit	63	1	192
9	coda	1		63	0	192
10	prelude	1	after coda	64	0	192
11	loop	64	r_1:1 r_3:63; 29 exits	35	29	192
12	coda	29		35	0	192
13	prelude	29	after coda	64	0	192
14	loop	64	r_1:29 r_2:1 r_4:34; 35 exits	29	35	192
15	coda	35		29	0	192
16	prelude	35	after coda	64	0	192
17	loop	64	r_1:35 r_2:29; 9 exits	55	9	192
18	coda	9		55	0	192
19	prelude	9	after coda	64	0	192
20	loop	64	r_1:9 r_2:35 r_3:20; 21 exits	43	21	192

HUGINN: REFILL ($K=1$)

	Stage	B	Detail	Q_R	Q_C	Q_P
0	init			0	0	256
1	prefill	21	8192 prompt tokens	0	0	235
2	prelude	20	after prefill	20	0	235
3	prefill	34	8192 prompt tokens	20	0	202
4	prelude	33	after prefill	53	0	202
5	prefill	11	1785 prompt tokens	53	0	192
6	prelude	11	after prefill	64	0	192
7	loop	64	r_1:64	64	0	192
8	loop	64	r_2:64	64	0	192
9	loop	64	r_3:64	64	0	192
10	loop	64	r_4:64	64	0	192
11	loop	64	r_5:64	64	0	192
12	loop	64	r_6:64	64	0	192
13	loop	64	r_7:64; 7 exits	57	7	192
14	coda	7		57	0	192
15	prelude	7	after coda	64	0	192
16	loop	64	r_1:7 r_8:57; 22 exits	42	22	192
17	coda	22		42	0	192
18	prelude	22	after coda	64	0	192
19	loop	64	r_1:22 r_2:7 r_9:35; 18 exits	46	18	192
20	coda	18		46	0	192

Table 3: The first 20 stages for Ouro on ShareGPT with $W=64$, using no-refill (left) and refill with $K=16$ (right).

OURO: NO-REFILL

	Stage	B	Detail	Q_D	Q_P
0	init			0	256
1	prefill	21	8192 prompt tokens	20	235
2	prefill	33	8192 prompt tokens	52	203
3	prefill	12	1998 prompt tokens	64	192
4	prelude	64		0	192
5	loop	64	r_1:64	0	192
6	loop	64	r_2:64; 1 exit	0	192
7	loop	63	r_3:63; 29 exits	0	192
8	loop	34	r_4:34; 34 exits	0	192
9	coda	64		64	192
10	prelude	64		0	192
11	loop	64	r_1:64	0	192
12	loop	64	r_2:64; 13 exits	0	192
13	loop	51	r_3:51; 39 exits	0	192
14	loop	12	r_4:12; 12 exits	0	192
15	coda	64		64	192
16	prelude	64		0	192
17	loop	64	r_1:64	0	192
18	loop	64	r_2:64; 17 exits	0	192
19	loop	47	r_3:47; 39 exits	0	192
20	loop	8	r_4:8; 8 exits	0	192

OURO: REFILL ($K=16$)

	Stage	B	Detail	Q_R	Q_C	Q_P
0	init			0	0	256
1	prefill	21	8192 prompt tokens	0	0	235
2	prelude	20	after prefill	20	0	235
3	prefill	33	8192 prompt tokens	20	0	203
4	prelude	32	after prefill	52	0	203
5	prefill	12	1998 prompt tokens	52	0	192
6	prelude	12	after prefill	64	0	192
7	loop	64	r_1:64	64	0	192
8	loop	64	r_2:64; 1 exit	63	1	192
9	loop	63	r_3:63; 29 exits	34	30	192
10	coda	30	$\lvert Q_C \rvert \geq 16$	34	0	192
11	prelude	30	after coda	64	0	192
12	loop	64	r_1:30 r_4:34; 34 exits	30	34	192
13	coda	34	$\lvert Q_C \rvert \geq 16$	30	0	192
14	prelude	34	after coda	64	0	192
15	loop	64	r_1:34 r_2:30; 10 exits	54	10	192
16	loop	54	r_2:34 r_3:20; 20 exits	34	30	192
17	coda	30	$\lvert Q_C \rvert \geq 16$	34	0	192
18	prelude	30	after coda	64	0	192
19	loop	64	r_1:30 r_3:31 r_4:3; 25 exits	39	25	192
20	coda	25	$\lvert Q_C \rvert \geq 16$	39	0	192

