Recurrent Looped Transformer

Yifan Zhang **Jichen Feng** **Shihan Qin**

yifanzhangresearch@gmail.com

September 12, 2026*

Abstract

Recurrent Looped Transformer (RLT-1) feeds the previous token's final decoder state into the current token's decoder input through a gated merge. A causal encoder supplies token representations and global key–value memory, while the decoder maintains local sliding-window attention (SWA) caches. This recurrence extends the computation path with sequence length at a fixed number of block evaluations per token, but requires sequential decoder updates during both training and prompt prefill. We evaluate five eight-layer encoder–decoder splits against a decoder-only Transformer on six algorithmic tasks, using three initialization seeds and shared training and test data. Several RLT-1 splits learn parity earlier, and the $5 + 3$ and $7 + 1$ splits retain 100% accuracy at 256 bits in all three seeds. On swaps-based permutation tracking at length 512, $4 + 4$ reaches $55.70 \pm 25.78\%$ final-state accuracy, compared with $0.85 \pm 0.30\%$ for the Transformer (mean $\pm$ sample standard deviation). The gains depend on the task and depth split: teacher-forced addition accuracy falls beyond trained operand widths, flat modular arithmetic is sensitive to initialization, and standard permutation tracking remains difficult.

Project Page: https://github.com/yifanzhang-pro/recurrent-looped-tranformer

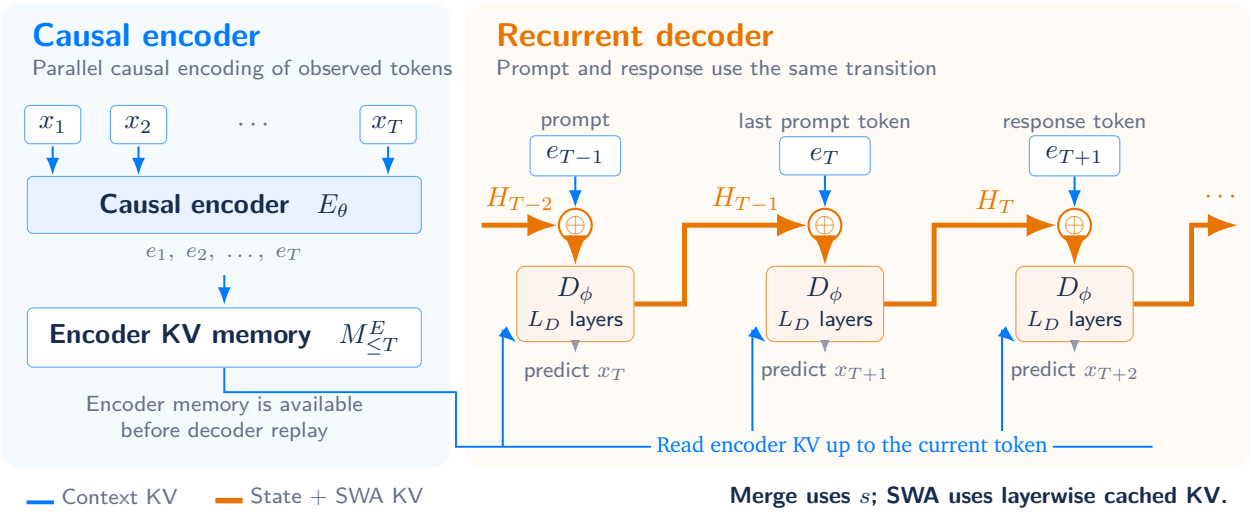

Figure 1 Recurrence across prompt and response. The encoder builds causal key–value (KV) memory; the decoder processes tokens in order. Orange arrows carry the decoder state $H_t = (s_t, C_t^D)$: the merge reads s_t, and each SWA layer reads its own cache. Blue arrows supply encoder memory up to the current position. The drawing shows the last two prompt updates and the first response update.

*Revised: September 17, 2026

1 Introduction

Algorithmic state tracking requires a model to update a state as each input arrives. A recurrent network represents this process directly: the output of one update becomes an input to the next. For a Transformer language model, adding such a connection lets later tokens build on the computation performed for earlier tokens through a path that grows with sequence length. The practical questions are where to place the feedback and how much computation to allocate to each update.

We study Recurrent Looped Transformer (RLT-1), which places feedback between a causal encoder and a Transformer decoder. The encoder builds token representations and global key–value memory. At each position, a gated merge combines the current encoder representation with the preceding final decoder output. The decoder then attends to the encoder prefix and its own recent activations before predicting the next token. Prompt and response tokens follow the same transition (Figure 1).

The encoder and decoder serve different computational roles. Known tokens can be encoded in parallel, while decoder updates follow token order. With decoder depth L_D, processing t tokens creates a recurrent path through tL_D decoder blocks. Moving layers from the encoder to the decoder lengthens this path at a fixed total layer count, but also increases sequential work during training and prefill. We therefore compare several depth splits instead of fixing an equal allocation.

Across six algorithmic tasks and three initialization seeds, the best split depends on the computation being learned. The $5+3$ and $7+1$ models generalize parity to 256 bits in every seed, whereas larger decoder allocations perform better on long swaps-based permutation sequences. Neither pattern extends uniformly across tasks: addition deteriorates beyond trained widths, and flat modular arithmetic has large differences between initializations. We compare five depth splits with a decoder-only Transformer, measuring each task both within and beyond its training lengths.

RLT-1 combines encoder-memory reuse (Sun et al., 2024) with cross-token feedback, a mechanism explored by Feedback Transformer, Recurrent Transformer, Full-bandwidth Transformer, T^{2}MLR, and Latent Recurrent Transformer (Fan et al., 2020; Oncescu et al., 2026; Wang et al., 2026; Cai et al., 2026; Huang et al., 2026). Our study examines feedback through the full decoder and the effect of allocating depth between encoding and recurrent updates. The appendices specify training, cache reuse, and policy replay, and describe RLT-2, an untested extension that shares one feedback state across a chunk to parallelize known-token decoding.

2 Method

This section defines RLT-1, the architecture evaluated in our experiments. Appendix I gives the chunked RLT-2 extension.

2.1 Sequence and state

Let $x_{1:S}$ be an independent sequence beginning with $x_1 = \mathrm{BOS}$. At inference, $x_{1:T}$ is the prompt and later tokens form the response. The model uses the same conditional distribution on either side of T. Let L_E and L_D be the encoder and decoder depths, and let d be the residual width. We write the encoder representation $e_t \in \mathbb{R}^d$ and recurrent output $s_t \in \mathbb{R}^d$ as column vectors.

The complete decoder state is $H_t = (s_t, C_t^D)$. The cache C_t^D holds the key/value projections retained by each decoder SWA layer. The window size $W \geq 1$ includes the current token, so each

layer retains at most $W-1$ past positions after an update. We collect all parameters, including the learned initial state $s_\star$, in Θ. The encoder and decoder are denoted by E_θ and D_ϕ.

2.2 Causal encoder and memory

For an observed prefix, compute

$$e_{1:T} = E_\theta(x_{1:T}). \tag{2.1}$$

Positions within each encoder layer can be processed together using a causal mask. Encoder layers remain sequential. For decoder memory group $g \in \{1, \dots, G\}$, construct

$$k_t^g = \mathcal{P}_K^g(e_t, t), \qquad v_t^g = W_V^g \,\mathrm{RMSNorm}_E(e_t), \qquad M_{\le t}^g = \{(k_j^g, v_j^g)\}_{j=1}^t. \tag{2.2}$$

The key map includes normalization, projection, and any positional transformation. Decoder layer ℓ reads group $g(\ell)$: $G = 1$ shares memory across layers, while $G = L_D$ allows separate projections for each layer. The decoder reads two stores through separate attention sublayers. Cross-attention reads encoder memory $M_{\le t}$ over the full prefix; causal SWA reads decoder KV within a bounded window.

2.3 One transition for every token

Initialize once, before BOS:

$$H_0 = (s_\star, \varnothing). \tag{2.3}$$

For every observed or sampled token, apply

$$u_t = \mathrm{Merge}(e_t, s_{t-1}), \tag{2.4}$$

$$H_t = (s_t, C_t^D) = D_\phi(u_t; M_{\le t}, C_{t-1}^D, t), \qquad t \ge 1, \tag{2.5}$$

$$p_\Theta(x_{t+1} \mid x_{1:t}) = \mathrm{softmax}\big(W_o \,\mathrm{RMSNorm}_o(s_t)\big)_{x_{t+1}}. \tag{2.6}$$

Define $F_t(H) = D_\phi(\mathrm{Merge}(e_t, s); M_{\le t}, C^D, t)$ for $H = (s, C^D)$. The encoder features can be computed before the decoder runs. To predict the first response token, the decoder must then process every prompt position:

$$H_T = F_T \circ F_{T-1} \circ \cdots \circ F_1(H_0). \tag{2.7}$$

Generation continues from H_T. After sampling x_{T+1} from s_T, encode it incrementally, append its encoder-derived KV, and compute $H_{T+1} = F_{T+1}(H_T)$, including every decoder SWA cache update. Neither component of H_T is reset at the serving boundary.

2.4 Prompt prefill and incremental decoding

Prefill first computes the causal encoder representations and memory of the prompt, then evaluates $H_1, \dots, H_T$ in order. At decoder position t, attention is restricted to $M_{\le t}$ even though the entire prompt memory is available. Generation uses

$$(e_t, C_t^E) = E_\theta^{\mathrm{step}}(x_t, C_{t-1}^E) \tag{2.8}$$

with encoder cache C^E, followed by a memory append and decoder update. Observed tokens may be encoded in chunks if the encoder cache is preserved. The decoder still processes every token in order: each update produces the hidden state and SWA KV needed by later positions.

2.5 Merge and decoder blocks

We use the following gated merge:

$$r_{t-1} = \mathrm{RMSNorm}_s(s_{t-1}), \tag{2.9}$$

$$g_t = \sigma\big(W_g[e_t; r_{t-1}] + b_g\big), \tag{2.10}$$

$$u_t = e_t + \alpha\, g_t \odot W_s r_{t-1}. \tag{2.11}$$

Here $W_g \in \mathbb{R}^{d\times 2d}$, $W_s \in \mathbb{R}^{d\times d}$, and α controls the feedback scale. The experiments use $\alpha = 0.1$. A scalar gate or low-rank W_s would reduce the merge cost.

Starting from $z_t^0 = u_t$, each decoder block applies causal SWA, encoder-memory cross-attention, and a feed-forward network (FFN):

$$q_t^{D,\ell} = \mathcal{P}_Q^{D,\ell}(z_t^{\ell-1}, t), \tag{2.12}$$

$$k_t^{D,\ell} = \mathcal{P}_K^{D,\ell}(z_t^{\ell-1}, t), \qquad v_t^{D,\ell} = W_V^{D,\ell}\,\mathrm{RMSNorm}_{S,\ell}(z_t^{\ell-1}), \tag{2.13}$$

$$b_t^\ell = z_t^{\ell-1} + \mathrm{Attn}_\ell^D\Big(q_t^{D,\ell}, \{(k_j^{D,\ell}, v_j^{D,\ell})\}_{j=\max(1,t-W+1)}^{t}\Big), \tag{2.14}$$

$$a_t^\ell = b_t^\ell + \mathrm{Attn}_\ell^M\Big(\mathcal{P}_Q^{M,\ell}(b_t^\ell, t), M_{\le t}^{g(\ell)}\Big), \tag{2.15}$$

$$z_t^\ell = a_t^\ell + \mathrm{FFN}_\ell(\mathrm{RMSNorm}_{D,\ell}(a_t^\ell)), \qquad s_t = z_t^{L_D}. \tag{2.16}$$

The attention operators include their output projections; query/key maps include their respective normalizations and positional transformations. At each layer, current KV is formed before SWA, using the layer input, so current-position attention introduces no circular dependency. Historical decoder KV comes from C_{t-1}^D. After the update, retain positions $\max(1, t-W+2), \ldots, t$ in C_t^D; this set is empty for $W = 1$. Position metadata follows the same convention during prefill, sampling, and replay. Alternative sublayer orders define different variants and must be used consistently in all execution modes.

Figure 6 expands the encoder and decoder stacks for one token update.

2.6 Recurrent depth and execution cost

After t tokens, the feedback path traverses tL_D decoder blocks, while each token evaluates $L_E + L_D$ blocks. The influence of earlier states depends on the gates, projections, and products of transition Jacobians. Known tokens can be encoded in parallel, but a prompt of length T requires T sequential decoder updates. Updates from independent sequences can share a batch, with separate states and caches. Appendix B derives the work and storage costs and proves invariance to the prompt–response split.

2.7 Training and state replay

Training uses teacher forcing and cross-entropy on the selected next-token targets. Full backpropagation through time (BPTT) follows recurrent outputs, decoder KV, and encoder memory; masking a target loss does not skip its state update. The truncation interval used in our experiments covers every training sequence, so it cuts no gradients. For reinforcement learning (RL), evaluating a sampled response after a parameter update requires replaying its history to rebuild the state under the current parameters. Appendix D gives the objectives and replay conditions for pretraining, supervised fine-tuning, and policy gradients.

3 Algorithmic Experiments

We compare encoder–decoder depth splits on addition, parity, modular arithmetic, and permutation state tracking. Each of six architectures is trained on all six tasks with initialization seeds 42, 43, and 44: 108 completed runs, each with 2,000 optimizer steps. We report training-length validation curves and length generalization from the best in-distribution checkpoint of each run. All aggregate results show the mean and sample standard deviation across the three initializations.

3.1 Models, tasks, and evaluation protocol

All models have eight logical layers, width 512, FFN width 1,365, and four attention heads. We compare untied RLT-1 splits $4+4$, $5+3$, $6+2$, $7+1$, and $8+0$ with an eight-layer decoder-only Transformer. RLT-1 uses an SWA window of eight, one shared encoder-memory group, and feedback scale $\alpha = 0.1$. The $8+0$ variant has no decoder blocks but still applies the gated recurrent merge. RLT-1 has 26.10–28.73M parameters, compared with 25.31M for the Transformer (Table 4).

All runs use global batch 512, microbatch 32, AdamW with $(\beta_1, \beta_2) = (0.9, 0.95)$, weight decay 0.1, and gradient clipping at norm 1. The base learning rate rises to 10^{-4} over 200 steps, then follows cosine decay to 5×10^{-6} at step 2,000. Both architectures use the same width-μP recipe and run in CPU FP32 with four threads. RLT-1 uses a TBPTT interval of 128 tokens, which covers every training sequence in these tasks and therefore cuts no gradients here. Validation runs every 100 steps. The full budget is 1,024,000 training examples per run.

- **Addition:** randomly sample 1–8-digit operands and serialize their digits in reverse order. The 256 validation examples use the same range. We measure teacher-forced answer-token accuracy: each prediction receives the correct preceding tokens, and scoring includes answer formatting and EOS while excluding prompt and padding positions.
- **Parity:** train on binary strings of lengths 3–40 and supervise the final parity bit. Validation pools 256 examples at each of lengths 32, 36, and 40.
- **Modular arithmetic:** evaluate expressions modulo five using $+$, $-$, and $\times$. The flat variant respects multiplication precedence and trains on odd expression lengths 3–39, with validation lengths 33, 35, and 39. The bracketed variant uses expression trees, trains on lengths 3–40, and validates at 32, 36, and 40. Each validation length has 256 examples; only the final value is supervised.
- S_5 **state tracking:** compose 32 permutations and supervise the running product after each input, following Grazzi et al. (2024). Standard inputs range over all 120 permutations; the swaps variant uses the identity and the ten single transpositions. Each variant has 256 validation sequences of length 32. We report final-state accuracy as the primary metric and prefix-token accuracy in the appendix.

Every task uses initialization seeds 42, 43, and 44 with fixed data-generation seeds, training examples, and validation sets. For each run, we pool counts across its configured validation lengths before averaging across initializations. The sample standard deviation (SD, denominator $n-1$) measures initialization variability on this fixed data stream. Figure 2 shows all three seeds at each optimizer step; Table 1 compares every model at step 2,000, after 1,024,000 training examples per run.

