

HOW MODEL GROWTH, RECURSION, AND BOUNDARY OPERATORS INFLUENCE SCALING EXPONENTS

Zixi Chen^{1†} Akshay Vegesna² Samip Dahal² Andrew Gordon Wilson^{1,2}

¹New York University ²Q Labs

ABSTRACT

Scaling laws predict how loss decreases with increases in computation. We show, contrary to conventional wisdom, that architectural interventions can modify scaling exponents in pre-training, leading to exponential improvements in performance with increases in computation. As an anchoring point, we consider the architectural formulation of looped transformers. Although not typically used in this way, looping, also known as recursive depth, provides a mechanism for model growth, by increasing the number of loops during training. Model growth, with and without shared weights, provides the biggest changes to the scaling exponents. In particular, a 7.4B model growth architecture matches GPT-3 13B on CORE with roughly $20\times$ less compute, and has compute efficiency gains that increase with scale. Moreover, simply using a boundary operator in a vanilla transformer, which normalizes and injects an earlier block, also provides increasing compute-efficiency gains, although to a lesser extent. In the data-constrained, multi-epoch setting, standard looping has a useful regularizing effect, where we find it is compute-optimal to increase the number of loops with scale. These results can be understood through the lens of computational depth: for a given computational budget, we wish to increase the usable depth of the transformer, which can lead to efficiency gains that increase with scale.

1 INTRODUCTION

Scaling laws predict loss as a function of numbers of parameters and datapoints. They provide a recipe for configuring a balance of training data and model size to be on the *compute-optimal frontier*, providing the lowest loss for any computational budget (Kaplan et al., 2020; Hoffmann et al., 2022). Scaling laws follow a power law that depends on scaling constants and exponents. The scaling constants govern the vertical translation of loss curves as a function of compute, while the exponents affect the shapes of the curves themselves (Kaplan et al., 2020; Hoffmann et al., 2022). Modifying even the constants can have a significant effect on common practice. For example, Qiu et al. (2026) and Liu et al. (2025) recently showed that with the correct hyperparameter scaling, the Muon optimizer can provide a 40% compute efficiency gain over the optimizer AdamW, across scales. Muon is thus a promising candidate as the new default optimizer, de-throning Adam after nearly a decade of dominance.

In this paper, we ask what architectural interventions could possibly influence scaling *exponents*. It is the conventional wisdom that changes to the architecture generally only affect the scaling constants (Bansal et al., 2022; Hestness et al., 2017; Chen et al., 2026). But a change to the scaling exponent could be transformative, leading to power-law improvements in performance with increases in computation. And perhaps a change to the exponent is not as elusive as it might seem — even seemingly small hyperparameter details can influence whether an intervention affects the scaling law, as has been seen with Muon (Qiu et al., 2026).

Our starting intuition is the idea of *computational depth*: we may wish to achieve the greatest depth for any computational budget, in order to capture hierarchical structure in data, and compose many steps of computation. To this end, we consider *model growth*, whereby we grow the depth of the

†: Work done as an intern at Q Labs.

Code available at <https://github.com/qlabs-eng/scaling-exponents>

model during training. This approach is motivated by evidence that neural networks tend to learn simpler patterns early in training, with more complex functions or finer-scale components emerging as training progresses (Nakkiran et al., 2019; Rahaman et al., 2019). We hypothesize that, under a fixed computational budget, allocating greater depth to these later stages may therefore be beneficial. Although not typically used for this purpose, *looping* (Dehghani et al., 2019; Yang et al., 2024), also known as *recursive depth* (Geiping et al., 2025), provides a mechanism for model growth. A looped transformer applies the same core block of layers several times, corresponding to the number of loops, in a single forward pass, and every pass shares one set of weights. Typically the number of loops is fixed or random, and thus does not provide model growth. However, if we increase the number of loops during training, we effectively increase depth, without increasing the number of parameters. Alternatively, we can *untie* the weights, giving each pass its own copy of the core, so that growing the number of passes adds new blocks with distinct parameters. To explore these questions, we use the architectural formulation of looped transformers in Geiping et al. (2025), which compartmentalizes a transformer into prelude, core, and coda blocks, with a core block that is looped. We also consider standard looped transformers, which do not provide any model growth during training, and the architectural specification of the looped transformer without looping, which is simply a standard transformer but with a boundary operator between blocks and prelude injection. We illustrate each of these architectures in Figure 1.

With increased computation, we scale each of these architectures in a compute-optimal fashion, which means increasing the size of prelude, core, and coda blocks equally. We distinguish standard compute-optimal scaling, which scales the size of the model and data with increased computation, and model growth, which grows the size of the model during training itself. We consider performance in data unconstrained settings, and data constrained settings where we train for multiple epochs. We note that looping is mostly used for *inference-time* compute scaling in reasoning tasks, or for parameter efficiency, rather than as a way to train more efficiently for a fixed compute budget (Geiping et al., 2025; Saunshi et al., 2025; Yang et al., 2024).

We highlight some of our key results in Figure 2:

- Contrary to conventional wisdom, it is possible to change the scaling exponent in pre-training through architectural interventions. Earlier growth and looping studies do not focus on compute-optimal scaling with architecture-specific scaling hyperparameters, which may explain why the effect has gone unnoticed.
- The exponent differences are most obvious in looking at *compute multipliers*: the multiple of compute a standard transformer would require to reach the same value of the loss. For a scaling exponent improvement, the compute multipliers *increase* with scale, as they do for every variant we consider, both in FLOPs and in wall-clock time (Figure 15b).
- Growing the model during training, by increasing the number of core passes with *untied* weights, improves the scaling exponent. The grown model reaches the loss of a standard transformer with $1.55\times$ less compute at 10^{20} FLOPs, and this gap widens with scale.
- Looping provides a mechanism for parameter-efficient model growth. Growing the number of loops with tied weights obtains the exponent improvement from growth with the same number of parameters as a vanilla transformer, and trails untied growth by only a small constant factor, for a $1.36\times$ compute multiplier over a standard transformer at 10^{20} FLOPs.
- Notably, simply using the boundary operator in a vanilla transformer, which normalizes and injects the prelude block, also improves the scaling exponent, though less than model growth, with a $1.25\times$ compute multiplier over a standard transformer at 10^{20} FLOPs.
- In the multi-epoch setting, looping has a helpful regularizing effect. Training for 10 epochs on 100M tokens, the optimal number of loops increases with compute, and a larger number of loops decreases overfitting. Scaling the number of loops reaches the best loss of a weight-decay-tuned standard transformer with $2.2\times$ less compute.

Moreover, Figure 4 shows that our scaling laws hold under extrapolation. A 7.4B model growth architecture trained at $8\times$ the largest fitted compute lands on the predicted loss curve, and matches GPT-3 13B on CORE (Li et al., 2024; Karpathy, a) with roughly $20\times$ less compute (Brown et al., 2020). Because the compute multiplier grows with scale, the $1.8\times$ advantage over a standard transformer at this budget (1.2×10^{21} FLOPs), marked in the figure, is projected to reach $2.7\times$ at 10^{25} FLOPs.