A.5 EFFICIENT EXECUTION

CUDA graphs and static shapes. A model forward pass consists of many GPU operations, such as matrix multiplications, normalization, and attention. Many of these operations are executed by separate GPU kernels that the CPU normally launches individually. The small overhead of each launch accumulates across layers and sequences. We reduce this launch overhead with *CUDA graphs*, which record a sequence of GPU operations once and replay it with little CPU involvement. The recorded operations, tensor shapes, and tensor memory addresses must remain fixed between replays, only the values stored in the tensors may change. We therefore preallocate reusable input and output buffers whose contents are updated for each batch. To support different batch sizes, we pad batches to a set of supported buckets and capture the corresponding graphs during warm-up. For CB, each graph captures the complete decode forward pass. For CDB, we capture separate graphs for the

prelude, recurrent core, and coda because the scheduler executes these stages independently. Both engines capture prefill separately from decode. In prefill, the attention operation depends on how prompt tokens are divided among sequences, which determines which tokens attend to one another. Capturing a separate compute graph for every possible combination of prompt lengths would be infeasible. We therefore break the prefill graph at each attention operation following SGLang.

Attention and output projection. We use FlashAttention-3 for both prefill and decode (Shah et al., 2024). Prefill uses its variable-length (*varlen*) path, and decode uses its *fast path* for paged KV caches. The output projection over the vocabulary can also be expensive for large prefill batches. Only the final token of a completed prompt produces a generated token, so we compute output logits only for these positions instead of for every prompt token.

B Recurrent-Step Latency Details

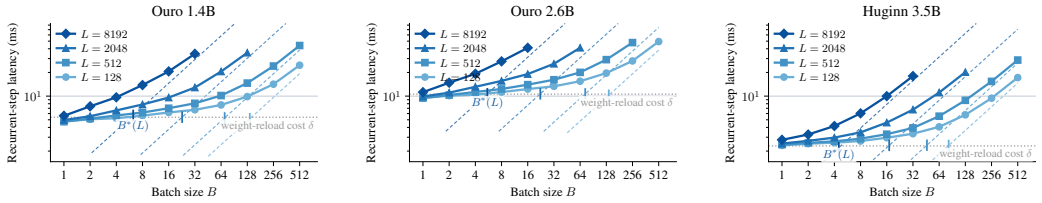

Figure 9: Decode latency of a recurrent step versus batch size at context lengths L, for Ouro 1.4B (left), Ouro 2.6B (center), and Huginn 3.5B (right). The dotted line shows the weight-reload cost δ, with blue ticks marking the saturation batch size $B^*(L)$. The blue dashed lines show compute-bound scaling, where latency grows linearly.

We measure decode latency of a single recurrent step on an 80 GB H100 GPU through our CB implementation with CUDA graphs and paged FlashAttention-3 (Kwon et al., 2023; Shah et al., 2024). For each batch size B and context length L, we submit B equal-length requests at full depth (no early exit). We set up the CB scheduler to complete prefill before any decode begins, so all B requests start decoding together from context length L and the decode batch size is exactly B.

For each (B, L), we measure the wall-clock time τ of a full decode step by averaging over 16 consecutive token generations. We then separate the fixed boundary cost from the per-step cost by varying r. To isolate the recurrent-step latency, we vary $r \in \{1, 2, 3, 4\}$ for Ouro and $r \in \{2, 4, 8, 12, 16\}$ for Huginn and fit

$$\tau(r) = \tau_0 + \tau_r \cdot r, \tag{3}$$

where τ_r is the recurrent-step latency, and τ_0 is the prelude and coda overhead. For Ouro, $\tau_0 \approx 0.4$ ms across all operating points (embedding and LM head only). For Huginn, τ_0 ranges from 2.8 ms at short context ($L = 128$) to 3.3 ms at $L = 8192$ (due to transformer layers in prelude and coda), about $1.2\tau_r$. The fit is essentially perfect ($R^2 \geq 0.999$), confirming that decode latency is linear in depth.