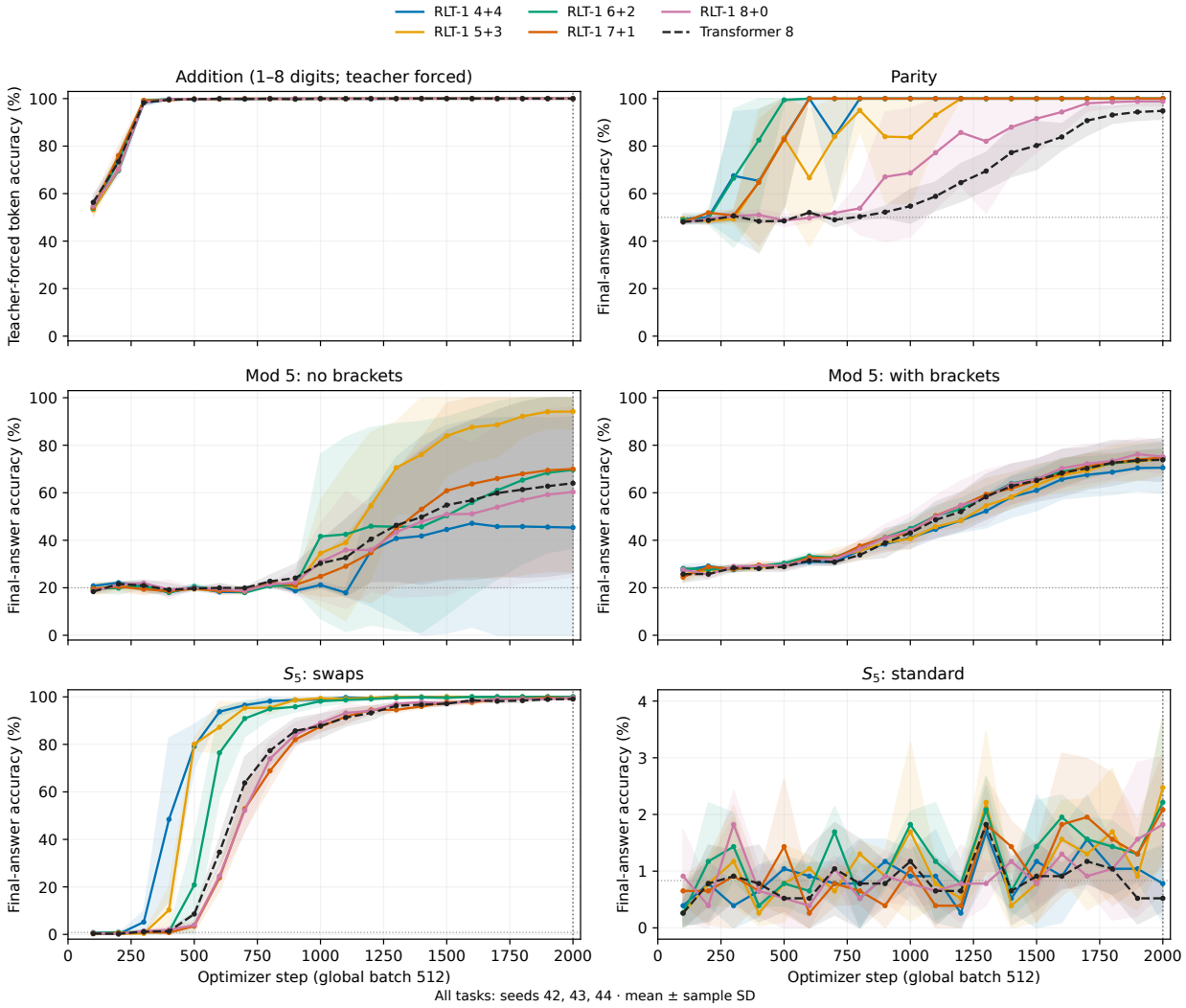

Figure 2 Validation accuracy during training, three seeds for every task. Addition measures teacher-forced answer-token accuracy; the other tasks measure final-label or final-state accuracy. Lines and bands show mean $\pm$ sample SD across seeds 42, 43, and 44 ($n = 3$ at every step); bands are clipped to the accuracy range. Curves are unsmoothed and extend through 2,000 steps. Horizontal dotted lines show uniform-prediction accuracy. Standard S_5 uses a narrower vertical scale.

Table 1 Validation accuracy (%) after 2,000 optimizer steps. All entries are mean ± sample SD over seeds 42, 43, and 44. Addition uses teacher-forced answer tokens; the formal tasks use final labels or states. Every run has consumed 1,024,000 training examples.

Task	RLT-1 4+4	RLT-1 5+3	RLT-1 6+2	RLT-1 7+1	RLT-1 8+0	GPT 8
Addition	100.00 ± 0.00	100.00 ± 0.00	100.00 ± 0.00	100.00 ± 0.00	100.00 ± 0.00	100.00 ± 0.00
Parity	100.00 ± 0.00	100.00 ± 0.00	100.00 ± 0.00	100.00 ± 0.00	98.83 ± 1.92	94.84 ± 3.43
Mod 5, no brackets	45.36 ± 46.43	94.18 ± 7.29	69.62 ± 33.01	70.01 ± 43.15	60.33 ± 35.70	64.02 ± 37.64
Mod 5, brackets	70.53 ± 10.75	74.35 ± 6.71	74.87 ± 3.72	74.78 ± 3.01	75.17 ± 6.78	73.87 ± 9.07
S_5, swaps	100.00 ± 0.00	100.00 ± 0.00	100.00 ± 0.00	99.35 ± 0.81	99.61 ± 0.68	99.09 ± 0.23
S_5, standard	0.78 ± 0.68	2.47 ± 1.26	2.21 ± 1.48	2.08 ± 0.98	1.82 ± 1.19	0.52 ± 0.23

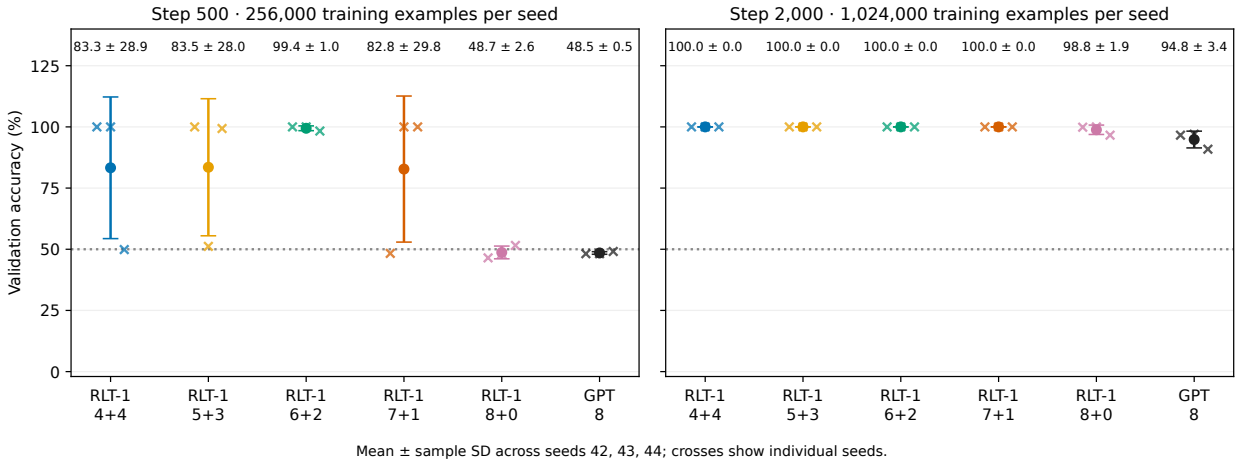

Figure 3 Parity at 500 and 2,000 training steps. Circles and whiskers show the mean and sample SD across seeds 42, 43, and 44; crosses show individual seeds with a small horizontal offset. SD whiskers can extend beyond 100%. Every model uses the same 768 validation examples; the panels correspond to 256,000 and 1,024,000 training examples per seed.

3.2 Parity: learning speed and initialization variability

At 500 steps, RLT-1 $6 + 2$ reaches $99.44 \pm 0.98\%$ validation accuracy, compared with $48.48 \pm 0.53\%$ for the Transformer. RLT-1 $4 + 4$, $5 + 3$, and $7 + 1$ average about 83%, with SDs of 28–30 percentage points; $8 + 0$ remains near chance. All three $6 + 2$ seeds reach 100% by step 600. At step 2,000, splits $4 + 4$ through $7 + 1$ reach $100 \pm 0\%$, while $8 + 0$ reaches $98.83 \pm 1.92\%$ and the Transformer $94.84 \pm 3.43\%$. Figure 3 compares the early and final checkpoints.

3.3 Modular arithmetic with and without brackets

Flat modular arithmetic shows substantial initialization variability at 2,000 steps. RLT-1 $5 + 3$ has the highest mean, $94.18 \pm 7.29\%$, compared with $64.02 \pm 37.64\%$ for the Transformer. The $4 + 4$ scores are 17.58%, 19.53%, and 98.96% for seeds 42, 43, and 44; their mean is $45.36 \pm 46.43\%$. Several models therefore have both successful and near-chance runs at this training budget.

Bracketed expressions produce closer model means: RLT-1 splits range from 70.53% to 75.17%, compared with $73.87 \pm 9.07\%$ for the Transformer. The two generators differ in operator structure and label distribution. Parentheses also consume positions, so equal token lengths can contain

different numbers of arithmetic operations.

3.4 Addition and permutation state tracking

All six architectures reach $100 \pm 0\%$ teacher-forced token accuracy on the mixed 1–8-digit addition validation set at step 2,000. On swaps-S_5, RLT-1 $4 + 4$, $5 + 3$, and $6 + 2$ reach $100 \pm 0\%$ final-state accuracy. The $7{+}1$ and $8{+}0$ means are $99.35{\pm}0.81\%$ and $99.61{\pm}0.68\%$, compared with $99.09{\pm}0.23\%$ for the Transformer.

Standard S_5 remains difficult: mean final-state scores range from 0.52% to 2.47%, against a uniform 120-class reference of 0.83%. Mean prefix-token accuracy ranges from 5.40% to 7.99% (Table 5); averaging over prefixes includes easier early states.

3.5 Length generalization

For every task, architecture, and initialization seed, we select the checkpoint with the lowest in-distribution validation example loss over the completed 2,000-step run. Ties select the earliest checkpoint; test results do not enter selection. All 18 model–seed combinations for a task receive the same examples at each length: 256 addition pairs or 1,024 formal-task sequences. Figure 4 shows the full grids; Table 2 reports the longest tested lengths. Appendix J.5 gives checkpoint steps and supplementary metrics.

Addition and parity. Teacher-forced addition accuracy falls when operand widths exceed the 1–8-digit training range. At nine digits per operand, RLT-1 $7 + 1$ reaches $68.05 \pm 3.64\%$, compared with $58.61 \pm 2.44\%$ for the Transformer. By 32 digits, model means lie between 14.89% and 16.84%. Appendix J.7 reports a separate sixteen-layer study with both greedy exact-answer and teacher-forced token accuracy.

Parity generalization differs across depth splits even when training-length accuracy is perfect. At 256 bits, $5 + 3$ and $7 + 1$ retain $100 \pm 0\%$, while $6 + 2$ reaches $84.05 \pm 27.63\%$ and $4 + 4$ reaches $66.76{\pm}28.78\%$. The $8{+}0$ mean is $68.91{\pm}27.39\%$, and the Transformer is near chance at $50.07{\pm}1.63\%$.

Modular arithmetic. On flat expressions of length 63, RLT-1 $5 + 3$ reaches $66.96 \pm 40.04\%$, compared with $24.48 \pm 3.89\%$ for the Transformer. At length 127, $5 + 3$ retains $39.65 \pm 19.81\%$; the Transformer is at $19.30 \pm 1.04\%$. By length 255, all model means lie between 18.00% and 21.42%, near the 20% uniform reference. The large intermediate-length SDs reflect the different training outcomes seen in the flat mod-5 validation curves. Bracketed mod-5 also loses accuracy as expressions grow longer.

Permutation state tracking. At 256 operations on swaps-S_5, RLT-1 $4 + 4$ reaches $97.30 \pm 2.76\%$ final-state accuracy, compared with $0.85 \pm 0.30\%$ for the Transformer. The $5{+}3$ and $6{+}2$ means are $92.71 \pm 1.87\%$ and $78.91 \pm 22.26\%$; $7{+}1$ and $8{+}0$ are near the uniform reference. At 512 operations, $4 + 4$ retains $55.70 \pm 25.78\%$ final-state accuracy and $91.16 \pm 6.09\%$ prefix-token accuracy, versus $0.85 \pm 0.30\%$ and $9.33 \pm 0.11\%$ for the Transformer. Standard S_5 remains low at this length, with mean final-state accuracies from 0.72% to 1.43%. The preferred allocation therefore depends on the task: $5 + 3$ and $7 + 1$ consistently extrapolate parity, whereas larger decoder allocations support longer swaps-S_5 sequences.

Figure 5 compares three measures of state-tracking accuracy across test lengths. Prefix-token accuracy averages correctness over all intermediate states; final-state accuracy scores only the last state. Whole-sequence accuracy requires every prefix prediction to be correct, measuring whether

Table 2 Accuracy (%) at the longest tested length. Entries are mean ± sample SD across three initialization seeds, using the checkpoint and scoring rules of Figure 4. Addition reports teacher-forced token accuracy; formal tasks report final-label or final-state accuracy.

Task (test length)	RLT-1 4+4	RLT-1 5+3	RLT-1 6+2	RLT-1 7+1	RLT-1 8+0	GPT 8
Addition (32)	15.30 ± 0.44	14.89 ± 2.27	15.74 ± 1.58	15.59 ± 1.57	15.31 ± 0.93	16.84 ± 1.45
Parity (256)	66.76 ± 28.78	100.00 ± 0.00	84.05 ± 27.63	100.00 ± 0.00	68.91 ± 27.39	50.07 ± 1.63
Mod 5, no brackets (255)	18.00 ± 0.39	20.57 ± 0.62	21.42 ± 0.49	19.34 ± 0.54	20.44 ± 0.91	20.35 ± 2.01
Mod 5, brackets (256)	25.20 ± 1.71	25.81 ± 4.34	21.58 ± 1.86	22.04 ± 1.72	22.30 ± 1.13	25.07 ± 2.05
S_5, standard (512)	1.24 ± 0.30	1.43 ± 0.60	1.30 ± 0.31	0.88 ± 0.20	0.72 ± 0.31	0.81 ± 0.06
S_5, swaps (512)	55.70 ± 25.78	34.86 ± 6.10	22.14 ± 20.95	0.85 ± 0.06	0.85 ± 0.20	0.85 ± 0.30

tracking remains correct throughout the sequence. A correct final state can follow incorrect intermediate predictions, while prefix-token accuracy can remain high when errors occur late in the sequence.

3.6 Training stability and comparison scope

Parity RLT-1 $5+3$ with seed 42 drops from 100% at step 500 to 48.05% at step 600, then returns to 100% at step 700. Figure 15 shows the unsmoothed losses, and Appendix J gives individual parity curves. This temporary regression and the flat mod-5 seed differences show that final averages can hide unstable learning trajectories.

The comparison with a decoder-only Transformer changes feedback, attention structure, parameter count, and compute together. RLT-0 retains the encoder–decoder split and attention structure while removing the learned initial state and gated feedback (Figure 12); matched runs are needed to isolate feedback. The present experiments cover algorithmic accuracy and learning curves, with no hardware-throughput or RL measurements.

4 Related Work

Hybrid Transformer–RNN models. Chen et al. (2018) combine a Transformer encoder with an LSTM-based RNMT+ decoder for machine translation. RLT-1 uses the same broad split between parallel encoding and recurrent decoding. Its decoder consists of Transformer blocks with encoder-memory cross-attention and SWA, and it processes both prompt and response tokens in a causal language model.

Encoder-derived memory. YOCO builds reusable KV memory for an upper cross-decoder and allows early exit during prefill (Sun et al., 2024). DeepSeek-V4.1-Flash projects global decoder KV from final encoder states and maintains separate decoder SWA (DeepSeek-AI, 2026). RLT-1 also builds global memory from encoder states, but its recurrent decoder must process every prompt token.