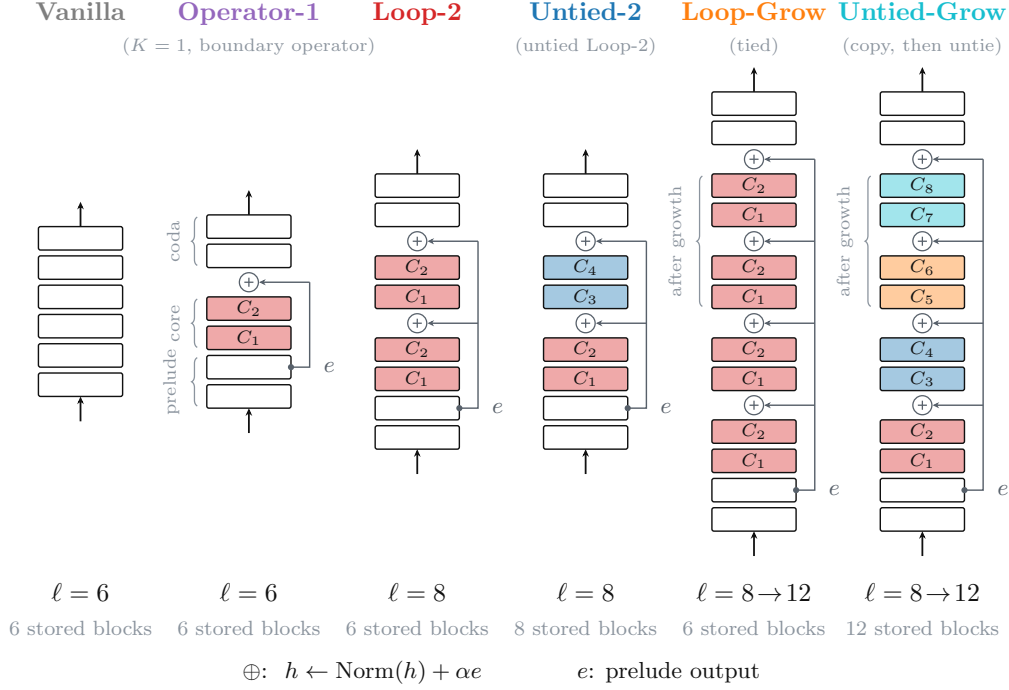


Figure 1: **Model growth, looping, and untied looping are members of one prelude–core–coda family.** In this family, a prelude of transformer blocks embeds the input into a representation e , a core of blocks is applied K times to a state h , and a coda of blocks produces the output. All variants share the same structure and differ on three axes, each isolated by one comparison. First, Vanilla and Operator-1 match in parameters and FLOPs and isolate the boundary operator $\oplus$, which normalizes the residual stream and adds back e . Second, Loop-2 and Untied-2 have the same FLOPs and isolate weight sharing, since Loop-2 stores one core and Untied-2 stores two. Third, Loop-Grow and Untied-Grow isolate growth, with braces marking the passes that turn on at the transition and ℓ giving the depth before and after.

We can gain insights into these results through the lens of *computational depth*. In particular, we define the computational depth as the number of layers meaningfully influencing the predictive distribution for a given computational budget. In Section 6, we interpret our results through the frame of computational depth.

The rest of the paper is organized as follows. Section 2 provides background on scaling laws, model growth, looped transformers, and the curse of depth. Section 3 introduces the prelude–core–coda family that unifies model growth, looping, and standard transformers. Section 4 considers single-epoch training, where we fit a compute-optimal recipe for each architecture (Section 4.1), show that model growth and the boundary operator improve the scaling exponent (Section 4.2), validate the fitted laws through extrapolation (Section 4.3), and give prescriptions for practitioners (Section 4.4). Section 5 then turns to multi-epoch training in the data-constrained regime, where we find that scaling the number of loops is preferable to scaling parameters. Section 6 interprets these results through the lens of computational depth. Finally, in Section 7 we discuss directions for future work.

2 BACKGROUND

Scaling laws and compute multipliers. A scaling law predicts how loss falls as training compute grows. Hoffmann et al. (2022) write loss as three terms: a power law in the parameter count N , a power law in the number of training tokens T , and an irreducible floor E , giving $L(N, T) = E + (N/N_0)^{-\alpha} + (T/T_0)^{-\beta}$ with N_0, T_0, α, β fitted. Each training token costs about $6N$ floating-point operations, so training compute is $C \approx 6NT$ (Kaplan et al., 2020), and a fixed compute budget

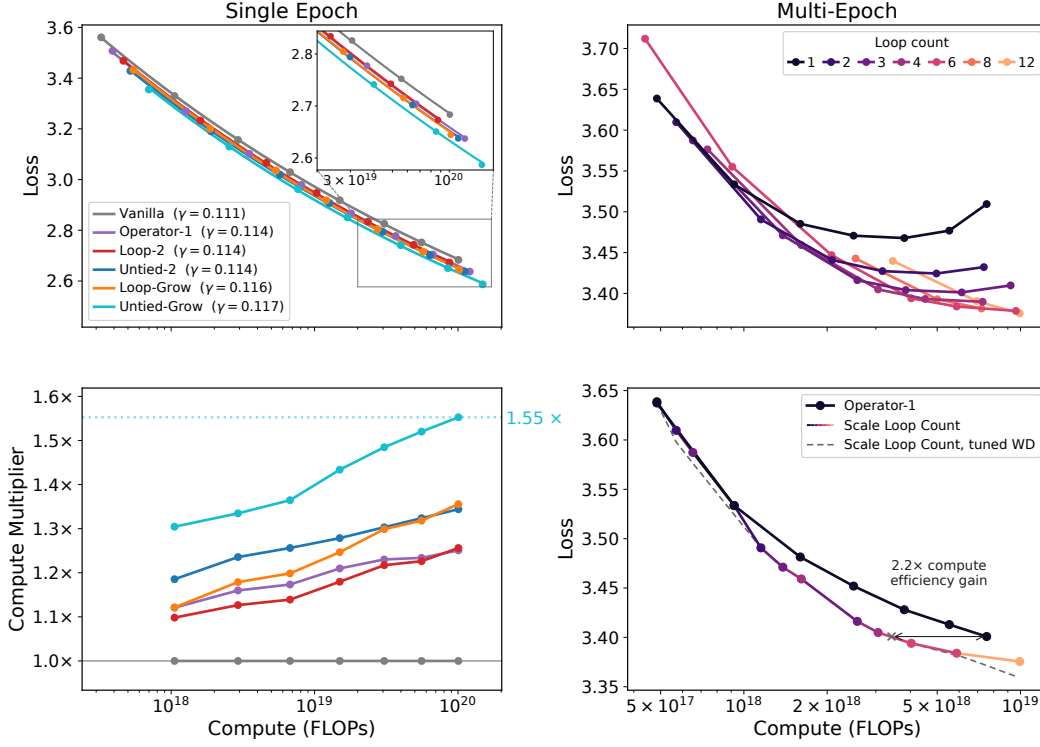


Figure 2: **In single-epoch compute-optimal training, model growth and the boundary operator improve the scaling exponent, and in multi-epoch training the optimal loop count grows with compute. Left: single epoch.** In the top panel, we fit scaling laws to validation loss on FineWeb, showing that the boundary operator and model growth can improve scaling exponents (Equation 1). In the bottom panel, we see these interventions lead to compute efficiency multipliers over vanilla transformers that increase with scale. The gain is largest for Untied-Grow, which reaches $1.55\times$ at 10^{20} FLOPs. **Right: multi epoch.** We fix a pool of 100M unique FineWeb tokens, and train for ten epochs. In this regime, the optimal number of loops increases with compute, and scaling the number of loops beats scaling parameters. At the marked point, Operator-1 needs $2.2\times$ the compute to match the looped loss. The top panel shows validation loss against compute at fixed numbers of loops from 1 to 12 and fixed weight decay. The bottom panel compares two ways of spending more compute: scaling the number of loops at fixed weight decay, or scaling the size of Operator-1 with tuned weight decay. Scaling the number of loops does not require much hyperparameter tuning, since the dashed line, which tunes weight decay at every loop count, stays close to the fixed-weight-decay curve.