We then fit the recurrent-step latency τ_r with the latency model of Section 5, $t(N_c, N_b) = \delta + \gamma\, N_b + \alpha\, N_c$. At a fixed context length L, each of the $N_b = B$ batch tokens reads L KV entries, so $N_c = BL$ and the model reduces to

$$t(B) = \delta + (\gamma + \alpha L)\, B. \tag{4}$$

The intercept δ is the context-independent weight reload cost, γB (mainly) the cost from matrix multiplications, and αLB the KV cache streaming cost. The saturation batch size is $B^* = \delta/(\gamma + \alpha L)$, which falls with L as the KV-cache streaming cost αL grows.

Fig. 9 shows the fitted curves including the larger Ouro model with 2.6B parameters. The weight-reload cost δ depends on layer shape (Anthony et al., 2024). Huginn's recurrent core has 1.6B parameters in four wide layers, while Ouro 1.4B has 1.2B in 24 narrow layers. Huginn's $\delta \approx 2.3$ ms is less than half of Ouro 1.4B's $\delta \approx 5.4$ ms. Ouro 2.6B doubles the layer count to 48, which approximately doubles both δ and the slopes, leaving their ratio B^* nearly unchanged. For Ouro 1.4B, B^* drops from about 140 at $L = 128$ to single digits at $L = 8192$. Huginn follows a similar trajectory from about 80. Because the measured curves approach the compute-bound asymptote

from below without fully reaching it at the largest batch that fits in memory, the fitted slopes mildly underestimate the true per-token cost, so these B^* values are mild overestimates.

C Benchmark Workloads

C.1 Input and Output Tokens

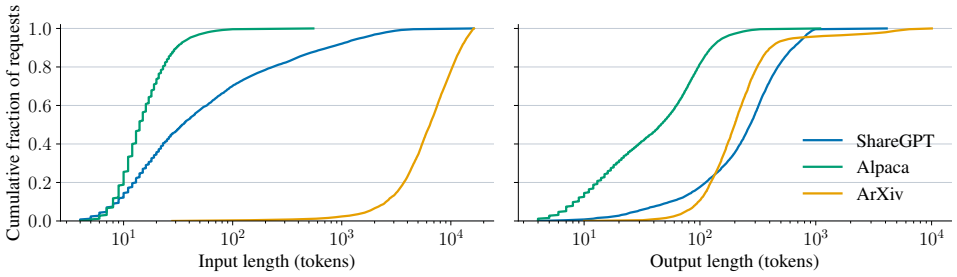

Figure 10: Empirical CDFs of prompt (left) and output (right) token lengths for the three workloads under the Ouro tokenizer.

In our experiments, we do not let the models generate their own responses, as they may deviate between runs. Instead, we take prompt and reference-output lengths from the ShareGPT (ShareGPT Team, 2023), Alpaca (Taori et al., 2023), and ArXiv-Summarization (Cohan et al., 2018) datasets. From ShareGPT we take one request per conversation by using the first user-assistant pair, so each request is single-turn. For Alpaca we pair each instruction with its reference output, and for ArXiv we do the same for each article and its corresponding abstract. We tokenize every prompt and reference output with each model's tokenizer, drop pairs with fewer than four prompt or output tokens, and drop requests whose combined length exceeds the context length of 16,384 tokens. All tested models replay the same requests in the same order. Only small differences in token lengths arise from their different tokenizers.

This leaves 56,067 ShareGPT, 49,443 Alpaca, and 11,899 ArXiv requests. The offline threshold sweeps replay 6,000 ShareGPT, 10,000 Alpaca, and 1,000 ArXiv requests. The reported throughput is stable at these sample sizes, and running more requests would only increase experiment duration. For the serving experiments, each point replays a fixed 10-minute Poisson arrival trace, so its request count is determined by the offered request rate.

Fig. 10 shows the resulting input and output token length distributions under the Ouro tokenizer (Zhu et al., 2025). Alpaca requests are short and tightly concentrated (mean prompt and output lengths of 18 and 60 tokens), whereas ShareGPT requests are long and heavy-tailed (mean 267 and 311 tokens, with a 99th prompt-length percentile of 3264 tokens against Alpaca's 75). ArXiv provides a prefill-heavy workload: its mean prompt and output lengths are 6,989 and 370 tokens, with medians of 6,436 and 199 tokens. Its prompts are approximately 26× longer than ShareGPT's on average, while its outputs are only 1.2× longer. ShareGPT prompts and outputs are in turn on average 14.8× and 5.2× longer than Alpaca's.