Temporal feedback. Feedback Transformer forms a learned weighted sum of each token's representations across layers and lets subsequent tokens attend to this shared memory at every layer (Fan et al., 2020). RLT-1 instead injects the preceding final decoder output through a gated merge at the decoder input, while retaining separate encoder-derived global memory and layerwise decoder SWA caches. Recurrent Transformer constructs each layer's persistent KV from that layer's

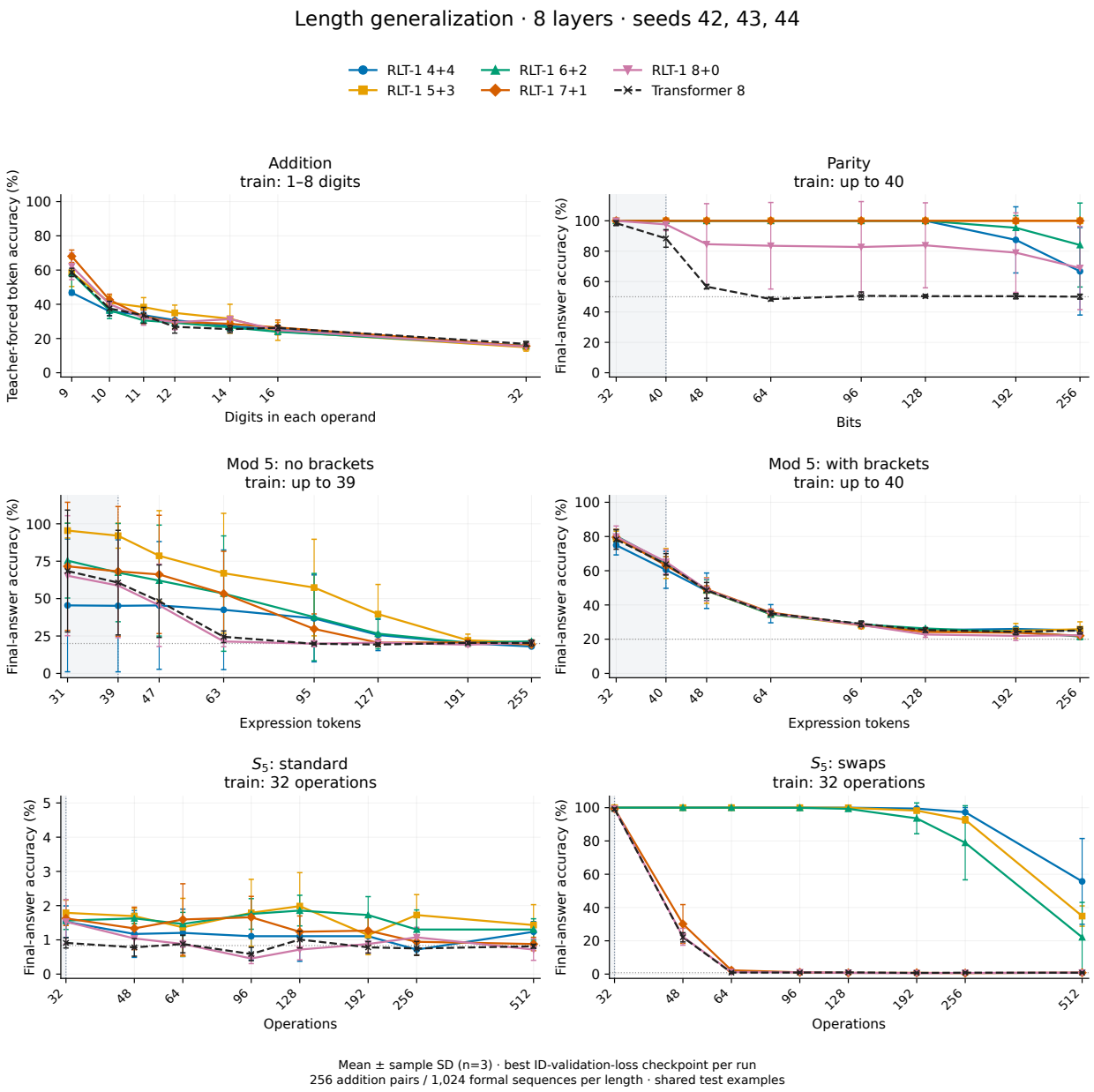

Figure 4 Length generalization of the best in-distribution checkpoints, three initialization seeds. Addition measures teacher-forced answer-token accuracy on 256 pairs per operand width; the other tasks measure final-label or final-state accuracy on 1,024 sequences per length. Points and error bars show mean ± sample SD across seeds 42, 43, and 44, using identical test examples. SD whiskers may extend outside the accuracy range. Gray regions mark trained lengths; horizontal dotted lines show uniform-prediction accuracy. Flat mod-5 uses actual odd expression lengths, excluding BOS and the final equals sign.

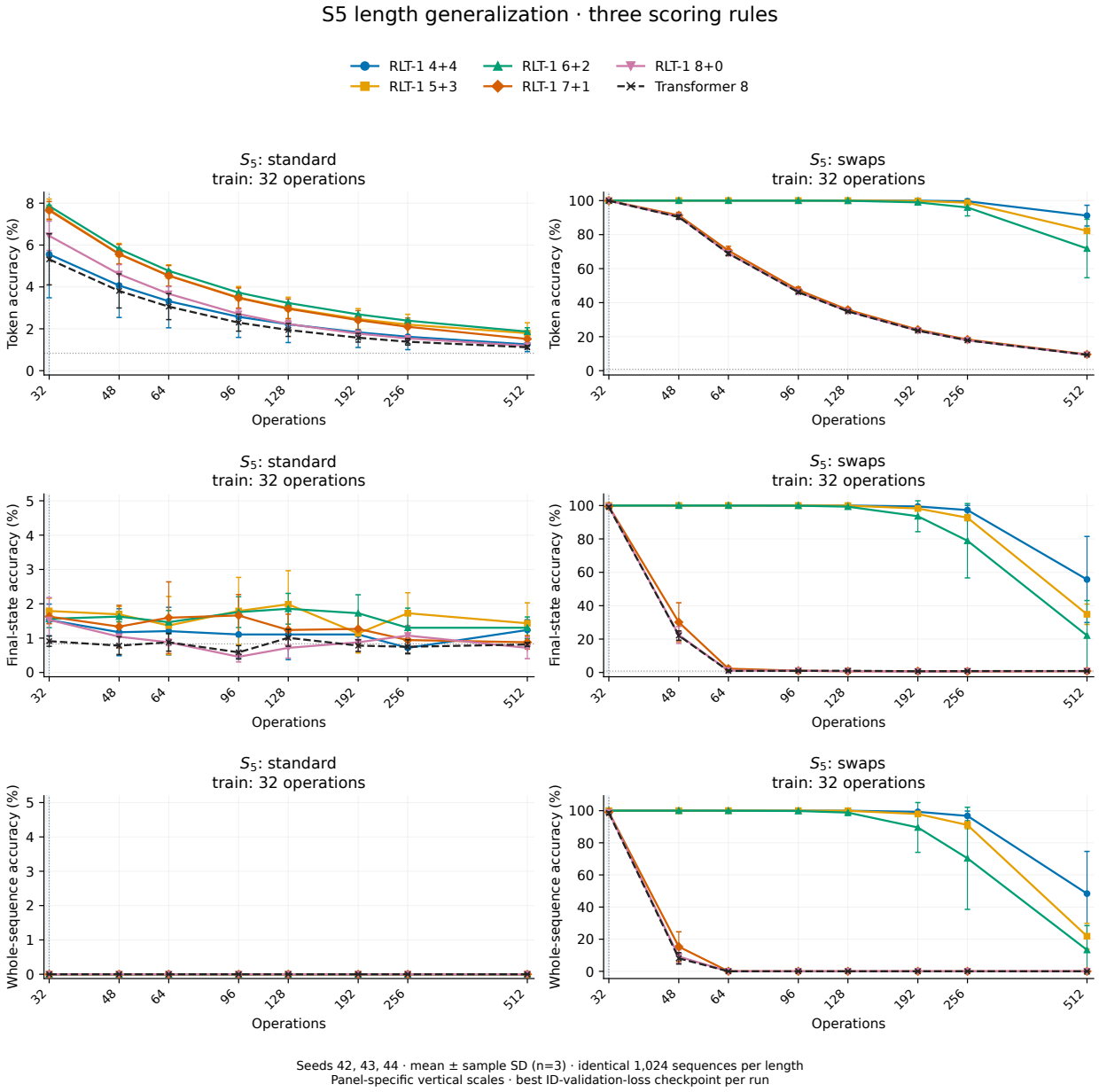

Figure 5 S_5 **length generalization under three scoring rules.** Rows show prefix-token, final-state, and whole-sequence accuracy; columns show standard and swaps inputs. Every model and seed receives the same 1,024 sequences per length. Points and error bars show mean ± sample SD across three initialization seeds. Whole-sequence success requires all prefix predictions to be correct. Panel-specific vertical scales expose low accuracies on standard S_5.

output and provides an exact tiling schedule to improve memory movement (Oncescu et al., 2026). RLT-1's recurrent dependency spans the full decoder, whereas Recurrent Transformer's feedback is layerwise.

Cross-token latent feedback. Full-bandwidth Transformer combines the previous top-layer hidden state with the next token embedding through a gated linear unit, retaining the Transformer stack and KV cache (Wang et al., 2026). Its multi-pass training shifts hidden states between passes to allow token-parallel teacher forcing, with a prefix mixin to address differences between prompts and generation. RLT-1 places this kind of output-to-input feedback at the encoder–decoder interface and trains by replaying the decoder sequentially over the full history.

T^2MLR feeds a cached middle-layer representation from the previous token into an earlier layer at the current position (Cai et al., 2026). Its experiments find that recurrence between middle layers can outperform recurrence through the full network. Training approximates temporal states with a fixed number of Jacobi iterations and controls backward depth separately. RLT-1 uses feedback through the entire decoder and takes full-history replay with full BPTT as the reference, with optional TBPTT.

Latent recurrent language models. Latent Recurrent Transformer (LRT) reuses a high-level state from the previous token through KV projection and residual injection, keeping a decoder-only backbone and one forward pass per generated token (Huang et al., 2026). Its interleaved parallel training refines subsets of positions from a shared state buffer. For RL, it initializes this buffer with detached rollout states to reduce differences between rollout and recomputation. RLT-1 separates encoder memory from the recurrent decoder and rebuilds the full history under current parameters for exact policy replay.

Continuous latent computation. Coconut feeds the last hidden state back as the next input embedding during a latent reasoning phase, using a curriculum that replaces textual reasoning steps with continuous states (Hao et al., 2024). PonderLM-2 inserts latent steps between ordinary tokens during pretraining and uses Jacobi iterations to approximate their recurrent dependencies in parallel (Zeng et al., 2025). RLT-1 advances its state once per ordinary prompt or response token, without adding latent positions.

Block- and segment-level recurrence. Block-Recurrent Transformers update persistent state vectors with attention and gates, processing a block of tokens in parallel at each recurrent step (Hutchins et al., 2022). Their block-feedback variant lets all layers cross-attend to the recurrent state from the preceding block. Recurrent Memory Transformer appends write-memory tokens to each segment and passes their final representations to the next segment as memory inputs, with BPTT through this connection (Bulatov et al., 2022). RLT-1 updates its final decoder output and SWA caches at every token, alongside separate encoder memory. Its finer update interval requires sequential decoder work within each block. RLT-2 also updates feedback at chunk boundaries, but uses the last ordinary token's final decoder output as its boundary state and gates it into every decoder input in the next chunk, without dedicated memory tokens or a separate bank of recurrent state vectors.

Weight sharing across depth. Universal Transformers share weights across depth and optionally adapt the number of refinement steps by position (Dehghani et al., 2018). Saunshi et al. (2025) study how repeated applications of a shared Transformer stack increase effective depth for reasoning, while recurrent-depth language models vary latent computation at inference time (Geiping et al., 2025). DeepLoop analyzes the effect of repeated parameter visits on residual scaling in weight-tied

looped Transformers and derives a loop-aware scaling rule for the Post-LN architecture (Li et al., 2026). RLT-1 advances its state across tokens and optionally shares compatible encoder and decoder weights.

5 Conclusion

RLT-1 combines encoder memory that can be built in parallel with a decoder that updates recurrently at every token. At a fixed total depth of eight layers, the allocation between these components affects both learning speed and length generalization. Parity is most consistent with $5 + 3$ and $7 + 1$, while longer swaps-based permutation sequences benefit from deeper decoders; addition and modular arithmetic generalize less reliably. The results motivate studying the depth assigned to recurrent computation as an architectural choice.

References

Aydar Bulatov, Yuri Kuratov, and Mikhail S. Burtsev. Recurrent memory transformer. *arXiv preprint arXiv:2207.06881*, 2022. URL https://arxiv.org/abs/2207.06881.

Ziyang Cai, Xingyu Zhu, Yihe Dong, Yinghui He, and Sanjeev Arora. T^2MLR: Transformer with temporal middle-layer recurrence. *arXiv preprint arXiv:2607.15178*, 2026. URL https://arxiv.org/abs/2607.15178.

Mia Xu Chen, Orhan Firat, Ankur Bapna, Melvin Johnson, Wolfgang Macherey, George Foster, Llion Jones, Niki Parmar, Mike Schuster, Zhifeng Chen, Yonghui Wu, and Macduff Hughes. The best of both worlds: Combining recent advances in neural machine translation. *arXiv preprint arXiv:1804.09849*, 2018. URL https://arxiv.org/abs/1804.09849.

DeepSeek-AI. DeepSeek-V4.1-Flash: Pushing the limits of KV cache compression. Technical report, DeepSeek-AI, 2026. URL https://huggingface.co/deepseek-ai/DeepSeek-V4.1-Flash/resolve/main/DeepSeek_V41_Tech_Report.pdf. Sections 2.2 and 3.2.2; publicly available from the official DeepSeek model repository.

Mostafa Dehghani, Stephan Gouws, Oriol Vinyals, Jakob Uszkoreit, and Łukasz Kaiser. Universal transformers. *arXiv preprint arXiv:1807.03819*, 2018. URL https://arxiv.org/abs/1807.03819.

Angela Fan, Thibaut Lavril, Edouard Grave, Armand Joulin, and Sainbayar Sukhbaatar. Addressing some limitations of transformers with feedback memory. *arXiv preprint arXiv:2002.09402*, 2020. URL https://arxiv.org/abs/2002.09402.

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. URL https://arxiv.org/abs/2502.05171.

Riccardo Grazzi, Julien Siems, Arber Zela, Jörg K. H. Franke, Frank Hutter, and Massimiliano Pontil. Unlocking state-tracking in linear rnns through negative eigenvalues. *arXiv preprint arXiv:2411.12537*, 2024. URL https://arxiv.org/abs/2411.12537.

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. URL https://arxiv.org/abs/2412.06769.

Zeyi Huang, Xuehai He, Liliang Ren, Yiping Wang, Baolin Peng, Hao Cheng, Shuohang Wang, Pengcheng He, Jianfeng Gao, Yong Jae Lee, and Yelong Shen. Latent recurrent transformer: Architecture exploration, training strategies, and scaling behavior. *arXiv preprint arXiv:2605.26797*, 2026. URL https://arxiv.org/abs/2605.26797.

DeLesley Hutchins, Imanol Schlag, Yuhuai Wu, Ethan Dyer, and Behnam Neyshabur. Block-recurrent transformers. *arXiv preprint arXiv:2203.07852*, 2022. URL https://arxiv.org/abs/2203.07852.

Shuzhen Li, Yifan Zhang, Jiacheng Guo, Quanquan Gu, and Mengdi Wang. DeepLoop: Depth scaling for looped transformers. *arXiv preprint arXiv:2607.13491*, 2026. URL https://arxiv.org/abs/2607.13491.

Costin-Andrei Oncescu, Depen Morwani, Samy Jelassi, Alexandru Meterez, Mujin Kwun, and Sham Kakade. The recurrent transformer: Greater effective depth and efficient decoding. *arXiv preprint arXiv:2604.21215*, 2026. URL https://arxiv.org/abs/2604.21215.

Nikunj Saunshi, Nishanth Dikkala, Zhiyuan Li, Sanjiv Kumar, and Sashank J. Reddi. Reasoning with latent thoughts: On the power of looped transformers. In *International Conference on Learning Representations*, 2025. URL https://arxiv.org/abs/2502.17416.

Yutao Sun, Li Dong, Yi Zhu, Shaohan Huang, Wenhui Wang, Shuming Ma, Quanlu Zhang, Jianyong Wang, and Furu Wei. You only cache once: Decoder-decoder architectures for language models. *arXiv preprint arXiv:2405.05254*, 2024. URL https://arxiv.org/abs/2405.05254.

Xi Wang, Ziyang Cai, Zheng Zhan, Harry Dong, Ying Fan, Gustavo de Rosa, Tim Pearce, and John Langford. Full-bandwidth transformer. *arXiv preprint arXiv:2608.08888*, 2026. URL https://arxiv.org/abs/2608.08888.

Boyi Zeng, He Li, Shixiang Song, Yixuan Wang, Ziwei He, Xinbing Wang, and Zhouhan Lin. PonderLM-2: Pretraining LLM with latent thoughts in continuous space. *arXiv preprint arXiv:2509.23184*, 2025. URL https://arxiv.org/abs/2509.23184.

Yifan Zhang et al. Reliable RL scaling requires accounting for Prefill–Decode kernel mismatch. Technical report, Pretraining-RL-Science project, August 2026. URL https://github.com/yifanzhang-pro/Pretraining-RL-Science/blob/master/Prefill_Decode_Kernel_Mismatch.pdf. Dated August 6, 2026; revised August 24, 2026.

Appendix

A Architecture and Execution Details

A.1 Sharing encoder and decoder weights

The reference tied RLT-1 sets $L_E = L_D = L$. Encoder self-attention at layer ℓ and decoder SWA at layer ℓ share compatible query, key, value, and output projection matrices; their FFNs are also shared. The encoder uses its causal context, whereas decoder SWA uses decoder activations within its window. Decoder cross-attention has separate query/output projections and the memory projections of Equation (2.2). The encoder and decoder retain separate normalizations; the merge and readout are also separate modules. Cross-attention adds computation to the shared attention and FFN.