leads to a trade-off between parameters and tokens. Choosing the best split at every budget gives the compute-optimal loss, which is again a power law,

$$L(C) = E + A \left(\frac{C}{C_0} \right)^{-\gamma}, \quad (1)$$

where A and γ are fitted and C_0 is a chosen base compute, which we set to the compute at which Vanilla is tuned. An architecture is therefore a family of models indexed by compute. The compute-optimal recipe depends on how N and T grow with compute (Hoffmann et al., 2022) and on how hyperparameters are scaled (Yang et al., 2022; Qiu et al., 2026).

To compare two architectures, we invert Equation 1 and ask how much compute each needs to reach the same loss. Let $\hat{C}_A(\ell)$ denote the compute at which architecture A ’s fitted law reaches loss ℓ . The compute multiplier of architecture B over A at loss ℓ is then $\pi(\ell) = \hat{C}_A(\ell)/\hat{C}_B(\ell)$. For example, $\pi(\ell) = 1.25$ means A needs $1.25\times$ the compute of B to reach loss ℓ . Throughout, A is the standard transformer and B is the variant. The multiplier can also be indexed by the standard transformer’s

compute, $\pi(C) = \pi(L_A(C))$, the multiplier at the loss the standard transformer reaches with budget C .

Techniques that help at small scale can fail to improve, or the gap can close, at larger scales (Rae et al., 2021), so improvements must be shown across scales (Kaplan et al., 2020; Liu et al., 2025; Krajewski et al., 2024; Potapczynski et al., 2024). Some interventions improve the exponent: transformers scale with a better parameter scaling exponent than LSTMs by exploiting long contexts (Kaplan et al., 2020), and Mixture-of-Experts scaling laws predict a growing compute-efficiency advantage over dense transformers as training budgets increase, with further gains from optimizing expert granularity (Krajewski et al., 2024). Others improve the constant, including structured matrices for MoE (Potapczynski et al., 2024) and the Muon optimizer over Adam (Liu et al., 2025). Constant improvements are more common than exponent improvements. On the theoretical side, Bordelon et al. (2025) show that feature learning improves the exponent on hard tasks.

Compute-optimal scaling laws depend on the training recipe. In particular, incorrect hyperparameter scaling yields a worse compute-optimal law (Yang et al., 2022; Qiu et al., 2026). To rule out hyperparameter scaling issues from the comparison, we fit a separate compute-optimal recipe for every architecture (Section 4).

Model growth. When training a ladder of models, each larger model is trained from scratch, so the compute spent on the smaller runs is wasted. Model growth addresses this waste by reusing the trained weights of a smaller model to initialize a larger training run (Chen et al., 2015; Du et al., 2024). Du et al. (2024) compare several ways to expand the parameter count and find that stacking transformer blocks, i.e., copying the trained blocks to increase depth partway through training, is the most efficient, reaching the same loss with a 50% speedup. However, Liew & Kato (2025) show that the more extensively a base model is pretrained, the less benefit further pretraining provides. This finding suggests that the benefits of checkpoint reuse may depend on the allocation of tokens between the two pretraining stages, motivating joint optimization of their token budgets. Related growth work, from staged training with function-preserving operators (Shen et al., 2022) to recycling converged mixture-of-experts checkpoints (Wang et al., 2026b), reports gains at one or a few target sizes rather than a change in the compute-optimal scaling law. In this work, we set aside the checkpoint-reuse motivation altogether and instead treat model growth as a way to increase the computational depth of the final model.

Looped transformers. A looped transformer applies the same block of layers several times in one forward pass (Dehghani et al., 2019; Yang et al., 2024). Looping therefore raises computational depth by reusing a block rather than adding a new one. Looping has two main motivations: an inductive bias toward iterative computation (Dehghani et al., 2019), and the ability to trade extra passes for accuracy at test time (Geiping et al., 2025). At matched computational depth, Saunshi et al. (2025) find that looping performs similarly to a dense transformer on reasoning tasks but much worse on memorization tasks. Nonetheless, recursive reasoning models, a close relative of looped transformers, reach results comparable to state-of-the-art models of the time on reasoning tasks with far fewer parameters and far less training compute (Wang et al., 2025; Jolicoeur-Martineau, 2025).

In the pretraining setting, HRM-Text (Wang et al., 2026a), a 1B-parameter recursive reasoning model trained from scratch on 40B synthetic and real tokens for about \$1,500, was recently shown to perform competitively with 2–7B-parameter open models on reasoning benchmarks. However, its setup differs substantially from standard pretraining, since it trains on instruction–response pairs with a task-completion objective rather than raw text, which makes the contribution of looping hard to isolate. Controlled comparisons in standard pretraining are more mixed. Prairie et al. (2026) show that looped transformers achieve lower loss than dense transformers at matched parameter and data budgets, and study compute-optimal allocation between looping and data at fixed model size. They do not, however, establish an advantage over dense transformers when model size and training data are jointly optimized for compute. Similarly, Schwethelm et al. (2026) show that, at matched computational depth, more recurrence leads to strictly worse performance when model size and data are jointly optimized. In this work, we show that with a proper block allocation and an optimal loop count, looping can improve performance under either control. Concurrently, Wang et al. (2026c) show that a sparse looped MoE has compute-efficiency gains that grow with scale, on proprietary data and architecture. In our setting, by contrast, we find that weight sharing alone is not enough to improve the exponent.

Depth scaling and the curse of depth. The effectiveness of depth scaling has been contested. On one side, Tay et al. (2021) argue that deep-narrow T5 models are Pareto-better on downstream tasks despite similar pretraining losses, and Liu et al. (2024) find that scaling depth beats scaling width at sub-billion scale. On the other, Kaplan et al. (2020) find that depth does not have a strong effect on the scaling law, and Levine et al. (2020) derive that the optimal depth should grow only logarithmically with width. One explanation for depth’s limited returns is known as the *curse of depth*. In pre-norm transformers, where normalization is applied to the input of each block rather than to the residual stream itself, the residual stream grows with depth, so each block’s update is a shrinking fraction of the stream and deeper blocks drift toward doing nothing (Liu et al., 2020; Sun et al., 2025). The curse can be measured with the logit lens, which decodes the residual stream after each block into a prediction and measures how far it is from the model’s final output distribution. The depth beyond which blocks stop changing the prediction is the effective depth (Nostalgebrais; Csordás et al., 2026). We refer to this quantity as the KL effective depth for clarity in this work.