C.2 Early-exit Distribution

Similar to the output tokens, we do not use an active exit gate during our experiments. To make exit decisions deterministic, we run all requests through the models once in advance and record their per-token exits, which are then replayed during the experiments. Fig. 11 shows the resulting exit-depth distributions for different gate thresholds for Ouro and Huginn. For Ouro, a higher gate threshold makes fewer tokens exit early; for Huginn, a lower convergence threshold has the same effect. Across all three workloads, the exit-depth distributions remain similar at a given threshold, with only modest workload-dependent shifts. Huginn's convergence gate spreads exits over a wider range of depths than Ouro's gate.

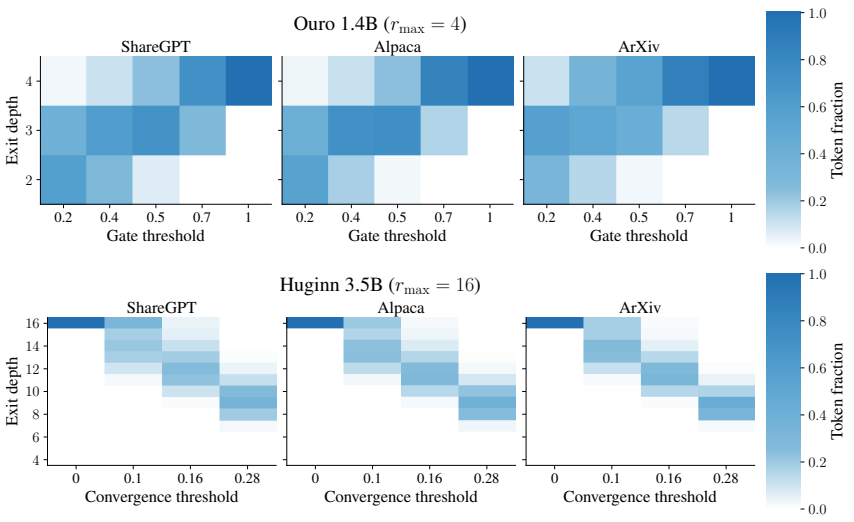

Figure 11: Exit-depth distributions for different gate thresholds. Each cell shows the fraction of tokens exiting at a given depth.

D Complete Online Serving Results

Figs. 12 and 13 extend the ShareGPT comparison of Section 6.3 to Alpaca and ArXiv and more exit thresholds. The general observations hold for all workloads.