The two passes share weights but compute separate activations. An untied E_θ, D_ϕ uses separate weights and keeps the same recurrence. Memory groups can be shared in either version; each decoder layer still maintains its own SWA cache.

B Computational Properties

B.1 Prompt–response consistency

Proposition B.1 (Invariance to the serving split)**.** Fix the parameters, token sequence, position convention, and initial state for an independent sequence. Assume exact arithmetic, mathematically equivalent causal encoder execution, identical SWA windows and cache updates, and deterministic decoder operations. Processing a prefix with batched encoder prefill followed by recurrent decoder updates gives the same states and next-token distributions as processing it incrementally. The conditional distribution for a fixed token history is therefore unchanged when the prompt–response split moves.

Proof. Causal encoder equivalence gives the same e_t and $M_{\le t}$ in both schedules. Both start from $H_0 = (s_\star, \varnothing)$. If their states agree at $t-1$, Equation (2.5) applies the same operations to the same inputs at t, so their states agree at t. The result follows by induction, since the transition does not depend on the serving split. □

B.2 Prefill work and sequential depth

Let $C_E^{\mathrm{pf}}(T)$ be encoder prefill work, $C_M(T)$ the memory projection work, and $C_D^{\mathrm{step}}(t)$ a decoder evaluation over t encoder-memory entries and at most W decoder positions per layer, including merge overhead. Then

$$C_{\mathrm{prefill}}(T) = C_E^{\mathrm{pf}}(T) + C_M(T) + \sum_{t=1}^{T} C_D^{\mathrm{step}}(t). \tag{B.1}$$

For dense attention and width-proportional KV, a coarse arithmetic estimate is

$$O\big((L_E + L_D)(Td^2 + T^2 d) + GTd^2 + L_D T \min(W, T) d\big). \tag{B.2}$$

Prefill requires T sequential decoder transitions, each containing L_D blocks. This dependency can reduce hardware utilization even when the arithmetic complexity has the same order as a dense Transformer.

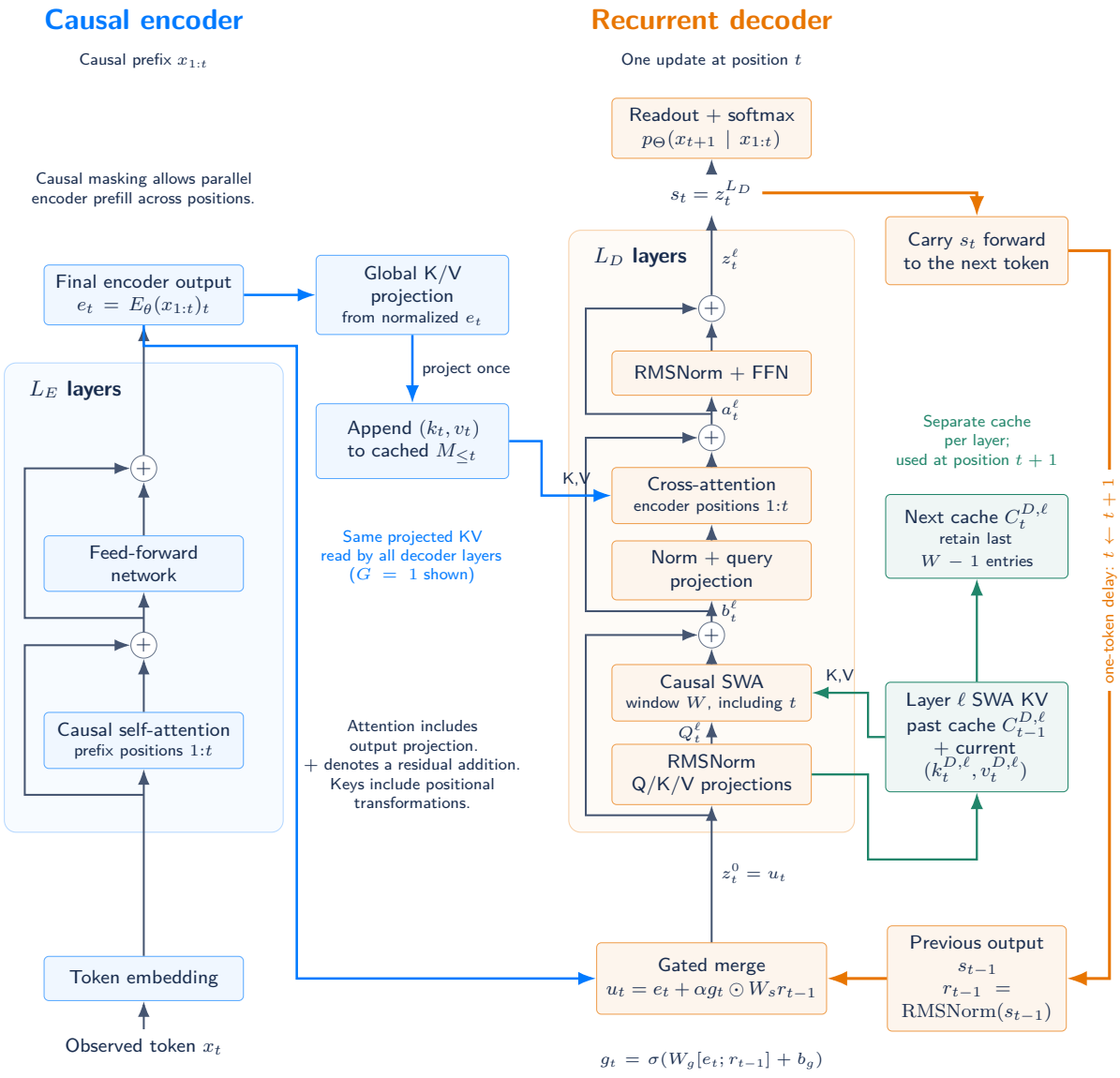

Figure 6 RLT-1 architecture for one token update. Computation flows upward through the encoder and decoder stacks; each outlined stack repeats for its indicated depth. The encoder output supplies both the gated merge and the global KV projection. The drawing uses $G = 1$: all decoder layers read the same cached, projected global KV with their own queries. Each decoder layer constructs separate SWA KV from its own input, attends to its local window, then reads global memory and applies an FFN. Residual paths bypass each attention or FFN sublayer. The final decoder output s_t feeds the next token's merge, while each layer retains its own SWA cache. Blue arrows carry encoder features and global memory, green arrows carry local KV, and orange arrows carry recurrent hidden states.

Each new token evaluates $L_E + L_D$ blocks, plus merge and memory projection. Attention work still grows with context length. With effective KV width d_{KV}, inference cache storage is approximately

$$O\big((L_E + G)t\, d_{\mathrm{KV}} + L_D \min(t, W-1) d^D_{\mathrm{KV}} + d\big). \tag{B.3}$$

Here d^D_{KV} is the effective decoder SWA KV width, and the $O(d)$ term stores the current recurrent output. The SWA term counts retained history; current-position KV and training activations require additional storage. Weight sharing reduces parameter storage, while both passes and their caches remain necessary.

B.3 Depth of the recurrent path

After a prompt of length T and n processed response tokens, the recurrent path has passed through $(T+n)L_D$ decoder blocks. Figure 7 compares this growing path with the fixed block count per token. The influence of earlier states depends on the learned gates, projections, and products of transition Jacobians.

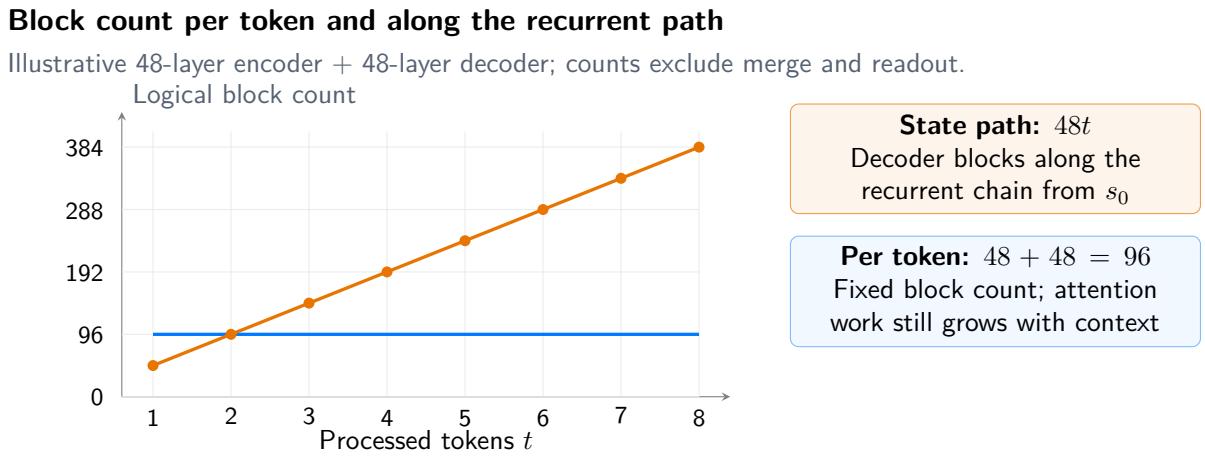

Figure 7 Recurrent path length and block count per token. For $L_E = L_D = 48$, the path traverses $48t$ decoder blocks after t tokens. Each token evaluates 96 encoder and decoder blocks. Orange counts decoder blocks along the recurrent path; blue counts blocks in both stacks per token.

C Hardware Execution

C.1 Batching independent sequences

For a known training sequence or prompt, the encoder computes features and memory projections across positions in parallel. The decoder then processes positions in order. As Figure 8 shows, independent sequences can contribute their next decoder update to one batch. Each sequence keeps its own encoder prefix, SWA cache, recurrent output, and position.

At inference, each step consists of an encoder update, memory append, merge, decoder pass, and readout. Batching requests increases matrix-operation sizes and allows weight reuse. Small

batches and uneven sequence lengths may limit utilization.

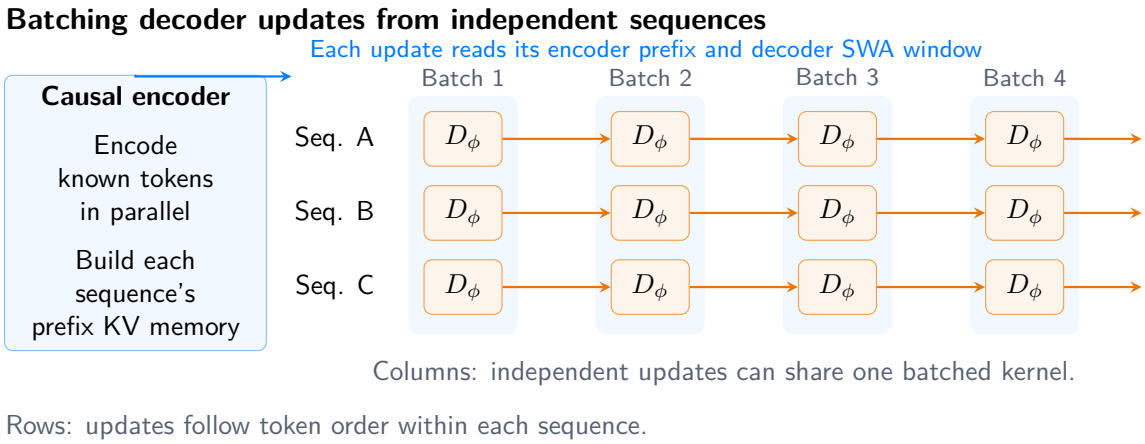

Figure 8 Batching decoder updates across sequences. Columns group independent updates into a batch; rows follow each sequence in token order. Each update reads its own encoder prefix and decoder SWA window. Known tokens can be encoded in parallel before replay; generated tokens are encoded as they arrive.

C.2 Memory traffic and parameter reuse

Encoder KV stays fixed while the prefix and parameters are unchanged. Decoder layers reuse that memory and append their own KV to separate SWA caches. Fewer memory groups reduce KV storage and projection work; more groups allow different transformations for each layer. Sharing encoder and decoder weights may also help keep weights resident on the accelerator.

One proposed optimization is to fuse normalization, gating, state projection, and residual addition while preserving their dependencies and numerical behavior. Attention kernels must preserve the encoder-prefix mask and local SWA window.

C.3 Training memory and numerical agreement

Activation checkpointing reduces stored activations by recomputing them during backpropagation through time (BPTT). Batch size and checkpoint placement depend on sequence length and available accelerator memory. Full BPTT retains gradients through the entire recurrent history.

Zhang et al. (2026) describe how the executed policy depends on precision, cache construction, reductions, and sampling transforms as well as weights. Sampler and trainer implementations therefore need consistent positional conventions, stochastic behavior, and cache contents, with numerical checks across execution modes.

D Training Objectives

D.1 Autoregressive pretraining

Full-sequence next-token prediction is the base objective:

$$\mathcal{L}_{\mathrm{PT}}(\Theta) = -\mathbb{E}_{x_{1:S}}\left[\frac{1}{S-1}\sum_{t=1}^{S-1}\log p_\Theta(x_{t+1} \mid x_{1:t})\right]. \tag{D.1}$$

Initialize $H_0 = (s_\star, \varnothing)$, compute causal encoder features, and update the decoder over positions $1, \ldots, S-1$. Every non-BOS target contributes to the loss; the encoder, memory projections, merge, and decoder train jointly. At each independent document, reset both decoder and encoder state, reset positions, and prevent attention across document boundaries. When training on segments, specify the initial context and state: dropping an earlier state changes the history on which the likelihood is conditioned.

D.2 Supervised fine-tuning

Let $m_{t+1} = 1$ for assistant targets and 0 for user, system, tool, or padding targets. For examples with at least one selected target, optimize

$$\mathcal{L}_{\mathrm{SFT}}(\Theta) = -\mathbb{E}_x\left[\frac{1}{\sum_{t=1}^{S-1} m_{t+1}}\sum_{\substack{1\le t<S\\ m_{t+1}=1}}\log p_\Theta(x_{t+1} \mid x_{1:t})\right]. \tag{D.2}$$

The mask selects loss terms only. User, system, and tool tokens still update the decoder state, and gradients from assistant losses flow through those updates, including encoder memory and decoder KV. The model continues from the existing state when an assistant turn begins.

D.3 Policy gradients and replay

Let $c = x_{1:T}$ be a prompt and $y = (y_1, \ldots, y_N)$ a sampled response, with $y_i = x_{T+i}$. The policy is

$$\pi_\Theta(y \mid c) = \prod_{i=1}^{N} p_\Theta(y_i \mid c, y_{<i}). \tag{D.3}$$

For a sequence reward $R(c, y)$ independent of Θ, define $J(\Theta) = \mathbb{E}_{c,\, y\sim\pi_\Theta}[R(c, y)]$. Its on-policy score-function gradient is

$$\nabla_\Theta J = \mathbb{E}_{c,\, y\sim\pi_\Theta}\left[(R(c, y) - b(c))\sum_{i=1}^{N}\nabla_\Theta \log p_\Theta(y_i \mid c, y_{<i})\right], \tag{D.4}$$

where $b(c)$ is a response-independent baseline, held constant when taking the policy gradient. During replay, the sampled tokens are fixed and gradients pass through the recurrent computation used to evaluate their log-probabilities.

Figure 9 shows the replay procedure. The sampler records each action's log-probability under the distribution μ that sampled it. The trainer rebuilds encoder features, recurrent outputs, and

decoder SWA caches under the current parameters Θ, starting from the initial state and processing the full prompt and sampled response prefix. The resulting action probabilities give the ratios

$$r_i(\Theta) = \exp\left(\log p_\Theta(y_i \mid c, y_{<i}) - \log \mu(y_i \mid c, y_{<i})\right). \tag{D.5}$$

Here the target is the raw model policy p_Θ. The behavior probabilities must include any temperature scaling, truncation, and renormalization used by the sampler. Exact importance sampling requires $p_\Theta(\cdot \mid h) \ll \mu(\cdot \mid h)$: at each relevant history, every action with positive target probability must also have positive behavior probability. Top-k or top-p sampling generally excludes actions with positive probability under an untruncated softmax target. If the target policy itself uses a sampling transform, apply it consistently in the objective and ratios.

The trainer must preserve both the forward probabilities and their parameter derivatives (Zhang et al., 2026). It therefore differentiates the recurrent computation or an equivalent implementation.

For fixed prompts and sequence rewards, exact trajectory importance sampling uses $\prod_i r_i$, subject to support and integrability conditions. Tokenwise clipping and other surrogate losses may introduce bias even with correct replay probabilities.