Prior work mitigates the curse by scaling the pre-normalization output inside the residual branch (Sun et al., 2025), normalizing the residual stream (Wang et al., 2026a; Loshchilov et al., 2025), or injecting earlier representations into later ones (Wang et al., 2026a; Karpathy, b). Similar ingredients appear in the boundary operators of looped transformers. These depth-related interventions improve loss at a fixed model size, but none has been shown to improve the scaling exponent, leaving open whether mitigating the curse of depth can change the scaling law.

Data-constrained scaling and multi-epoch training. Compute is growing faster than the stock of high-quality text (Villalobos et al., 2024), so pretraining will increasingly repeat data over multiple epochs. Repeated tokens are worth less than fresh ones. Muennighoff et al. (2023) fit a scaling law in which repeated data counts for less than new data, and later work studies how loss behaves when the amount of unique data is fixed and compute keeps growing (Kim et al., 2026; Lovelace et al., 2026). The cost of repetition is that the models start overfitting, which can be exacerbated with model size. Weight decay is the standard remedy. Kim et al. (2026) show that larger models need more of it, so it must be retuned at every scale, and Lovelace et al. (2026) show that at a fixed weight decay the overfitting penalty follows a power law in model size. Looping adds depth without adding parameters, and Section 5 tests whether this lets a model add capacity in this regime without adding overfitting.

3 ARCHITECTURES

Every model we train is one three-stage network, following the prelude–core–coda formulation of Geiping et al. (2025): a prelude of transformer blocks embeds the input, a core of blocks is applied K times, and a coda of blocks produces the output (Equation 2). Model growth, looping, and deep transformers are all variants of this one network, so we can compare them on equal footing.

$$e = \mathcal{P}(s), \quad h_k = \mathcal{R}_{\theta_k}(\phi(h_{k-1}, e)), \quad k = 1, \dots, K, \quad y = \mathcal{C}(\rho(h_K, e)) \quad (2)$$

Here $\mathcal{P}$, $\mathcal{R}$, and $\mathcal{C}$ are the prelude, core, and coda, ϕ and ρ are boundary operators that mix the state with the prelude output e before each core pass and before the coda, θ_k are the weights of the k th pass, and K is the loop count. With P , C , and D blocks in the three stages, a token passes through $\ell = P + KC + D$ blocks, which we call the executed depth. We distinguish executed depth from the number of stored blocks, which we use for model size throughout. We fix the width-depth ratio at 128.

The architecture variants we study differ from one another on three axes: what happens at the boundary between passes, whether the passes share weights, and when the passes turn on. We isolate each axis with a matched comparison that holds everything else fixed (Figure 1), so a difference in scaling can be attributed to a single change.

Boundary operator (BO). A standard transformer is $K = 1$ with identity boundary operators. Between core passes and before the coda, we instead apply

$$\text{BO}(h, e) = \text{Norm}(h) + \alpha e, \quad (3)$$

which sets $\phi = \rho = \text{BO}$ in Equation 2. The operator has two parts, each with its own purpose. In a pre-norm block the residual stream grows with depth, so each update is a shrinking fraction of the stream, and normalizing lets every pass write at full weight. Adding back the prelude output e keeps every pass conditioned on the input. Normalizing between passes is standard in looped transformers (Geiping et al., 2025; Wang et al., 2026a) and a known remedy for the curse of depth. Likewise, re-injecting the input appears in recurrent models (Geiping et al., 2025; Prairie et al., 2026; Schwethelm et al., 2026) and in fixed-depth transformers such as nanochat and modded-nanogpt (Karpathy, b; Jordan et al., 2024). Prior looped transformers normalize before the coda but re-inject e only between core passes. We instead apply the same normalize-and-inject map before the coda as well, and find it to be important (Table 4, Figure 15a).

Looping. With the boundary operator fixed, the variants differ only in the core weights θ_k and in when the passes are active. Tying the weights, $\theta_1 = \dots = \theta_K$, gives a looped transformer: one core is stored and applied K times, so the model stores $P + C + D$ blocks and executes $P + KC + D$. Untying the weights gives each pass its own core. The untied model has the same computation graph and the same FLOPs as the tied one, but stores $P + KC + D$ blocks, so it is a deep transformer with the boundary operator. The untied model is therefore our control for separating the effect of depth from the effect of weight sharing.

Model growth. Using model growth, we start training at a small loop count K and raise it partway through. In the tied case the existing core is simply applied more times, adding no weights. In the untied case the trained core is copied and the copies are then trained separately, which is the depthwise stacking of Du et al. (2024). Growth changes only K , and only within a single run, whereas along a scaling ladder the prelude, core, and coda all grow together with model size. Finally, we fix the remaining choices that vary across prior looped transformers: the initial state is $h_0 = 0$, the loop count is fixed rather than sampled, and gradients flow through every pass.

Architecture variants. Figure 1 draws the six models we train at the smallest model size, arranged as three matched comparisons, one per axis. The first isolates the boundary operator: Vanilla is the standard pre-norm transformer, and Operator-1 matches it in parameters and FLOPs but adds the operator between the core passes and before the coda. The second isolates weight sharing: Loop-2 and Untied-2 both apply the core twice with the operator between passes and are matched in FLOPs and depth. The third isolates model growth: Loop-Grow and Untied-Grow each start as their fixed counterpart and double the core passes partway through training. Two operator-free controls, not shown in the figure, complete the family. Deep Vanilla is Untied-2 without the operator, and Deep Vanilla Grow is Untied-Grow without the operator, so the pair isolates growth without the operator. We give the full training procedure for Untied-Grow in Algorithm 1.

For each model along the compute-optimal scaling ladders, we use the notation $d\ell$ to denote a nominal depth of ℓ transformer blocks at our fixed width-depth ratio of 128, so $d8$ has a width of 1024. Vanilla, Operator-1, and the tied variants at $d\ell$ store exactly ℓ blocks, split across prelude, core, and coda, whereas the untied variants store an additional copy of the core for each extra pass, so Untied-2 at $d8$ stores 11 blocks (Table 2). With the family fixed, what remains is how to train each member compute-optimally, which is the subject of the next section.

4 IMPROVING THE COMPUTE-OPTIMAL SCALING EXPONENT

In this section we compare the compute-optimal scaling laws of the model families of Section 3 on fresh tokens, paying special attention to whether the gaps between them widen with scale or stay constant. Throughout, we train on FineWeb (Penedo et al., 2024) with the GPT-2 tokenizer (Radford et al., 2019). We describe the main setup here and defer details to Appendix A.