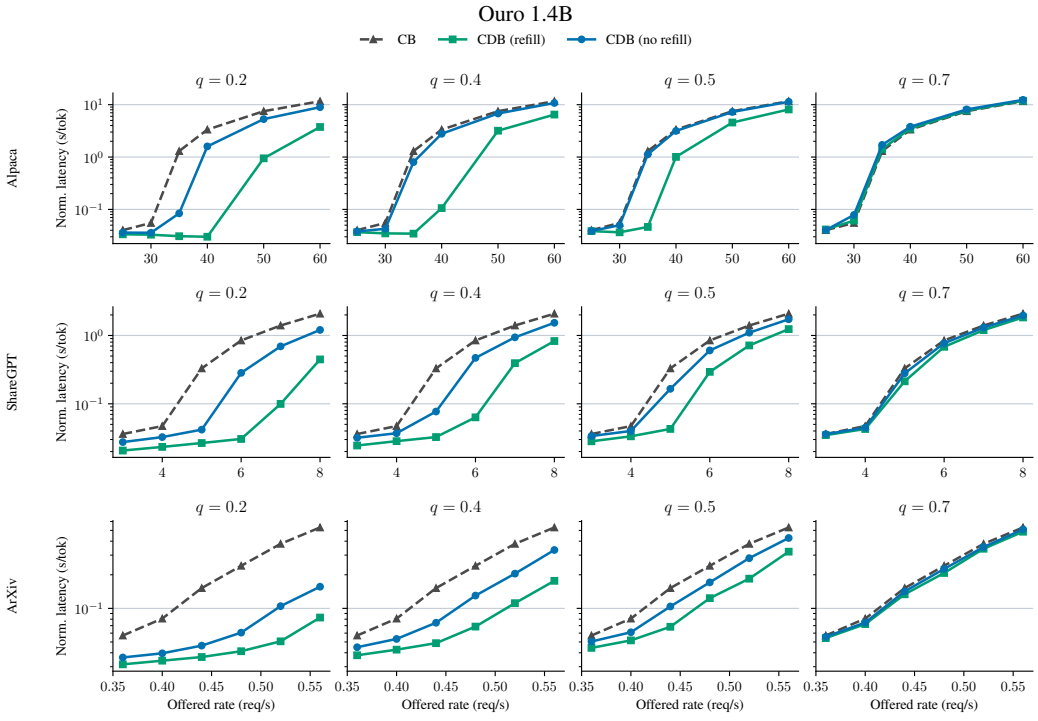

Figure 12: Normalized latency against request rate across all exit thresholds for Ouro on Alpaca, ShareGPT, and ArXiv. Columns run from aggressive (left) to conservative (right) thresholds. CB is threshold-independent (full depth).

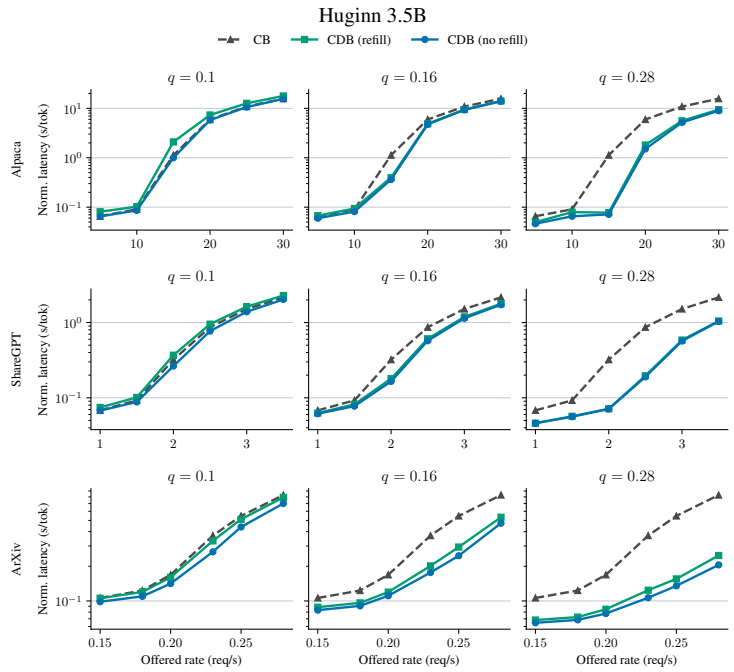

Figure 13: Same as Fig. 12 for Huginn, whose exit criterion is inverted: lower thresholds are more conservative, so the columns run from conservative (left) to aggressive (right).

E Accuracy-Efficiency Trade-off Ablations

Our work is about maximizing the inference efficiency of looped LMs. This section evaluates how well current models support the techniques we build on. We reimplement the `gsm8k_cot` task configuration of the LM Evaluation Harness (Cobbe et al., 2021; Sutawika et al., 2026) on top of our continuous-batching and continuous-depth-batching implementations, and evaluate Ouro 1.4B (Zhu et al., 2025) and Huginn 3.5B (Geiping et al., 2025). Two choices in our setup affect task accuracy: the KV cache policy and the exit gate, both trading off accuracy against throughput and memory.

E.1 KV Cache Sharing

Looped LMs share physical attention layers across iterations, so multiple iterations can also share KV cache slots. Neither model was trained for this setting, so KV sharing is a zero-shot capability here. We test how KV cache sharing affects accuracy and memory, evaluating both models at fixed depth. We compare a *full depth-indexed cache*, where each loop iteration writes to its own KV slot, against two shared layouts: *single-slot*, where every iteration overwrites one shared KV slot per physical layer, and *first-then-shared*, where the first iteration retains a private slot while all subsequent ones share a second. Writing s for the number of KV slots per recurrent layer, the three layouts use $s = r_{\max}$, $s = 1$, and $s = 2$, respectively. The total cache thus holds $\ell_0 + \ell_r s$ KV entries per token, where ℓ_0 denotes the number of attention layers outside the loop (prelude and coda) and ℓ_r those inside the recurrent core. Sharing therefore saves a factor of $\frac{\ell_0+\ell_r r_{\max}}{\ell_0+\ell_r s}$ over the depth-indexed cache.