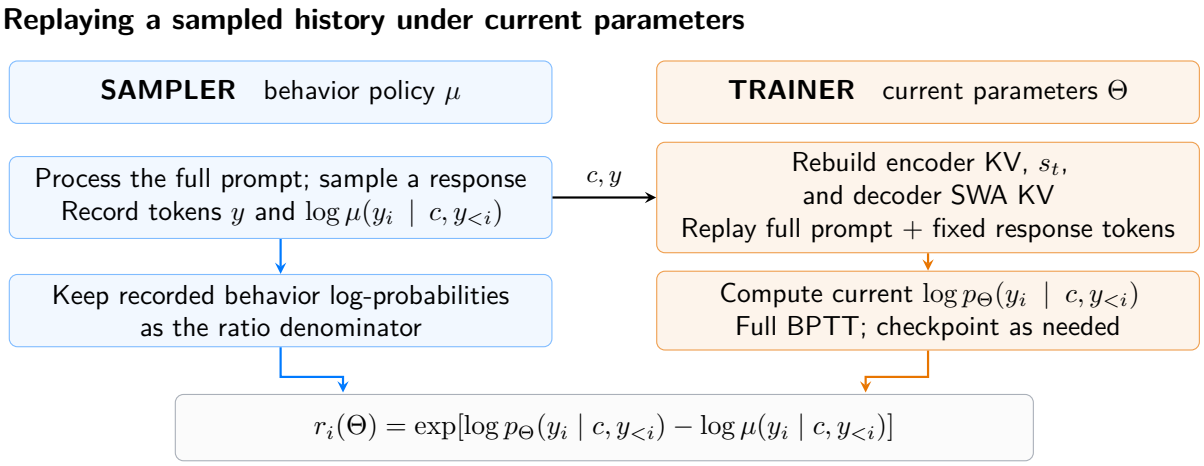

Figure 9 Replaying a sampled history under current parameters. The sampler records behavior probabilities. The trainer rebuilds the full prefix, including recurrent outputs and decoder SWA caches, to evaluate the current policy. The ratio compares the current action probability with the recorded behavior probability.

D.4 Gradients and cache reuse during training

Full BPTT follows gradients through encoder memory, recurrent outputs, and decoder SWA KV over the entire prompt and response. This remains true when only response tokens contribute to the loss. Recomputing the prompt under current parameters with `no_grad` gives the same forward values but drops the prompt-state derivatives.

The reference computation uses full BPTT, with activation checkpointing as needed. Checkpointing recomputes activations during backward using the same parameters and stochastic state. Truncated BPTT (TBPTT) instead detaches selected tensors at a boundary, keeping their values

while cutting their gradient paths. Detaching only s_t leaves paths through decoder KV and encoder memory, so a truncation scheme must state which tensors it detaches.

Keep parameters fixed throughout each replay and backward pass. After an optimizer update, rebuild encoder KV, recurrent outputs, and decoder SWA KV before the next exact replay. Recomputation may start from the sequence beginning or from an exact prefix checkpoint made with the current parameters and execution settings. For full BPTT, the checkpoint must retain its parameter dependencies in the computation graph or reconstruct them by recomputing the prefix during backward. The recorded behavior log-probabilities remain unchanged, since they describe the policy that sampled the actions.

E Multi-turn Inference and Cache Reuse

A conversation forms one token history beginning at BOS. User messages, tool results, role delimiters, and assistant tokens all receive encoder and decoder updates. For example, when a tool returns, encode its output using the cached prefix, then process those tokens through the decoder in order before resuming generation. Reset state only for a new independent sequence or an explicitly defined context reset.

To resume from a saved prefix, retain the encoder cache C_t^E, cross-attention memory $M_{\leq t}$, decoder state $H_t = (s_t, C_t^D)$, tokens and positions, SWA window convention, and model version. These values can be reused at fixed weights regardless of where the prompt ends. Reproducing the same sampled outputs also requires the sampler's random state. If only text is saved, replay it to rebuild the state.

Editing or truncating an earlier token invalidates the later cached state. Recompute from a valid checkpoint before the edit; this also applies to chat templates that rewrite earlier tokens. In multi-turn RL, user and tool tokens update the state and carry gradients, but receive no action importance-ratio factors because the policy did not sample them.

F Execution Procedures

The complete decoder state is $H_t = (s_t, C_t^D)$, with $H_0 = (s_\star, \varnothing)$. Encoder continuation state and encoder-derived cross-attention memory are maintained separately. All schedules use the block order and window convention in Equation (2.16).

F.1 Recurrent prompt prefill

1. Start an independent sequence with $x_1 = \mathrm{BOS}$ and $H_0 = (s_\star, \varnothing)$; initialize encoder caches and positions.
2. Encode $x_{1:T}$ causally in parallel and construct its encoder KV memory.
3. For $t = 1, \ldots, T$, compute $H_t = D_\phi(\mathrm{Merge}(e_t, s_{t-1}); M_{\le t}, C_{t-1}^D, t)$. Each decoder layer reads current KV and the retained SWA history. Both attention masks exclude positions after t.
4. Predict the first response token from s_T. Preserve H_T, encoder caches, memory, and positional metadata for the next update.

Each decoder update produces the KV needed by later updates, so this loop remains sequential even when the encoder runs in parallel.

F.2 RLT-0

RLT-0 removes feedback of the final decoder hidden state (Figure 12). It retains the causal encoder, encoder memory, decoder SWA, and prefix-restricted memory attention. The learned initial state $s_\star$, state normalization, gated merge, and feedback projection are removed. Each decoder position takes e_t as input in place of $\mathrm{Merge}(e_t, s_{t-1})$.

For known tokens, decoder positions run together within each layer under causal SWA and encoder-prefix masks. The resulting layerwise KV can be saved for incremental continuation. Generation remains sequential because future tokens are unknown. Full-sequence training uses ordinary backpropagation without a recurrent TBPTT boundary.

The control tests hidden-state feedback with the encoder–decoder split and attention structure held fixed.

F.3 Generation and new external inputs

Sample x_{t+1} from the distribution predicted by s_t, then consume it by updating the encoder cache and memory and advancing the complete decoder state to H_{t+1}. The resulting state predicts x_{t+2}. Every consumed token receives exactly one recurrent update and one KV insertion at every decoder SWA layer.

When a user or tool supplies multiple tokens, their encoder representations can be computed in a causal batch conditioned on the existing prefix. Decoder updates still occur in token order from the saved complete state. External tokens update both s_t and C_t^D.

If generation stops at a length limit before consuming its last emitted token, record that token separately. The cache represents the prefix already consumed; process the pending token once before any later token when resuming.

F.4 Pretraining and SFT

1. Compute $e_{1:S-1}$ using a causal encoder and construct memory.
2. Initialize $H_0 = (s_\star, \varnothing)$; unroll all positions $1, \ldots, S-1$, including every layerwise SWA cache update.
3. Accumulate all valid next-token losses for pretraining, or only assistant-target losses for SFT. All context tokens receive state updates.
4. Normalize each example by its number of selected targets, then average examples as in Equations (D.1) and (D.2). Backpropagate through the complete computation, using checkpointing if required.

Examples without selected targets are excluded from the loss average. Token averaging would give more weight to examples with more selected targets.

Packed independent sequences need separate recurrent states, SWA caches, encoder caches, positions, and attention masks. Both attention mechanisms must stay within document boundaries, even where losses are masked. Training cache operations must retain gradient dependencies or allow equivalent recomputation.

F.5 Current-policy RL replay

1. Read the rollout token history, action mask, actual behavior log-probabilities, and sampling metadata. Behavior probabilities include all sampling transforms and renormalization.
2. Hold current parameter values fixed throughout forward replay and backward. Use the target policy's positional, SWA, and execution conventions, with parameter gradients enabled.
3. Recompute encoder representations of the known history. From $H_0 = (s_\star, \varnothing)$, rebuild the recurrent output and every decoder SWA cache through all prompt tokens.
4. Replay subsequent tokens in order. Before consuming each sampled action, read its current-policy log-probability from the preceding state. Include EOS if sampled. Consume every intervening external token needed for later predictions.
5. Form the chosen RL loss from action log-probabilities, rewards or advantages, and any required behavior ratios. Apply action factors only to tokens sampled by the policy.
6. Backpropagate, update parameters, and invalidate parameter-dependent encoder and decoder caches before the next exact current-policy replay.

The action mask selects loss terms while preserving gradients through user and tool tokens. Replay all assistant turns with the external context between them. Prompt replay under `no_grad`, or detaching prompt KV, can keep forward values unchanged while dropping gradients required by full BPTT.

Importance weights must satisfy the support conditions described with Equation (D.5).

A deterministic forward pass, for example with dropout disabled, gives a reproducible probability for each action. For a stochastic policy, specify whether action probabilities condition on internal randomness or average over it. One replay evaluates a particular realization; marginal probabilities require averaging over that randomness.

G Causality and Gradient Paths

Proposition G.1 (Causality)**.** With a causal encoder, prefix-restricted encoder memory, and causal decoder SWA, $H_t = (s_t, C_t^D)$ depends only on $x_{1:t}$ and the parameters, including $s_\star$, under deterministic execution.

Proof. Encoder causality implies that each e_j depends only on $x_{1:j}$, hence $M_{\le t}$ depends only on $x_{1:t}$. The initial state contains no future-token information. Suppose H_{t-1} depends only on $x_{1:t-1}$. The next update uses this state, e_t, and $M_{\le t}$. At each decoder layer, current KV is formed from the causally available layer input, and SWA reads no position greater than t. Appending current KV and evicting old entries introduce no future information. Thus both s_t and C_t^D depend only on $x_{1:t}$, completing the induction. □

For teacher-forced encoder features held fixed, use a fixed-slot representation of the decoder cache, with validity masks during warm-up, and define

$$\mathcal{J}_t = \frac{\partial H_t}{\partial H_{t-1}}. \tag{G.1}$$

Then

$$\frac{\partial H_t}{\partial H_j} = \mathcal{J}_t \mathcal{J}_{t-1} \cdots \mathcal{J}_{j+1}, \qquad j < t, \tag{G.2}$$

where

$$\mathcal{J}_t = \begin{pmatrix} \dfrac{\partial s_t}{\partial s_{t-1}} & \dfrac{\partial s_t}{\partial C_{t-1}^D} \\ \dfrac{\partial C_t^D}{\partial s_{t-1}} & \dfrac{\partial C_t^D}{\partial C_{t-1}^D} \end{pmatrix}. \tag{G.3}$$

The block Jacobian includes paths through both the recurrent output and decoder KV.

Encoder memory retains information from past encoder outputs; decoder SWA gives direct access to recent decoder projections. Older decoder entries are evicted, but may still affect later computation through states or activations that read them before eviction.

Full parameter derivatives also include the parameter dependence of encoder features, memory, every transition, and decoder KV projections. Let B_T^E denote all encoder-side boundary tensors needed for response continuation, including encoder continuation KV and encoder-derived cross-attention memory. For a response loss $\ell(\Theta, H_T(\Theta), B_T^E(\Theta))$, the chain rule gives

$$\frac{d\ell}{d\Theta} = \frac{\partial \ell}{\partial \Theta} + \frac{\partial \ell}{\partial H_T}\frac{\partial H_T}{\partial \Theta} + \frac{\partial \ell}{\partial B_T^E}\frac{\partial B_T^E}{\partial \Theta}. \tag{G.4}$$

The first term holds the boundary arguments fixed; the other terms account for their prefix computation. In particular,

$$\frac{\partial \ell}{\partial H_T}\frac{\partial H_T}{\partial \Theta} = \frac{\partial \ell}{\partial s_T}\frac{\partial s_T}{\partial \Theta} + \frac{\partial \ell}{\partial C_T^D}\frac{\partial C_T^D}{\partial \Theta}. \tag{G.5}$$

Detaching s_T, decoder KV, or encoder boundary tensors removes the corresponding terms from the full gradient, even when forward probabilities stay unchanged.

H When Cached States Can Be Reused

A saved prefix can replace forward replay when it matches the current parameters, consumed tokens, positions, initialization, SWA window, and policy execution settings. Save the recurrent output, each decoder SWA cache, encoder continuation state, encoder memory, and associated metadata. Handle stochastic computation as specified by the policy.

A detached cache can hold correct values without the computation graph needed for training. Full BPTT requires retaining or recomputing the prefix graph so gradients reach every parameter-dependent boundary tensor.

Activation checkpointing within one update recomputes activations under the same parameters and stochastic state. Mutable cache implementations must restore the saved contents during recomputation and preserve activations that backward still needs.

Truncated BPTT instead preserves numeric values while cutting selected gradient dependencies. A complete decoder-state detach is

$$\widetilde{H}_t = (\mathrm{stopgrad}(s_t), \mathrm{stopgrad}(C_t^D)). \tag{H.1}$$

Detaching only s_t leaves possible paths through decoder KV; detaching only decoder KV leaves paths through the recurrent output. Encoder-side boundary tensors can provide additional paths across the same boundary.

SWA eviction removes old KV from future attention windows. Full BPTT still differentiates earlier computations that used those entries, so training may need more activation storage than the inference cache alone.

After a weight update, recompute the prefix values and gradient graph before the next exact training pass.

I RLT-2: Chunk-Parallel Hidden-State Feedback

RLT-2 holds the feedback state fixed within a chunk and updates it at the chunk boundary. Known tokens within that chunk can therefore run in parallel through each decoder layer. The causal encoder, decoder SWA, prefix-restricted encoder-memory attention, gated merge, and readout use the same components as RLT-1. This section specifies the proposed architecture and its execution costs; the experiments in Section 3 evaluate RLT-1 and the Transformer.

I.1 Chunk transition and causal masks

Choose a fixed chunk size $B \geq 1$, count BOS as position one, and anchor boundaries to the start of each independent sequence. For a known prefix of length T, define

$$K = \lceil T/B \rceil, \qquad a_k = (k-1)B + 1, \qquad b_k = \min(kB, T), \qquad I_k = \{a_k, \ldots, b_k\}. \tag{I.1}$$

Initialize the feedback register $h_0 = s_\star$ and empty decoder caches. Every position in chunk k receives the same preceding boundary state, with a position-dependent gate:

$$r_{k-1} = \mathrm{RMSNorm}_s(h_{k-1}), \qquad v_{k-1} = W_s r_{k-1}, \tag{I.2}$$

$$g_t = \sigma\big(W_g[e_t; r_{k-1}] + b_g\big), \tag{I.3}$$

$$z_t^0 = e_t + \alpha g_t \odot v_{k-1}, \qquad t \in I_k, \tag{I.4}$$

$$y_{I_k}, C_{b_k}^D = D_\phi^{\mathrm{chunk}}(z_{I_k}^0; M, C_{a_k-1}^D, I_k), \tag{I.5}$$

$$p_\Theta(x_{t+1} \mid x_{1:t}) = \mathrm{softmax}\big(W_o\,\mathrm{RMSNorm}_o(y_t)\big)_{x_{t+1}}, \tag{I.6}$$

$$h_k = y_{kB} \quad \text{when } kB \le T. \tag{I.7}$$

Here $y_t = z_t^{L_D}$ is the final decoder output at every position; only the last output of a complete chunk becomes the next feedback state. The chunk operator applies Equation (2.16) layer by layer, using all known positions in I_k together. At each layer, query t reads decoder keys at $\max(1, t-W+1) \le j \le t$, including retained keys from earlier chunks, and encoder memory only at $j \le t$. Neither SWA nor encoder positions reset at a chunk boundary. The chunk may be longer or shorter than the SWA window.

All positions of a decoder layer project their KV from that layer's inputs before masked attention, so causal SWA requires no serial loop over positions within the layer. The feedback input h_{k-1} depends only on tokens before a_k; causal encoder and decoder masks then ensure that y_t depends only on $x_{1:t}$. The last-position output summarizes the chunk through these causal computations; there is no pooling or additional summary token. With $B = 1$, the transition reduces to RLT-1 at identical weights and execution conventions. A sequence fitting in one chunk still receives the learned initial-state merge, whereas RLT-0 removes that merge and its parameters.

I.2 Training and prefill algorithm

For teacher-forced training, $x_{1:T}$ denotes the consumed input tokens, with a next-token target wherever one is available. The same forward procedure computes prompt prefill and known-history policy replay:

1. Encode the known tokens causally and build encoder KV memory. Initialize $h = s_\star$, empty decoder SWA caches, and global position one.
2. Visit chunks in increasing order. Compute $\mathrm{RMSNorm}_s(h)$ and $W_s\,\mathrm{RMSNorm}_s(h)$ once for the chunk; broadcast them to its token-specific gated merges.
3. For each decoder layer, project the chunk's Q/K/V together, apply causal SWA using retained and current-chunk KV, then apply prefix-masked memory attention and the FFN. Retain the last $W-1$ decoder KV entries per layer for continuation.
4. Compute logits and the selected next-token losses at every position. After a complete chunk, set h to its last decoder output. A partial final chunk leaves h at the preceding complete boundary.
5. For training, accumulate the sequence loss and backpropagate through all chunks before updating parameters. For prefill, save caches, h, the consumed-token count, and the last output or next-token logits.