4.1 COMPUTE-OPTIMAL RECIPE

A gap in scaling exponents is only meaningful if every architecture is well tuned, since otherwise a difference in exponents could be a difference in tuning (Qiu et al., 2026). Indeed, transferring Vanilla’s recipe to Operator-1 costs 7.8×10^{-3} loss at a depth of 8 transformer blocks ($d8$) and erases its exponent improvement along the ladder (Figure 16b). We therefore fit a complete compute-

Algorithm 1 Untied Grow: normalize and re-inject with untied core growth

Require: Training batches $\{(x_t, y_t)\}_{t=1}^T$; injection scale α

Require: Growth step g ; initial core count K_0 ; final core count $K_f = mK_0, m \geq 2$

```
1: Initialize prelude  $\mathcal{P}$ , coda  $\mathcal{C}$ , and independent cores  $\{\mathcal{R}_{\theta_k}\}_{k=1}^{K_0}$ 
2:  $K \leftarrow K_0$ 
3: for  $t = 1, \dots, T$  do
4:   if  $t = g + 1$  then ▷ Grow after  $g$  training steps
5:     for  $r = 1, \dots, m - 1$  do
6:       for  $j = 1, \dots, K_0$  do
7:          $\theta_{rK_0+j} \leftarrow \text{copy}(\theta_j)$  ▷ Stack a copy of the core
8:       end for
9:     end for
10:     $K \leftarrow K_f$  ▷ Activate new untied cores
11:  end if
12:   $e \leftarrow \mathcal{P}(x_t), \quad h \leftarrow 0$ 
13:  for  $k = 1, \dots, K$  do
14:     $h \leftarrow \text{RMSNorm}(h) + \alpha e$  ▷ Boundary before each core
15:     $h \leftarrow \mathcal{R}_{\theta_k}(h)$  ▷ Distinct weights for every pass
16:  end for
17:   $h \leftarrow \text{RMSNorm}(h) + \alpha e$  ▷ Also before the coda
18:   $\hat{y}_t \leftarrow \mathcal{C}(h)$ 
19:   $\mathcal{L}_t \leftarrow \text{CrossEntropy}(\hat{y}_t, y_t)$ 
20:  Update all active parameters using  $\nabla \mathcal{L}_t$  ▷ Backpropagate through all  $K$  passes
21: end for
22: return Trained model
```

optimal recipe for every architecture, in the same four stages. In order, these are base hyperparameters at $d8$, tokens per stored parameter, growth timing (if applicable), and a learning-rate scaling rule. The ladders themselves run to about 10^{20} FLOPs, and we compare at equal compute throughout. Here we state only what the ladders depend on, and refer the reader to Appendix A.4 for the grids, sweeps, and fits behind each stage.

Before any of the four stages, three choices are made once for the whole family, using the tied variants at 1B tokens and matched parameter counts (Appendix A.2, Figure 7). The first is how to split the blocks across the prelude, core, and coda. The best fraction of blocks in the core is roughly constant across depth, so we scale the three stages evenly, giving leftover blocks first to the core and then to the coda. The second is the number of core passes. On fresh data the optimum lies between one and two at every budget, so the fixed variants use two core passes. The third is how many core passes to grow to. Starting from two passes, we find that four is best at every budget we tried, so the growth variants go from two to four core passes. When to grow, and how the token budget should shift for a model that will grow, are fitted along with the rest of the recipe, which we turn to next.

The four stages are then fitted separately for every architecture, since the optimal values differ across families. First, base hyperparameters are tuned at $d8$ on 1B tokens, one architecture at a time (Appendix A.4.1, Table 5). Second, we fit the optimal tokens per stored parameter at five compute budgets and find that it does not drift with scale for any family, as in Hoffmann et al. (2022). We therefore adopt one value per architecture, the rounded mean across budgets: 5 for Vanilla, 6 with the boundary operator, and 7–8 with growth (Table 7). Third, we tune a transition point for model growth, which lands at one-half to four-fifths of training (Appendix A.4.3). Fourth, we fit a scaling rule for the learning rate at $d8$ – $d10$, under which the learning rate decreases with model size. This single rule is enough, since re-sweeping and scaling the remaining hyperparameters moves the compute multiplier by at most 3% (Figure 16a). With the recipe fixed, the differences we report in Section 4.2 reflect the architecture rather than its tuning.

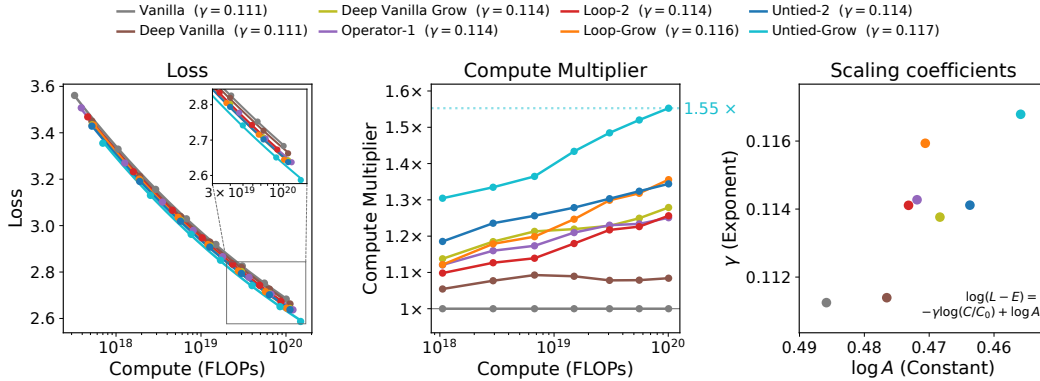


Figure 3: **Model growth and the boundary operator improve the scaling exponent, while untying improves only the constant.** **Left:** validation loss on FineWeb against training compute for eight compute-optimal ladders, one dot per trained model, with fitted power laws of Equation 1 sharing an irreducible loss fitted on Vanilla. The legend gives each arm’s fitted exponent. We find the regression standard error of the log–log slope to be lower than 10^{-3} in all cases. **Middle:** compute multiplier over Vanilla at the Vanilla budgets, interpolated in log compute without extrapolation. A flat curve is a constant improvement and a rising curve is an exponent improvement. Operator-1 matches Vanilla in parameters and FLOPs and rises from $1.12\times$ to $1.25\times$, so the operator alone improves the exponent. The grown families rise fastest, with Untied-Grow reaching $1.55\times$ at 10^{20} FLOPs. Untied-2 sits above Loop-2 by a factor that does not widen with scale, so untying moves only the constant. Deep Vanilla stays flat near $1.08\times$, a constant gain from a more favorable width–depth ratio rather than from the operator or growth. **Right:** fitted exponent γ against constant $\log A$ for each arm, where γ controls the slope and $\log A$ controls the vertical translation of scaling curves; better is up and to the right. The operator and growth each move arms up and right, so they improve exponent and constant independently, whereas untying and added depth move arms only rightward. Figure 15b shows the same ladders against wall-clock time.

4.2 IMPROVING THE EXPONENT OF THE SCALING LAWS

We run the compute-optimal recipes for each architecture independently. We fit the irreducible loss E in Equation 1 using a Huber loss for Vanilla (Hoffmann et al., 2022). For the remaining architectures, we fit an affine relationship in log–log space, $\log(L - E) = -\gamma \log(C/C_0) + \log A$, to obtain the exponent γ and the constant shift $\log A$. For compute multipliers, we estimate the compute required to reach Vanilla’s loss by linearly interpolating between the nearest two points in log–loss and log–compute space. On the x-axis, we plot the compute of the Vanilla run rather than the validation loss. We do not see a discrepancy between downstream metrics and validation loss, and in fact, Untied-Grow has slightly better downstream metrics when validation loss is controlled (Figure 19). We defer details to Appendix C.3.

The boundary operator improves the exponent. Operator-1 matches Vanilla in parameters and FLOPs and differs only by the boundary operator, and its multiplier over Vanilla increases with scale, from $1.12\times$ at 10^{18} FLOPs to $1.25\times$ at 10^{20} FLOPs. By simply adding a boundary operator, which has minimal effect on runtime, we improve the exponent of the scaling law. The same holds from Deep Vanilla to Untied-2, where the only difference is the boundary operator.