The results in Table 4 show that the Huginn model is insensitive to the KV sharing layout, while Ouro loses accuracy under both layouts, from about 6 points under first-then-shared to a complete collapse under single-slot sharing. We cannot reproduce the results from the Ouro paper, which reports near-lossless KV sharing (Zhu et al., 2025). This might be because they share only during decoding and keep a depth-indexed cache for the prompt. Still, they report a degradation of about 10 accuracy points once the sharing extends to prefill, which is far less than the drop we observe.

Our results are in line with Vendrell et al. (2026), who propose integrating KV sharing into Ouro's training as a solution, which then recovers most of the gap.

Surprisingly, Huginn actually performs better under single-slot sharing. Under a truncated loop (i.e., reducing $r_{\max}$), single-slot sharing retains substantially more accuracy than the full depth-indexed cache: 11.45% against 3.71% at depth 6 and 25.25% against 12.43% at depth 9, converging to within about two points at depth 16 (Table 5).

Overall, sensitivity to KV cache sharing depends on the model, and decreasing it should be a development target, as KV sharing enables not only much simpler CDB (Section 4.2) but also substantial memory savings: single-slot sharing shrinks the cache $4\times$ for Ouro and $8.5\times$ for Huginn.

Table 4: GSM8K accuracy (flexible match, greedy decoding) and KV cache footprint under different recurrent-depth KV cache layouts. KV slots are counted per physical layer and token. Cache sizes assume `bfloat16` and are reported per token in KiB and for one full 4096-token sequence in GiB.

	Ouro ($r = 4$)				Huginn ($r = 16$)			
KV cache layout	Slots s	KiB/tok	GiB@4k	Accuracy	Slots s	KiB/tok	GiB@4k	Accuracy
Full depth-indexed	4	768	3.00	78.09%	16	1402	5.48	30.93%
Single-slot	1	192	0.75	0.23%[†]	1	165	0.64	32.52%
First-then-shared	2	384	1.50	72.40%	2	248	0.97	32.83%

[†] Zhu et al. (2025) report 78.85% for this setting, while Vendrell et al. (2026) find that Ouro requires specialized fine-tuning to perform well with single slot KV cache sharing.

Table 5: Huginn 3.5B GSM8K accuracy (flexible match, greedy decoding) under different recurrent-depth KV cache layouts and truncated loops.

	Fixed recurrent depth r							
KV cache layout	6	7	8	9	10	12	14	16
Full depth-indexed	3.71%	4.85%	8.19%	12.43%	19.56%	27.07%	27.37%	30.93%
Single shared	11.45%	15.85%	20.39%	25.25%	28.51%	29.80%	30.78%	32.52%
First-then-shared	11.45%	14.78%	20.02%	24.79%	27.82%	30.25%	30.71%	32.83%

E.2 Early-exit Gating

Looped LMs naturally support test-time compute scaling by varying the loop depth. The simplest strategy is to reduce the fixed depth uniformly for all tokens together. In contrast, depth-adaptive decoding can allocate compute per token, looping less for "easy" tokens and more for "hard" ones. We evaluate whether early-exit gating outperforms fixed-depth decoding, and whether the delayed exit decision required by our asynchronous CDB scheduler (Section 4.3) degrades accuracy.

Setup. Ouro includes a trained exit gate, which we use as the baseline. Since our asynchronous CDB scheduler requires the exit decision one step before the actual exit (Section 4.3), we train two additional exit gates on the frozen model: a *lookahead gate*, which predicts the exit one step in advance, and a *pre-loop gate*, which predicts the exit before the loop starts. Both are trained via gate distillation on Nemotron-Post-Training-Dataset-v2 (Nathawani et al., 2025; NVIDIA, 2025). Huginn includes no gate, so we adopt the training-free convergence criterion of Geiping et al. (2025), which exits once the normalized L2 distance between consecutive loop states falls below a threshold. The threshold direction differs: for Huginn a larger value triggers more exits, for Ouro a smaller one. For Ouro, we set $r_{\min}$=2, as the model is not trained to produce useful outputs after a single iteration (Zhu et al., 2025). For Huginn, we set $r_{\max}$=16 rather than the trained 32, since accuracy is flat above 16, and use the shared KV cache layout (single-slot) for better performance (Appendix E.1).