Full BPTT differentiates through the boundary state, cross-chunk decoder KV, and encoder memory. Chunking changes the forward dependency graph; it does not imply a gradient detach or an optimizer update per chunk. An optional TBPTT scheme must specify which of these paths it truncates.

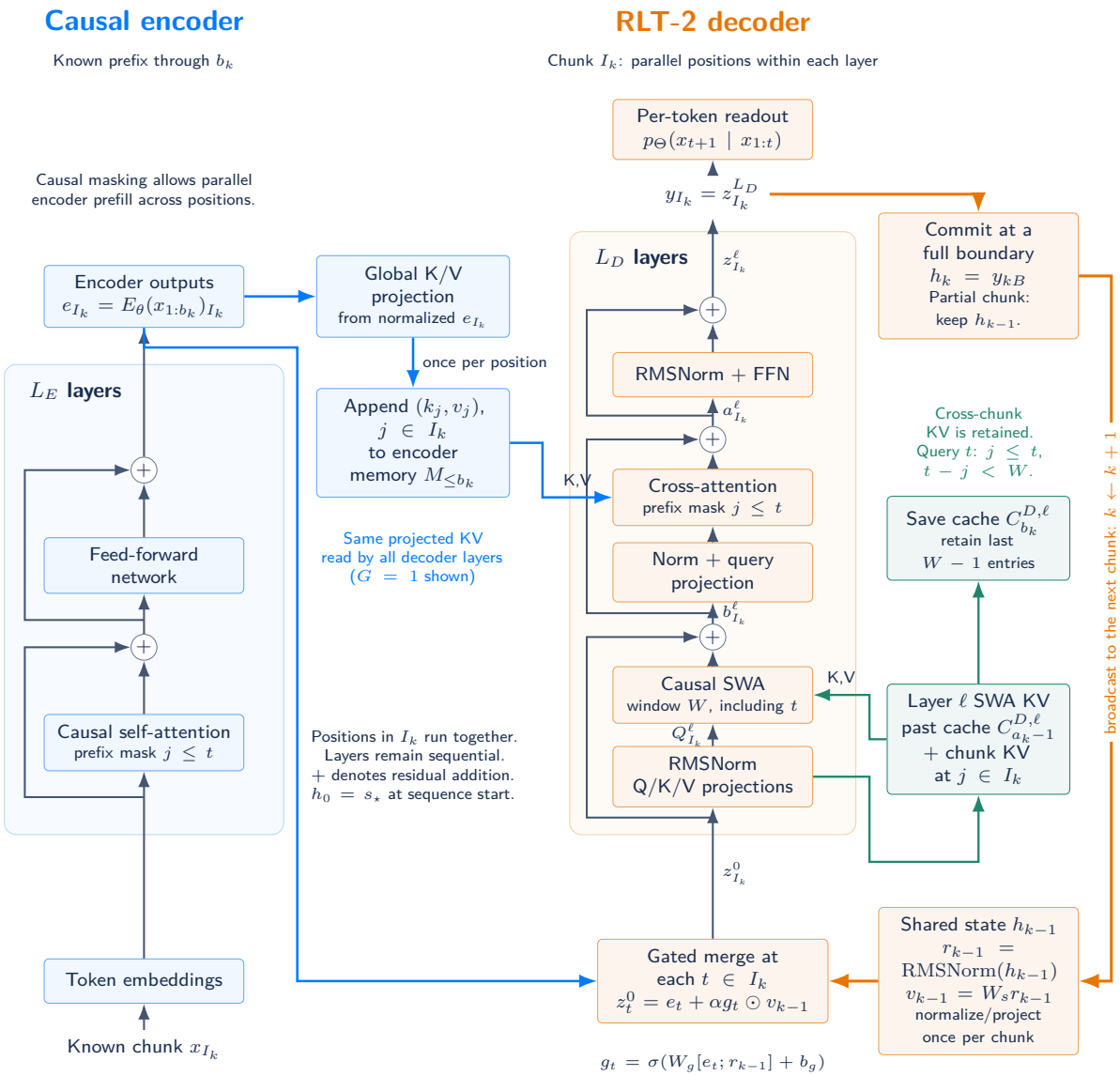

Figure 10 RLT-2 architecture for one chunk. Computation flows upward; the encoder and decoder stacks repeat for their indicated depths. Each position $t \in I_k$ merges its encoder output with the same preceding boundary state h_{k-1}, using its own gate. State normalization and projection are shared across the chunk. Each decoder layer applies causal SWA, prefix-restricted encoder-memory attention, and an FFN, with residual connections around each sublayer. For $G = 1$, all decoder layers read the same projected encoder KV; each layer retains its own SWA KV across chunk boundaries. Every output y_t predicts the next token, while only y_{kB} at a complete boundary becomes the next feedback state. A partial chunk preserves h_{k-1} during continuation. Blue arrows carry encoder features and memory, green arrows carry decoder KV, and orange arrows carry chunk-level feedback.

For RL, replay uses the same B, boundary anchor, masks, and positions as sampling, rebuilding the state under the current parameters before evaluating action probabilities.

I.3 Incremental generation and partial chunks

Let t be the number of consumed tokens and $r = t \bmod B$ the current chunk offset. The saved feedback register is $h_{\lfloor t/B \rfloor}$, including when the prompt ends inside a chunk. To consume the next observed or sampled token at position $j = t + 1$:

1. Update the causal encoder and append its memory KV. Merge e_j with the saved boundary state, keeping that state fixed throughout the chunk.
2. Run one decoder step with the usual causal SWA caches and prefix memory, producing y_j, updated KV caches, and next-token logits.
3. If $j \bmod B = 0$, replace the feedback register by y_j; otherwise preserve it. Advance the consumed-token count to j.

The latest y_j always drives the next-token prediction, even when it is not committed to the feedback register. For example, with $B = 4$ and a six-token prompt, tokens 5–8 all merge with $h_1 = y_4$; processing tokens 5 and 6 during prefill does not replace that register with y_6. The output y_6 predicts token 7, and y_8 becomes h_2 only after token 8 is consumed. Known continuation tokens can be batched up to the next fixed boundary, then the procedure continues with the new state. Padding, message boundaries, and the prompt–response split do not commit a partial chunk. Independent packed sequences each maintain their own boundary anchor and state.

With fixed weights and exact arithmetic, batched and incremental execution agree when both schedules use the same mathematical operators and disable stochastic operations or assign identical random draws to corresponding operations. Each position then has the same encoder prefix, preceding boundary state, and causally available layerwise KV in both schedules. Induction over layers within a chunk and then over chunks gives the same outputs and boundary states. Changing B or shifting the boundary anchor changes the model computation and invalidates a saved continuation state.

Chunkwise prefill and tokenwise decoding can use different matrix and attention kernels, with different floating-point reduction orders. These changes can alter logits and boundary states even under deterministic execution. For RL, compare chunkwise replay against a tokenwise reference that reproduces the rollout's encoder, decoder, and readout execution, including precision, batch layout, kernel choices, and stochastic settings. Report selected-action log-probability differences at unchanged parameters before attributing replay differences to a policy update.

I.4 Training and inference efficiency

Table 3 compares equal encoder and decoder depths, width, windows, and memory groups, with $L_D \geq 1$. Let T denote known input tokens, $K = \lceil T/B \rceil$, and N newly consumed generation tokens. The decoder block-stage counts describe sequential dependencies in the specified layerwise schedules; they exclude attention-kernel reductions, communication, and hardware-dependent latency. All three variants also require the common L_E-stage causal encoder prefill.

At fixed dimensions, the common forward arithmetic with dense encoder and memory attention is

$$O\big((L_E + L_D)Td^2 + (L_E + L_D)T^2d + GTd^2 + L_DT\min(W, T)d\big). \tag{I.8}$$

Quantity	RLT-0	RLT-1	RLT-2 (B)
Positions per decoder batch, known tokens	T	1	up to B
Decoder block stages, training forward / prefill	L_D	TL_D	KL_D
Backward dependency depth, full BPTT	$O(L_D)$	$O(TL_D)$	$O(KL_D)$
Final-state feedback updates over T tokens	0	T	$\lfloor T/B \rfloor$
Decoder block evaluations over T tokens	TL_D	TL_D	TL_D
Merge gate evaluations over T tokens	0	T	T
Shared feedback projections, known tokens	0	T	K
Autoregressive token steps for N tokens	N	N	N
Blocks per consumed generation token	$L_E + L_D$	$L_E + L_D$	$L_E + L_D$
Persistent feedback register	none	d values	d values

Table 3 Execution costs at matched stack dimensions. Counts omit common readout and encoder-memory projection work. RLT-2 caches the normalized and projected boundary state for reuse; the gate still depends on each token. Backward counts cover the decoder dependency graph, with the encoder's backward pass common to all variants. RLT-0 also omits the learned initial state and merge parameters. Counts describe dependencies and arithmetic; wall-clock speed depends on the implementation and hardware.

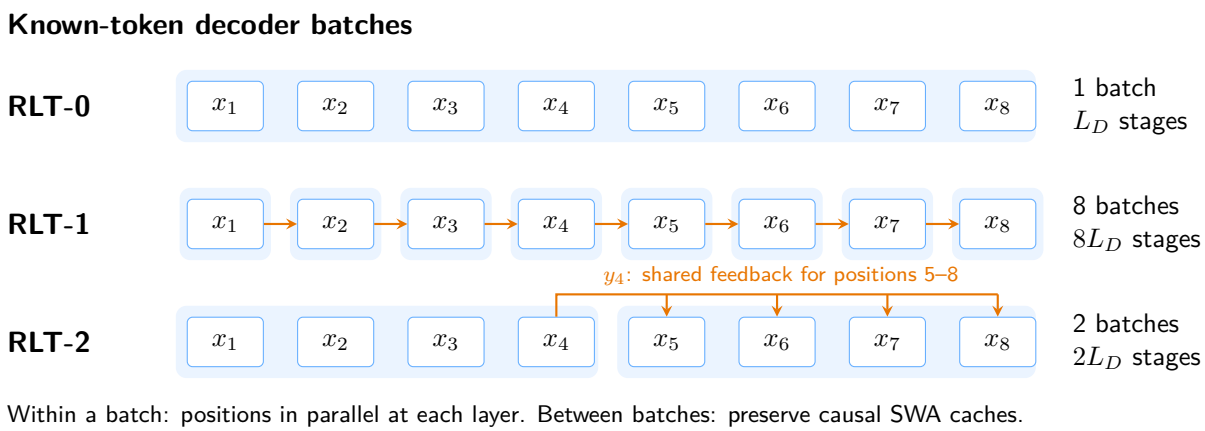

Figure 11 Decoder scheduling for eight known tokens, with $B = 4$ for RLT-2. Each shaded group is one layerwise decoder batch, containing L_D sequential blocks. RLT-0 batches all eight positions, RLT-1 uses eight single-position batches, and RLT-2 uses two four-position batches. Orange arrows show feedback dependencies; RLT-2 broadcasts y_4 to all four positions in its second chunk. Causal SWA applies in every row, with caches retained between batches. During autoregressive generation, all three rows consume unknown future tokens one at a time.

RLT-1 adds $O(Td^2)$ merge work. RLT-2 retains T token-specific gates and therefore also adds $O(Td^2)$ merge work, while reusing the state normalization and projection across each chunk. Its K sequential decoder batches permit larger matrix operations and weight reuse across up to B positions. Total decoder arithmetic still covers all T tokens, and full BPTT requires activations or recomputation across all chunks.

For generation at context length t, each variant still evaluates its encoder and decoder stacks per token and reads the available encoder memory. RLT-2 can reuse its feedback projection until the next boundary; the token-specific gate, decoder blocks, and KV writes remain per-token operations. The shared inference-cache scaling is

$$O\big((L_E + G)t\, d_{\mathrm{KV}} + L_D \min(t, W-1) d^D_{\mathrm{KV}}\big). \tag{I.9}$$

RLT-1 and RLT-2 add $O(d)$ feedback storage; RLT-2 also retains a chunk offset and may cache $O(d)$ normalized/projected state. Chunk batches require temporary activations and current-chunk KV during prefill. Increasing B shortens the feedback path from T token updates to K chunk evaluations, trading feedback frequency for known-token parallelism.

A matched benchmark should sweep $B \in \{1, 2, 4, 8, 16, 32\}$ and RLT-0 at fixed data, stack dimensions, precision, optimizer, batch size, and attention kernels. Report task accuracy alongside training tokens/s (forward, backward, and optimizer), prefill latency versus prompt length, time per generated token versus cached context length, and peak device memory. Use device synchronization and warmup, report hardware and batch sizes, and separate prefill from generation timing. The $B = 1$ run checks RLT-1 equivalence, and future-token perturbations check causality. At unchanged parameters, measure maximum and RMS differences in logits, selected-action log-probabilities, and boundary states across chunkwise, partial-chunk, and tokenwise execution, recording dtype and execution settings. A differentiable reference should also compare full-BPTT parameter gradients, including paths through boundary states, decoder KV, and encoder memory.

J Experimental Details

J.1 Parameter counts

Table 4 lists unique trainable parameters. Each RLT-1 decoder block has both SWA and encoder-memory cross-attention; each Transformer block has one attention sublayer. RLT-1 $8 + 0$ applies the learned state projection and gated merge at every token.

Table 4 Unique trainable parameters in the eight-layer models. Each model has the same size across all six tasks.

Model	Parameters (M)	Relative to GPT 8
RLT-1 4+4	28.73	1.135×
RLT-1 5+3	28.20	1.114×
RLT-1 6+2	27.68	1.093×
RLT-1 7+1	27.15	1.073×
RLT-1 8+0	26.10	1.031×
Transformer 8	25.31	1.000×

J.2 Implementation of RLT-0

The `rlt_parallel_tiny` control was merged in `nanogptpro-dev` PR #164. It uses an untied $4 + 4$ encoder–decoder split, width 512, four attention heads, FFN width 1,365, vocabulary capacity 277, an SWA window of eight, one encoder-memory group, and width-μP. It has 27,938,944 unique parameters.

The control removes the learned initial state, state normalization, merge gate, and feedback projection from RLT-1 $4+4$. It retains causal encoder attention, decoder SWA, and prefix-only attention to encoder memory. Known positions run in parallel within each decoder layer, and training uses full backpropagation without TBPTT. Generation maintains KV caches and emits tokens sequentially, without feeding the previous final hidden state into the next decoder input. The two models use separate checkpoint formats identified by their architecture markers.

J.3 Seed aggregation and metric definitions

Let $a_{m,s,t}$ be validation accuracy for model m, initialization seed s, and optimizer step t. For every task we report

$$\bar{a}_{m,t} = \frac{1}{3} \sum_{s \in \{42,43,44\}} a_{m,s,t}, \qquad \mathrm{SD}_{m,t} = \sqrt{\frac{1}{2} \sum_{s \in \{42,43,44\}} (a_{m,s,t} - \bar{a}_{m,t})^2}. \tag{J.1}$$

Every aggregate uses all three seeds at the same step, without interpolation or missing-seed imputation. All curves extend through step 2,000, the common comparison step in Table 1. The SD describes variability across model initializations on a fixed training stream and held-out set.

Parity and mod-5 supervise one final label per example. Their three equally sized validation groups are pooled within a run before averaging across seeds. S_5 supervises each prefix; final-state accuracy counts the last state in each sequence, while prefix-token accuracy pools all supervised positions. Addition uses teacher forcing on the full prompt and correct preceding answer tokens. Its denominator includes all answer tokens—digits, spaces, brackets, and EOS—and excludes prompt and padding positions. Addition loss is teacher-forced cross-entropy normalized per example.

Table 5 Final-state and prefix-token accuracy (%) on the fixed S_5 validation sets at step 2,000, reported as mean $\pm$ sample SD over seeds 42, 43, and 44.

Task	Metric	RLT-1 4+4	RLT-1 5+3	RLT-1 6+2	RLT-1 7+1	RLT-1 8+0	GPT 8
S_5, swaps	Final state	100.00 ± 0.00	100.00 ± 0.00	100.00 ± 0.00	99.35 ± 0.81	99.61 ± 0.68	99.09 ± 0.23
S_5, swaps	Prefix token	100.00 ± 0.00	100.00 ± 0.00	100.00 ± 0.00	99.96 ± 0.05	99.95 ± 0.06	99.91 ± 0.06
S_5, standard	Final state	0.78 ± 0.68	2.47 ± 1.26	2.21 ± 1.48	2.08 ± 0.98	1.82 ± 1.19	0.52 ± 0.23
S_5, standard	Prefix token	5.57 ± 2.11	7.82 ± 0.55	7.99 ± 0.29	7.67 ± 0.54	6.47 ± 0.65	5.40 ± 1.21

J.4 Reproducing the completed training snapshot

The formal tasks use data seeds 20260914 for training, 20260915 for validation, and 20260916 for test sets. Addition uses data seed 42. The three initialization seeds share the same indexed training stream and held-out examples for each task. All 108 runs completed 2,000 steps using the implementation based on upstream commit `d1a4516`. Native histories were frozen on September 17, 2026; the manifest records exact collection timestamps and source hashes.