We hypothesize that the operator improves the exponent because the fraction of blocks it recovers grows with depth, and compute-optimal models get deeper with compute (Section 6).

Model growth improves the exponent. Untied-Grow starts as Untied-2 and differs only by doubling the core passes partway through training. Its multiplier over Vanilla widens from $1.30\times$ at 10^{18} FLOPs to $1.55\times$ at 10^{20} , whereas Untied-2’s widens from $1.19\times$ to $1.34\times$ over the same range. Model growth’s multiplier over Untied-2 therefore grows from $1.09\times$ to $1.16\times$, so growth improves the exponent rather than only the constant. We hypothesize that growth improves the exponent because the fraction of depth a network cannot yet use early in training grows with depth,

and compute-optimal models get deeper with compute (Section 6). Growth does not depend on the operator. From the right panel of Figure 3, Deep Vanilla Grow, which duplicates blocks mid-training without the boundary operator, shifts the constants and exponents over Deep Vanilla by about the same factor that Untied-Grow shifts over Untied-2, so the two improvements add rather than one enabling the other. With weight tying, Loop-Grow achieves a slightly smaller gain over Loop-2 in both the exponent and the constant.

Untying improves the constant. The untied variants sit a fixed factor above their tied counterparts. Untied-2 is $1.08\times$ above Loop-2 at 10^{18} FLOPs and $1.06\times$ at 10^{20} , and Untied-Grow is $1.16\times$ and $1.14\times$ above Loop-Grow at the same budgets. Weight sharing therefore costs a fixed factor of compute at every scale, rather than a factor that grows with the budget. Tying the weights and growing the model thus retains the exponent improvement of model growth at Vanilla’s parameter count, trailing untied growth only by a constant factor.

Deeper shapes improve the constant Width-to-depth ratios move the constant, contrary to what Kaplan et al. (2020) finds. A compute-optimal ladder fixes the split of compute between parameters and tokens, but not between width and depth, which we hold at a ratio of 128. Deep Vanilla, which executes the same blocks as Untied-2 with no operator, improves over Vanilla by a flat $1.08\times$, indicating a smaller optimal aspect ratio. However, decreasing width-to-depth ratios further doesn’t bring further improvements to the scaling constant (Figure 16c).

4.3 EXTRAPOLATION OF COMPUTE-OPTIMAL SCALING LAW

The exponents of Section 4.2 were fitted on ladders spanning 10^{18} to 10^{20} FLOPs. To test whether the exponent gap persists to larger scales through extrapolation, we train Untied-Grow at $8\times$ the largest fitted compute and ask whether the fitted law predicts the loss of that run. The law does: the run lands on the extrapolated curve, and the gain carries over to downstream performance.

We rerun the Untied-Grow and Vanilla ladders on FineWeb-Edu using the same recipe fitted on FineWeb (Appendix C). The gains in scaling exponents from Vanilla to Untied-Grow are similar across the two datasets, while switching from FineWeb to FineWeb-Edu yields a constant multiplicative improvement in downstream performance (Figure 18). Building on the downstream scaling law in Grattafiori et al. (2024), we develop a scaling law to predict downstream performance (Appendix C.4) and set our training target to match the downstream capability of GPT-3 13B as estimated by Karpathy (a). Our scaling law predicts that Untied-Grow d_{26} , a 7.4B model that starts at 5B parameters, can reach this target using the compute-optimal recipe (Table 9).

Figure 4 (left) shows that the 7.4B run lands on, and in fact slightly below, the extrapolated curve, so the exponent improvement of Section 4.2 extend to larger scales. The same holds for the downstream scaling law: the run’s answer NLL and CORE score land on the curves predicted from the small-scale fits (Figure 4, middle and right).

The 7.4B Untied-Grow model reaches 0.3865 CORE, on par with the GPT-3 13B reference of 0.3852 (Karpathy, a), at 1.23×10^{21} FLOPs against 2.31×10^{22} for GPT-3 13B (Brown et al., 2020). The two models were trained on different data, and the GPT-3 reference is an estimate from a separate evaluation pipeline, so the roughly $20\times$ gap is indicative rather than a controlled comparison.

The compute-efficiency advantage of Untied-Grow over Vanilla widens with scale, as an exponent improvement predicts. Using the fitted FineWeb-Edu laws with a shared irreducible loss, the multiplier is $1.6\times$ at 10^{20} FLOPs, the end of the measured ladders, $1.8\times$ at 1.23×10^{21} FLOPs, the compute of the 7.4B run (Figure 4), and $2.7\times$ at 10^{25} FLOPs, the budget of a modern pretraining run.

4.4 PRESCRIPTIONS FOR PRACTITIONERS

For a standard transformer, a practitioner chooses model size and token count at a given budget. The family we study adds three choices: what fraction of the blocks goes in the core, how many times the core runs, and how much to grow and when. Fortunately, these choices can be kept fixed as we scale model size and tokens proportionally in the compute-optimal setting (Appendix A.2). This makes

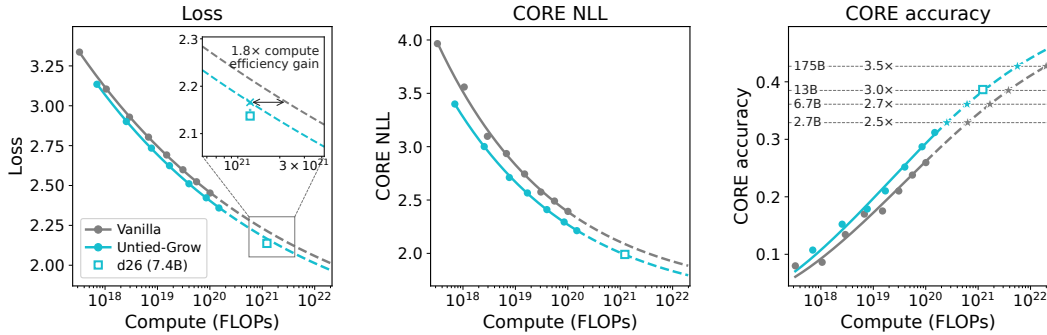


Figure 4: **The scaling law fitted on 10^{18} – 10^{20} FLOPs predicts an Untied-Grow run at $8\times$ the fitted compute, and the compute-efficiency advantage over Vanilla widens with scale.** Ladders are trained on FineWeb-Edu, solid lines are fits, dashed lines are their extrapolation, and the square is a 7.4B Untied-Grow run not used in any fit. **Left:** validation loss on FineWeb-Edu, with a shared irreducible loss fitted on Vanilla. The run lands 0.03 below the forecast. The inset arrow marks the compute multiplier at the run’s budget: Untied-Grow reaches Vanilla’s loss with $1.8\times$ less compute, and the fitted laws project $2.7\times$ at 10^{25} FLOPs. **Middle:** answer NLL, where the run lands within 0.002 of the forecast. **Right:** CORE accuracy predicted from compute (Appendix C.2), where the run reaches 0.387 against a forecast of 0.384. Horizontal lines are GPT-3 CORE scores from 2.7B to 175B, and stars mark where each curve crosses them. The Vanilla-to-Untied-Grow compute ratio at those crossings grows from $2.5\times$ to $3.5\times$, so the exponent improvement of Figure 3 carries over downstream.

the recipe straightforward to use: calibrate the architecture and growth schedule at small scale, then reuse them at larger budgets. In particular, retuning the token allocation or growth fraction at each size brings little benefit in our sensitivity tests (Appendix A.2.4). The full tuning procedure is in Appendix A.4, and Table 9 lists the recipes used in our experiments.