Results. Fig. 14 summarizes the results. Depth-adaptive decoding Pareto-dominates fixed depth, reaching higher accuracy at every mean recurrent depth. Ouro performs well down to a mean depth of ≈ 2.5 and Huginn down to ≈ 8. The exit distributions in the right panel show that Ouro spreads tokens over three of its four depths and Huginn over four to seven of its sixteen.

Regarding delayed exits, the trained lookahead gate closely matches the exit behavior of the teacher gate, achieving approximately the same accuracy and threshold curves. Thus, the exit can be predicted one step earlier just as well as in the same step, validating our scheduler design. Predicting the exit before the loop starts would give the scheduler even more flexibility, but the pre-loop gate does not perform well, confirming the observation of Bae et al. (2025b). For Huginn, the delayed exit decision requires only a threshold recalibration and also incurs no accuracy cost.

Overall, gating is effective for both models, and our delayed exits do not degrade accuracy. However, the minimum loop count remains 2 for the trained gate and 3 for the convergence criterion (which compares two consecutive latent states, plus the additional step from the delayed scheduler), limiting the exit distribution and the compute savings. Ouro and Huginn naturally fit our setup, as they do not exit at those depths, but shallower looped LMs might be more constrained.

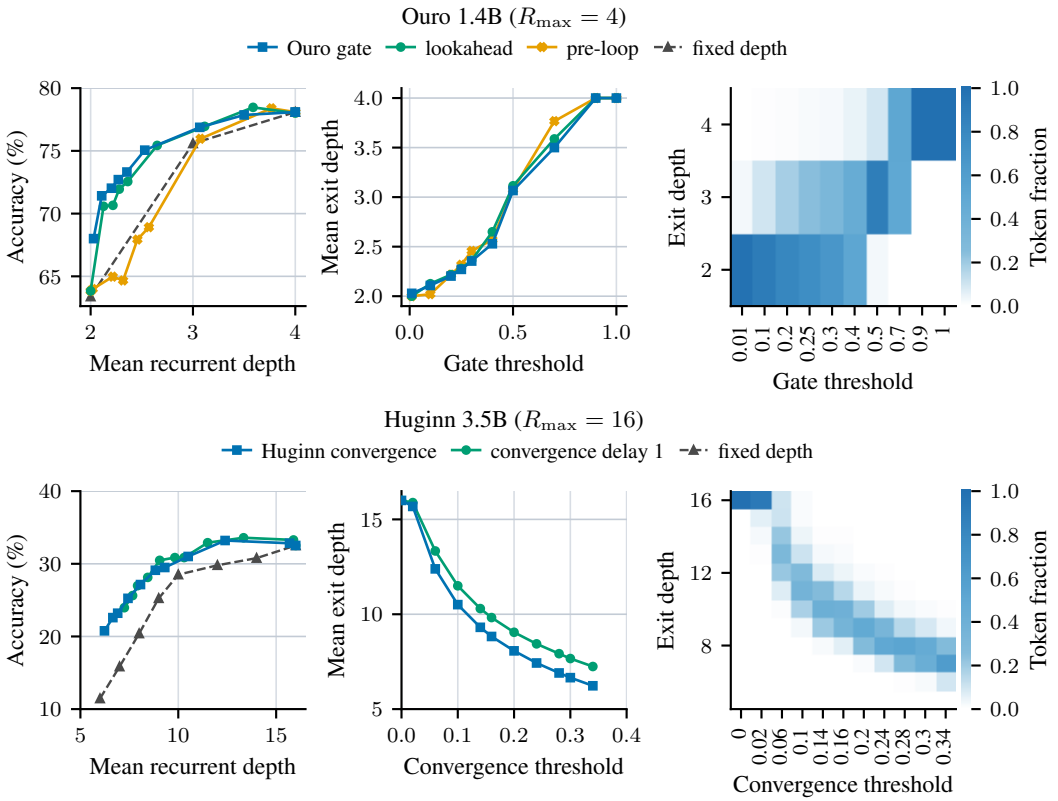

Figure 14: Depth-adaptive decoding against fixed-depth decoding on GSM8K (flexible match, greedy decoding), for Ouro (top) and Huginn (bottom). Each row shows the accuracy–depth frontier (left), the mean exit depth for each threshold (middle), and the exit-depth distribution (right). The distribution panel shows each model's own exit policy ("Ouro gate" and "Huginn convergence").