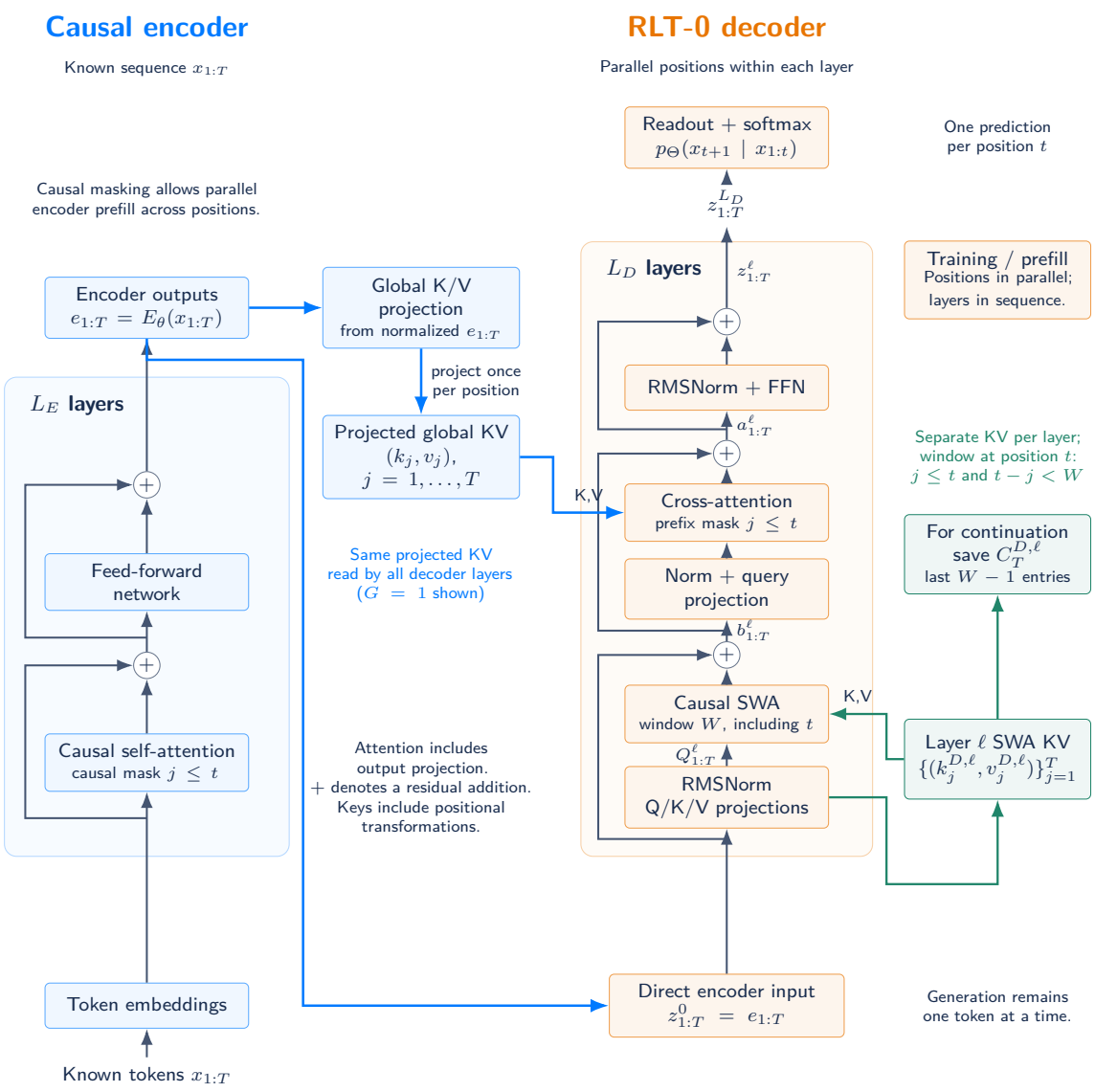

Figure 12 RLT-0 during training or prompt prefill. The decoder receives the encoder outputs directly, $z^0_{1:T} = e_{1:T}$, with the gated merge and final-hidden-state feedback path removed. Each outlined stack repeats for its indicated depth. For $G = 1$, all decoder layers read the same projected global KV; query position t can read encoder positions $j \le t$. Each decoder layer constructs its own SWA KV and applies a causal window of size W, including the current position. Known positions run together within each layer, while layers follow depth order. Incremental generation retains the encoder cache, global memory, and each layer's last $W - 1$ SWA entries and proceeds one token at a time.

The files `manifest.json` and `configs.json` in `data/experiments-depth8/` record run identifiers, settings, and source/corpus fingerprints. The collection contains 2,160 validation records and 216,000 unsmoothed training-step records. The script `plot_depth8.py` in `scripts/` regenerates the training figures and tables from these frozen records. The fixed-length token-accuracy export contains 1,800 validation records from 90 completed formal-task runs. Its plotting script is `plot_token_accuracy.py` in the same directory.

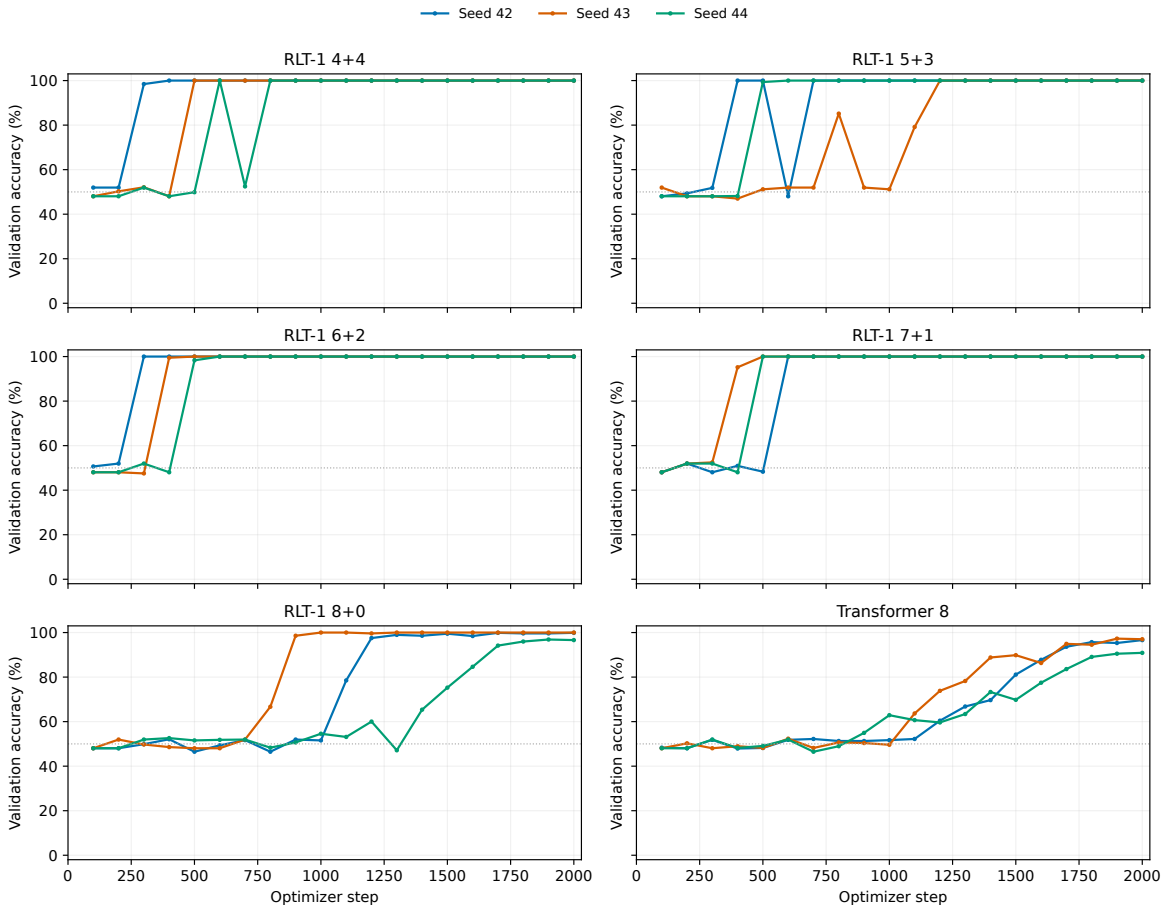

Figure 13 Parity accuracy for each initialization seed through 2,000 steps. All runs share the indexed training stream and validation set; curves retain temporary regressions and differences in learning speed.

J.5 Length-generalization protocol and supplementary metrics

The study uses all 108 completed runs, with seeds 42, 43, and 44 for every architecture and task. For each run, we minimize native ID-validation example loss, pooled over the configured validation lengths for formal tasks, and select the earliest step on ties. Table 6 lists the selected steps; checkpoint and training-result hashes accompany the data. Test results do not participate in selection.

The addition grid is 9, 10, 11, 12, 14, 16, and 32 digits in each operand. Parity and bracketed mod-5 use lengths 32, 40, 48, 64, 96, 128, 192, and 256; both S_5 tasks use 32, 48, 64, 96, 128, 192, 256, and 512. Flat expressions alternate operands and binary operators, so each requested even

length L maps to actual expression length $L-1$: 31, 39, 47, 63, 95, 127, 191, and 255. Formal-task lengths exclude BOS and any final equals sign; flat mod-5 has $L+1$ input tokens after these markers are added. There are 47 shared test sets and 846 model–seed–length evaluations.

Addition uses 256 unique unordered operand pairs per length, sampled from the native test partition with seed 20260916. Both operands have exactly the indicated width and no leading zero; reversed-digit serialization matches training. The evaluation computes teacher-forced answer-token accuracy without autoregressive generation. Each formal-task set contains 1,024 sequences from the original generator and test seed 20260916. All models and initializations use the same examples at a task and length. The shared holdouts are unchanged from the previous benchmark, and newly generated formal sets exclude duplicates and existing validation/test examples.

We reuse 697 model–seed–length results after matching selected checkpoints and holdout fingerprints, and evaluate the remaining 149 cells. All reported metrics are recomputed from saved per-example predictions or teacher-token counts. Evaluation uses CPU FP32, four intra-op threads, and the recorded training runtime; checkpoint weights remain fixed. New evaluations use batch 32, and reused native formal evaluations used batch four. The previous batch-size check matched all 32,768 state predictions across these batch sizes for 32 length-512 sequences per S_5 task with RLT-1 $4+4$. The teacher-only arithmetic evaluator also reproduces the saved per-example teacher-token counts for RLT-1 $4+4$ and Transformer 8 on 32 shared 32-digit pairs.

At each test length, we first compute a score for each initialization and then apply the mean/SD formula above with length in place of training step. Error bars show sample SD across the three initializations on shared test examples. Figure 14 reports supplementary token accuracy for all six tasks; the three S_5 scoring rules are compared in Section 3.5 (Figure 5).

The directory `data/experiments-depth8-generalization/` contains 846 per-seed rows, 282 aggregates, checkpoint/dataset fingerprints, and a verification report. The portable script `scripts/plot_generalization.py` checks complete seed/length coverage, count-derived scores, sample SDs, and checkpoint correspondence with the training snapshot before regenerating the figures and tables.

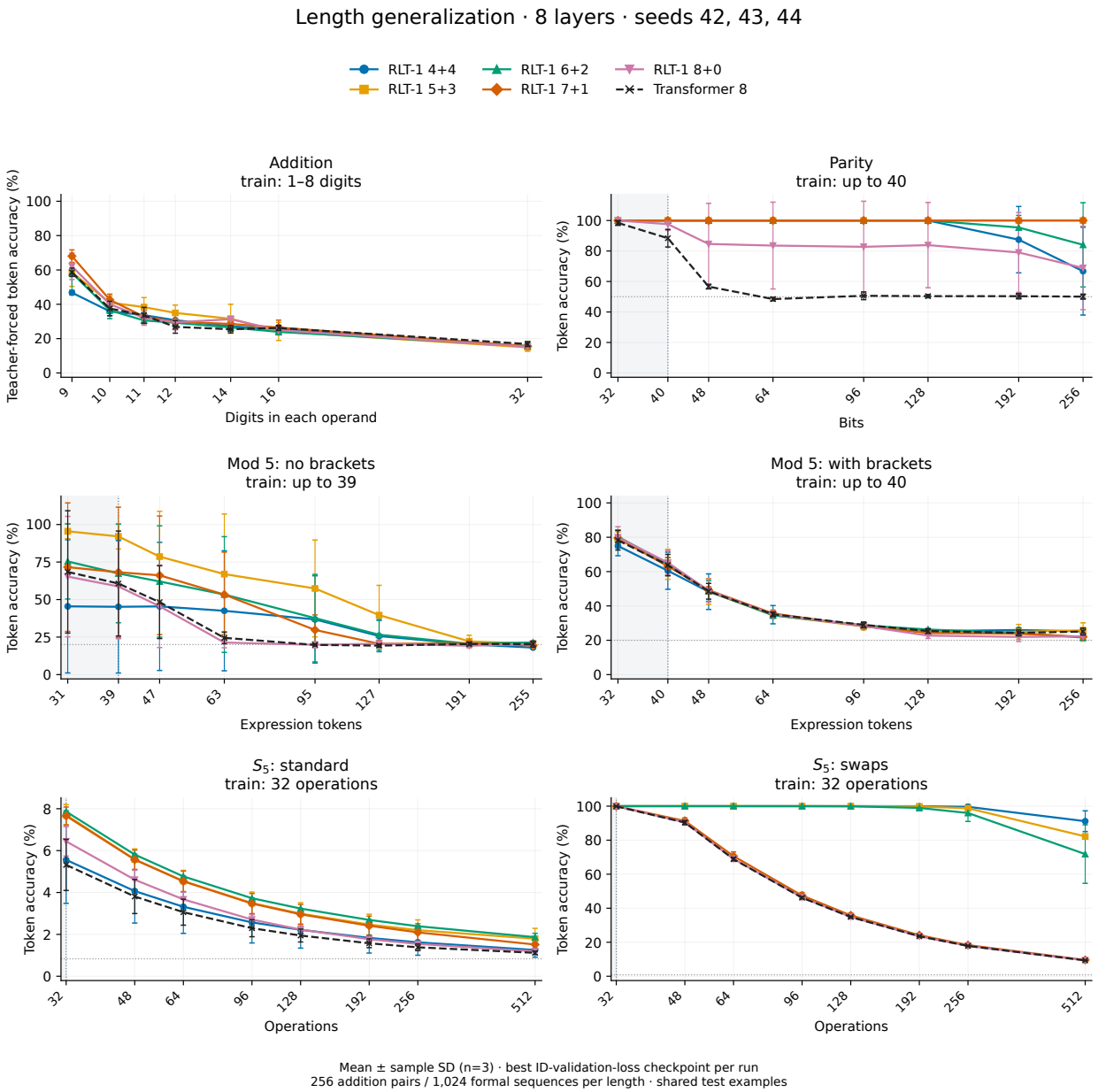

Figure 14 Token accuracy across test lengths, three initialization seeds. The checkpoints and shared examples are those of Figure 4. Addition uses teacher forcing on answer tokens; S_5 scores every prefix state; parity and mod-5 each score one final label. Points and error bars show mean ± sample SD across seeds 42, 43, and 44. Gray regions mark trained lengths.