We recommend careful, architecture-specific tuning and scaling of hyperparameters, as these are crucial to realizing the full scaling improvements (Appendices A.4 and B.2). In our setup, the learning-rate scaling rule in Equation 13 works well while the other hyperparameters remain at their base-tuned values. Reusing a standard transformer’s hyperparameters without tuning them for the new architecture can hide an exponent improvement (Figure 16b).

For single-epoch, compute-optimal pretraining, we recommend untied weights when memory is not a constraint: the untied variants reach the same loss with less compute than their tied counterparts (Section 4.2). Weight tying remains useful when parameter storage is the priority, retaining the exponent improvement from growth while trading a constant factor of compute efficiency for fewer parameters. In the data-constrained, multi-epoch setting, tying also provides a regularization benefit as we will see next.

5 LOOPED TRANSFORMERS IN THE DATA-CONSTRAINED REGIME

Section 4 showed that on fresh data, weight sharing is not compute-optimal, nor is increasing the loop count across scales. However, when data are repeated, we observe that the optimal loop count grows with compute, and tied weights become the better way to add capacity. This regime matters because compute is growing far faster than high-quality text (Villalobos et al., 2024), so pretraining will increasingly repeat data over multiple epochs (Muennighoff et al., 2023; Kim et al., 2026; Lovelace et al., 2026; Vegesna et al., 2026). Repeated data leads to overfitting, and the standard mitigation is weight decay retuned at every model size (Kim et al., 2026). Looping offers a different mitigation: adding depth without adding parameters to overfit.

We fix a pool of 100M unique FineWeb tokens and train every model for ten epochs, a budget of 1B tokens. We use the Operator-1 family from Section 4, with all hyperparameters other than weight decay fixed at the tuned values. There are then two ways to spend more compute: add parameters

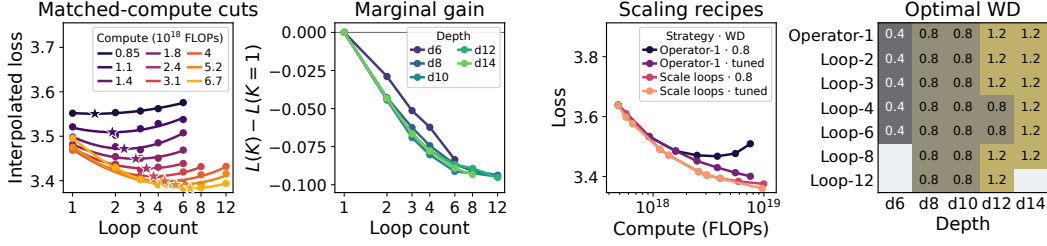


Figure 5: **In multi-epoch training, the optimal loop count grows with compute, scaling the loop count beats scaling model size even against tuned weight decay, and looping leaves the optimal weight decay nearly unchanged.** Models are trained on 100M unique FineWeb tokens for 10 epochs, and all losses shown are validation losses on FineWeb. **First: matched-compute cuts.** The loss of each loop count is interpolated in compute between neighbouring depths, curves are quadratics in log loss against log loop count, and stars mark their minima. The optimal loop count rises from 1.4 to 6.7 across budgets, the gap to Operator-1 grows from 0.00 to 0.11. **Second: marginal gain from looping.** For each depth, we plot the loss at loop count K minus the Operator-1 loss at the same depth. Every curve decreases monotonically, and from $d8$ upward the curves lie within 0.006 of one another at every loop count, so the gain from looping appears nearly independent of depth. **Third: weight decay when scaling loops versus model size.** Scaling model size at fixed weight decay overfits, and retuning weight decay at every model size only reaches 3.40 loss at 7.5×10^{18} FLOPs. Scaling the loop count at fixed model size and fixed weight decay matches that loss with $2.2\times$ less compute, and tuning weight decay on top gains at most 0.02, so loop-count scaling is more compute-efficient and nearly free of tuning. **Fourth: optimal weight decay by depth and loop count.** The optimum rises with depth but is nearly flat in loop count, so overfitting tracks stored parameters rather than executed depth.

at a fixed loop count, or add loops at a fixed parameter count. We train a grid over both, Operator-1 through Loop-12 at 120M to 1.4B parameters, and sweep weight decay over the same grid.

The optimal loop count increases with compute (Figure 5, first panel). We observe that every fixed-loop-count curve eventually overfits and turns upward, but the turn comes later and the minimum is lower for higher loop counts (Figure 2, upper right). At matched compute, the optimal loop count therefore rises from about one at the smallest budget to about seven at the largest, and the loss gap to Operator-1 grows from 0.00 to 0.11. Prior work shows that the optimal weight decay increases with parameter count on repeated data, because larger models overfit more and need more regularization (Kim et al., 2026). Looping appears to act as a similar regularizer: it delays overfitting without adding parameters, and its optimal strength similarly grows with compute.

In addition, we find that scaling the loop count is more compute-efficient than scaling model size, even when weight decay is tuned for model-size scaling (Figure 5, third panel). Here, scaling model size increases width and depth together at our fixed width–depth ratio. Retuning weight decay at every model size mitigates overfitting, but scaling the loop count at fixed model size and fixed weight decay matches the best tuned model-size-scaling loss with $2.2\times$ less compute. Moreover, weight-decay tuning on top of loop-count scaling reduces loss by at most 0.02, because the optimal weight decay of the Operator-1 model changes. This effect is visible in Figure 5 (fourth panel): the optimal weight decay rises with depth but is nearly flat in loop count, so the value tuned at one loop transfers to larger loop counts. Finally, untied looping adds the same depth with more parameters and does not beat Operator-1 (Figure 24), so weight sharing is an effective regularization technique in multi-epoch training.

We hypothesize that a model overfits with the parameters it stores, not with the blocks it executes. At small budgets the model is compute-limited, so a new parameter is preferred to a reused one, as on fresh data. Once the stored parameters begin to overfit the corpus, loops become the better way to add capacity, so the optimal loop count grows with compute. The last two panels of Figure 5 support the hypothesis directly: the marginal gains of looping are independent of the stored parameters and the optimal weight decay tracks stored parameters, not executed depth. We return to this picture in Section 6 through the lens of computational depth.

6 COMPUTATIONAL DEPTH

Our motivation is *computational depth*: for a fixed compute budget, we want a model to have as much usable depth as possible, since depth is what allows a network to compose many steps of computation. In particular, we define computational depth as the number of blocks that meaningfully influence the predictive distribution. Executed depth and computational depth need not coincide, however, and we see two ways in which compute spent on depth could go to waste. First, a block can execute without changing the prediction, which is the curse of depth (Sun et al., 2025). Second, a block can consume compute budget throughout training even though the network may only need the block near the end. A waste of either kind would only change the exponent if it grew with scale. We hypothesize that both wastes take up a larger fraction of depth at larger depths, and the compute-optimal depth rises with budget, so both should grow with scale. Each of our interventions plausibly removes one of these wastes, which would explain why both change the exponent in Section 4.2: the boundary operator keeps every executed block contributing to the prediction, and model growth keeps the model shallow until the added depth is needed.