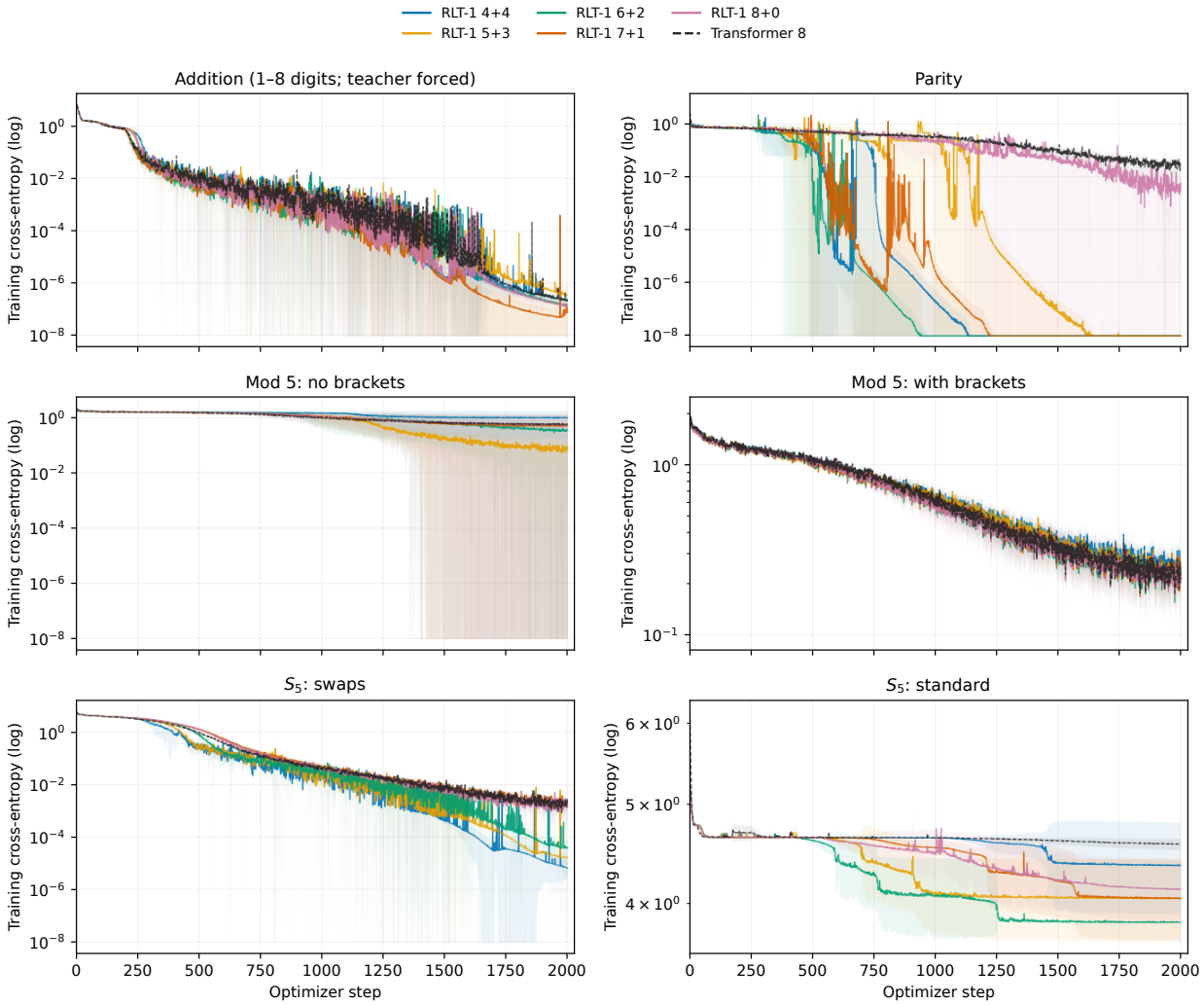

Figure 15 Unsmoothed global-batch training cross-entropy, shown as mean ± sample SD across three seeds for every task. Means and band limits below 10^{-8} are displayed at 10^{-8} on the logarithmic axis. Loss is measured before the optimizer update; logged gradient norms are measured before clipping. Compare losses within each task, since supervision differs across tasks.

Table 6 Optimizer steps selected by minimum in-distribution validation loss for length generalization. Every source run completed 2,000 steps; seeds 42, 43, and 44 are listed separately.

Task	Seed	RLT-1 4+4	RLT-1 5+3	RLT-1 6+2	RLT-1 7+1	RLT-1 8+0	GPT 8
Addition	42	2000	2000	2000	2000	2000	2000
	43	2000	2000	2000	2000	2000	2000
	44	2000	2000	2000	1900	2000	2000
Parity	42	900	1600	800	1500	2000	2000
	43	1100	2000	900	1500	1900	1900
	44	1400	1000	1200	1100	2000	2000
Mod 5, no brackets	42	800	2000	2000	2000	2000	400
	43	800	2000	2000	800	400	2000
	44	2000	2000	2000	2000	2000	1900
Mod 5, brackets	42	2000	2000	2000	2000	2000	1900
	43	2000	2000	2000	2000	2000	2000
	44	2000	2000	2000	2000	2000	2000
S_5, standard	42	1800	2000	2000	2000	2000	2000
	43	2000	2000	2000	2000	2000	2000
	44	2000	2000	1800	2000	2000	2000
S_5, swaps	42	2000	2000	2000	1900	2000	2000
	43	2000	2000	2000	2000	2000	1900
	44	2000	2000	2000	1900	2000	2000

J.6 Token-accuracy trajectories at lengths 32 and 33

Figure 16 compares token accuracy at length 32 for parity, bracketed mod-5, and both S_5 tasks, and at length 33 for flat mod-5. Flat expressions alternate numbers and binary operators, so their payload lengths are odd; 33 is the nearest configured validation length to 32. The completed snapshot contains 1,800 validation checkpoints from 90 runs: six architectures, five formal tasks, and three seeds, each trained for 2,000 steps. At length 32, S_5 token accuracy scores 8,192 prefix states across 256 sequences; parity and mod-5 each score one final label per sequence.

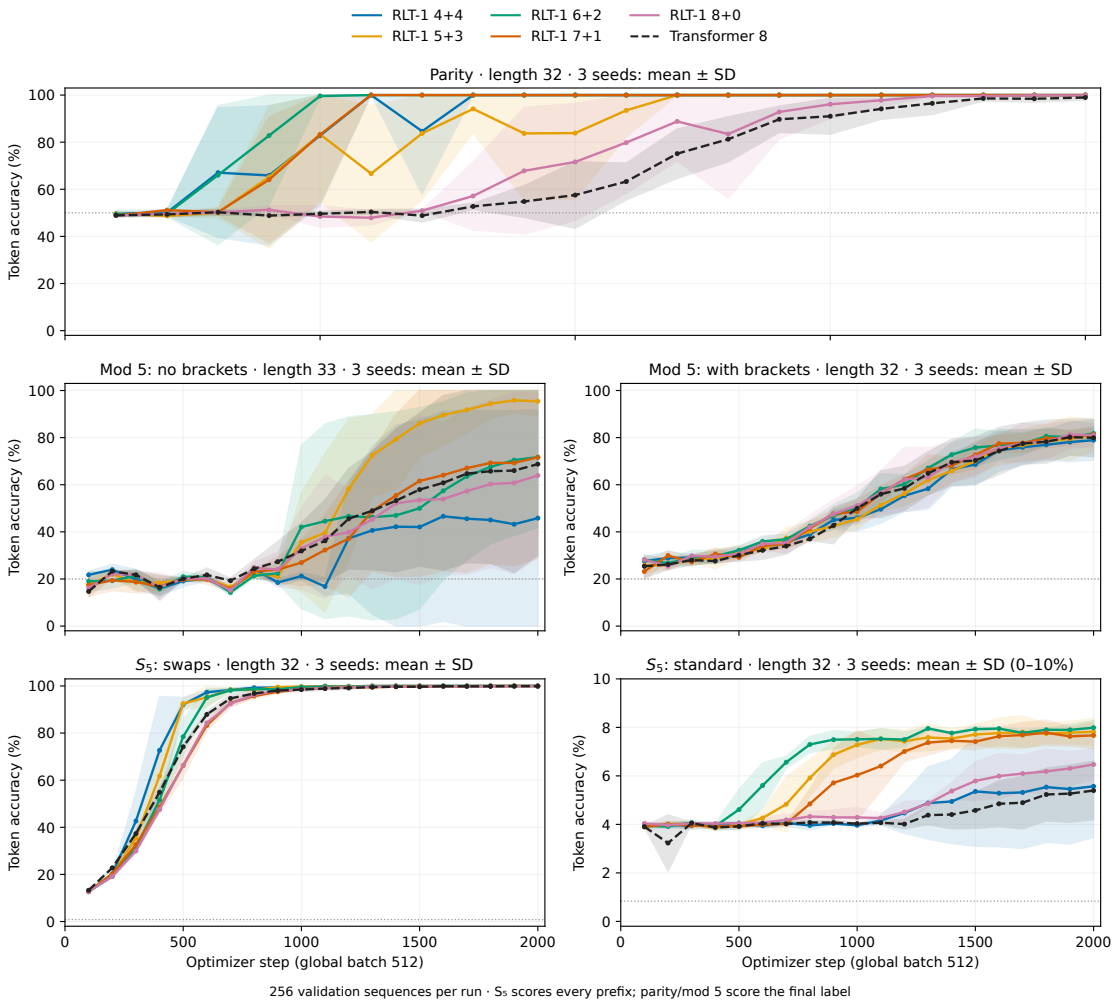

Figure 16 Validation token accuracy on five formal tasks, three seeds per task. Flat mod-5 uses payload length 33; all other panels use length 32. Every point uses all three initialization seeds at the same optimizer step; bands show mean ± sample SD, clipped to the accuracy range. S_5 scores all prefix states; parity and mod-5 score the final label. All curves extend through 2,000 steps. Standard S_5 uses a narrower vertical scale; horizontal dotted lines show uniform-prediction accuracy.

J.7 Sixteen-layer addition generalization

We separately evaluate RLT-1 $8+8$, $9+7$, $10+6$, $11+5$, $12+4$, $13+3$, $14+2$, $15+1$, and $16+0$, together with a sixteen-layer Transformer, using initialization seed 42. The models use the same mixed 1–8-digit training corpus, width 512, FFN width 1,365, four heads, optimizer settings, global batch 512, and microbatch 32 as the eight-layer addition runs; RLT-1 uses TBPTT 128. The sixteen-layer RLT-1 layouts have 51.27–56.00M parameters, and the Transformer has 50.49M. All ten models are evaluated on exactly the same 256 operand pairs at each of the seven widths in Appendix J.5, giving 70 additional model–length evaluations. We evaluate in CPU FP32 and report both teacher-forced token accuracy and greedy exact-answer accuracy, which requires the correct answer followed by EOS. Pointwise 95% Wilson intervals describe exact-answer uncertainty across test pairs.

Checkpoint selection minimizes ID validation example loss among checkpoints available at the September 16, 2026, 15:32:01 UTC snapshot, with earliest-step tie breaking. Four runs were unfinished: $8+8$, $9+7$, $10+6$, and $11+5$ had reached steps 1,200, 1,400, 1,500, and 1,800; their selected checkpoints were at steps 1,200, 1,300, 1,400, and 1,800, respectively. The other six runs had completed 2,000 steps and selected their final checkpoint. All ten selected checkpoints have 100% exact accuracy on the native mixed 1–8-digit validation set. The figures mark unfinished runs with a dagger and give the selected step in each legend entry.

Figure 17 shows that every RLT-1 split produces zero exact answers out of 256 pairs at every tested width from nine to 32 digits. The sixteen-layer Transformer produces three exact answers at nine digits (1.17%) and none at any longer tested width. Teacher-forced answer-token accuracy is higher: at nine digits, the Transformer reaches 84.51%, versus 47.32–65.82% across the RLT-1 splits. At 32 digits, all ten models lie between 13.95% and 16.21% on this metric. The gap between token and exact-answer accuracy reflects the difficulty of generating an entire correct answer beyond the training range.

Figure 18 compares $8+8$ with $4+4$ and includes the Transformer baseline at each depth. RLT-1 $8+8$ has 51.48% teacher-forced token accuracy at nine digits, versus 47.17% for $4+4$, but is lower at each tested width from ten through 32 digits. Training budgets differ: the $8+8$ checkpoint is at step 1,200, whereas $4+4$ and both Transformers are at step 2,000.

The frozen data in `data/arithmetic-generalization-depth16/` contain all 70 new measurements, 42 reverified eight-layer addition measurements, checkpoint identities, shared holdout hashes, and the prediction-level verification report. The script `scripts/plot_arithmetic_depth16.py` rebuilds these two standalone figures.

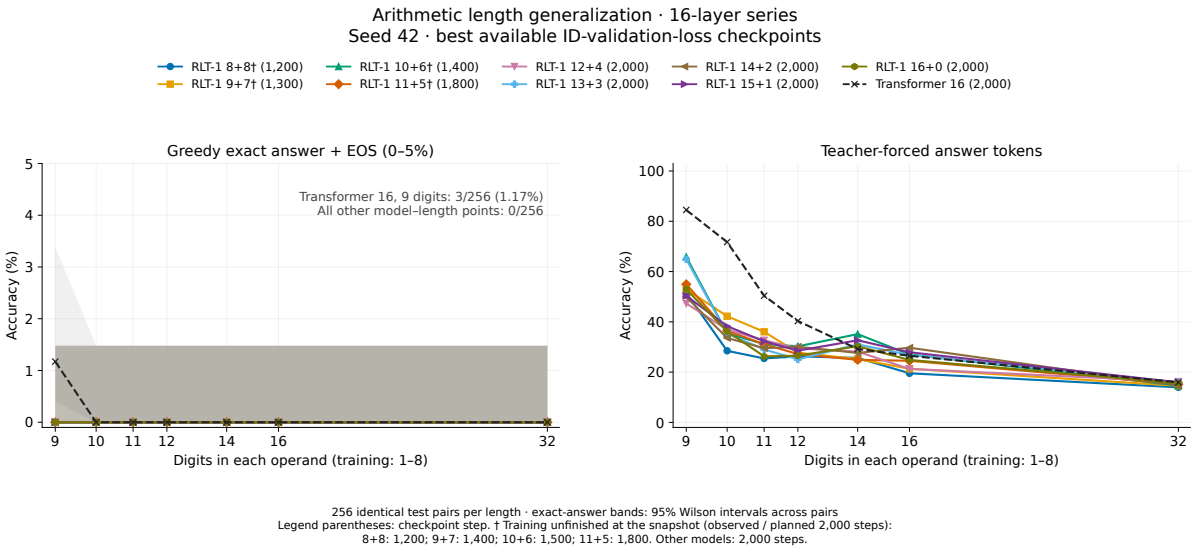

Figure 17 Addition generalization for the sixteen-layer series, seed 42. Nine RLT-1 depth splits and a sixteen-layer Transformer share the seven 256-pair holdouts used in the eight-layer study. The left panel uses a 0–5% scale for strict greedy answer-plus-EOS accuracy; the right panel reports teacher-forced answer-token accuracy on a 0–100% scale. Exact-answer shading shows pointwise 95% Wilson intervals across test pairs. Legend parentheses give the best available ID-loss checkpoint step; daggers mark the four runs that had not reached 2,000 training steps at the snapshot.

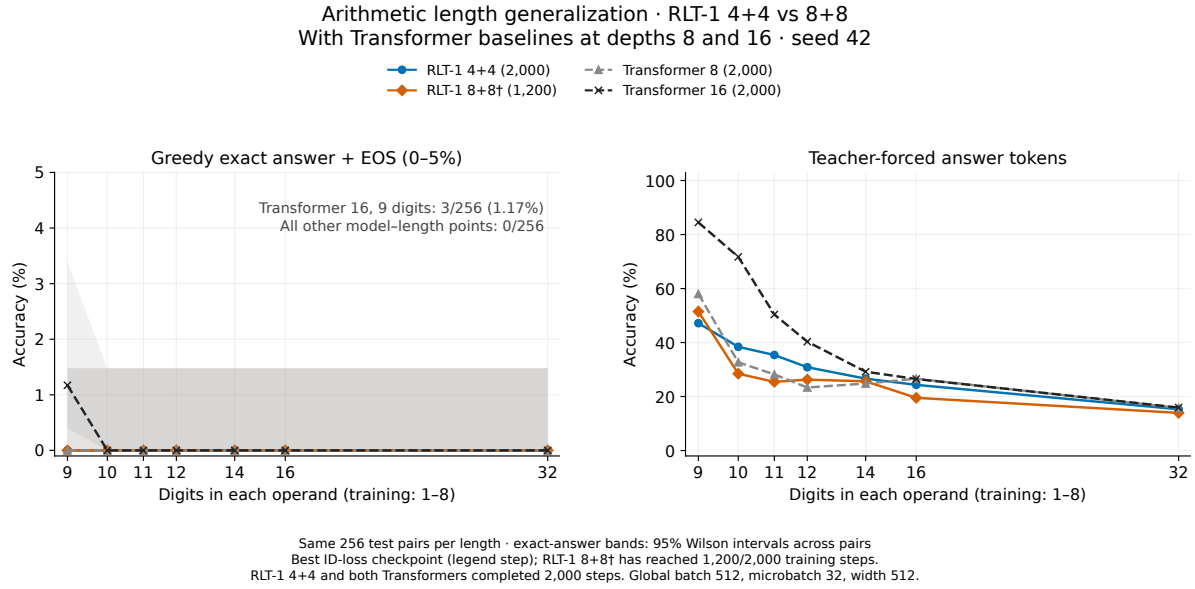

Figure 18 Addition generalization for RLT-1 $4+4$ and $8+8$, with Transformer baselines at both depths. All four seed-42 models use the same test pairs and scoring rules as Figure 17. RLT-1 $8+8$ uses its best checkpoint through step 1,200; the other three models completed 2,000 steps and selected that checkpoint. The training budgets therefore differ, and the Wilson bands describe uncertainty across test pairs rather than training initializations.