The boundary operator increases computational depth. We hypothesize that the boundary operator improves the exponent because it closes the gap between executed and computational depth. In a pre-norm transformer the residual stream grows with depth, so a growing share of the blocks a token passes through does little to change it. Normalizing the stream and re-injecting the input lets every block write at full relative weight. To measure how much depth a model uses, we compute its *KL effective depth* (Nostalgebrais; Csordás et al., 2026): we decode the residual stream after each block with the logit lens and record the first block after the KL peak at which the decoded prediction is within a KL threshold of the model’s final output. KL effective depth is only a proxy for computational depth, since falling within this threshold does not mean later blocks stop changing the prediction, and unused blocks in the middle of the network go undetected. Untied-2 and Deep Vanilla execute the same blocks, but Untied-2 reaches a KL effective depth of 24 against 20 at 10^{20} FLOPs (Figure 6, left), so the operator appears to improve computational depth. Operator-1 sits only slightly above Vanilla, so the operator changes little at the shallowest depths, and the gap widens as models get deeper, as the hypothesis predicts. Another evidence comes from width-only scaling. Scaling only the width, not the depth, turns the exponent improvement from Operator-1 to Vanilla to a constant one (Figure 16e). This suggests depth scaling is necessary for the exponent improvement with the boundary operator.

Model growth adds depth when it is needed. We hypothesize that model growth improves the exponent because a fixed-depth model pays for depth it does not yet need. Networks fit simple structure early in training and more sophisticated structure only toward the end (Nakkiran et al., 2019; Rahaman et al., 2019), so a fixed-depth model may not need all of its blocks early in training. We further hypothesize that the fraction of its depth a model cannot yet use grows with depth. Since compute-optimal models get deeper with scale, the compute a fixed-depth model spends on such depth is then a growing fraction of the budget. Three observations support this hypothesis. Growth timing matters: the transition sweeps of Appendix A.2.3 have interior minima at fixed model size and budget, so paying for depth too early or too late both cost loss. Grown models prefer a smaller starting model trained on more tokens, with the optimal tokens per parameter rising from 6 to 7–8 (Table 7), which is what the hypothesis predicts: spend most of the budget shallow and add depth late. Finally, Untied-Grow reaches a KL effective depth of 36 against 24 for Untied-2 at 10^{20} FLOPs (Figure 6, left), so growth appears to raise computational depth.

Looping adds depth without adding parameters to overfit. Under repeated data, we observe that adding depth through looping is better than adding parameters directly (Section 5). We hypothesize that this is because a model overfits with the parameters it stores, not the blocks it executes, so tied looping adds depth without adding anything to overfit. The control supports this hypothesis: untied looping adds the same depth with more parameters and does not beat tuned Operator-1 (Appendix D), so weight sharing, not depth, is what helps under repetition. KL effective depth tells the same story: scaling the loop count raises KL effective depth faster than scaling depth at $K = 1$ with tuned weight decay, reaching 18 against 14 layers at the largest budget (Figure 6, right).

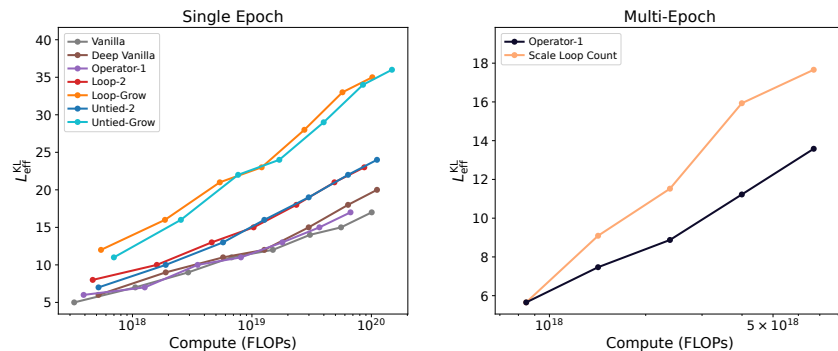


Figure 6: **Growth and looping raise KL effective depth along the compute-optimal single-epoch ladders, and in multi-epoch training, scaling the loop count raises KL effective depth faster than scaling depth.** KL effective depth is measured with the logit lens: we decode the residual stream after each block and record the first block after the KL peak at which the decoded prediction is within 2 nats of the model’s final output. **(Left)** Single-epoch scaling ladders. KL effective depth rises with compute for every family. Moreover, the curves group by depth multiplier: the one-pass models sit lowest, the two-pass models above them, and the models with model growth highest, reaching roughly twice Vanilla’s KL effective depth at 10^{20} FLOPs. Within each group, the tied and untied curves coincide, so weight sharing does not change KL effective depth. Deep Vanilla, however, executes as many blocks as Untied-2 yet has a smaller KL effective depth (20 versus 24 at 10^{20}), whereas Operator-1 lies on the Vanilla curve, so the boundary operator matters more at depth. **(Right)** Multi-epoch training on 100M unique tokens for 10 epochs, where each point is the best configuration at that budget (interpolated between neighboring checkpoints). Scaling the loop count at fixed weight decay gives a larger KL effective depth than scaling depth at $K = 1$ with tuned weight decay, and the gap widens with compute.

7 DISCUSSION

For many years, *data* interventions have been the main driver of advances in pretraining efficiency. By contrast, architectural innovations at pretraining have largely been absent, with the common belief that they can at best influence only scaling constants. However, we may be entering a new era, where the methodological landscape of research undergoes great change. We are now starting to see Muon challenge Adam as the default optimizer, after nearly a decade in which Adam dominated and hundreds of proposed alternatives never achieved broad adoption. Similarly, recursive depth, or looping, has recently been gaining mainstream traction for parameter-efficient representations, inference-time scaling, and reasoning. Moreover, as we become more data constrained, it will become increasingly natural to look to methodological interventions for further performance gains.

Contrary to the conventional wisdom, we have shown that architectural interventions at pretraining can influence scaling exponents. Both model growth, and even a simple boundary operator, provide compute efficiency gains that *increase* with scale. Selecting for computational depth, reaching the largest usable depth for a given computational budget, can explain the effect of these interventions on scaling laws. Furthermore, in the data-constrained setting, where we train for multiple epochs, increasing the loop count with scale becomes compute-optimal. This finding may be particularly salient as data becomes a more constrained resource in the future.

Going forward, proposing architectural interventions that increase computational depth could lead to further exponent improvements. Context length, the number of experts in a mixture-of-experts model, and width all grow with compute, yet all are fixed before training begins, so growing them on the schedule the network needs may change the exponent. Within depth itself, staged schedules (two passes, then four, then six) may increase the exponent further still. More broadly, the distinction between constant and exponent improvements deserves to be a standard part of how new architectures and training recipes are evaluated: a better constant saves the same factor of compute at every scale, while a better exponent saves a factor that compounds as budgets grow.

Acknowledgements. We thank Jonas Geiping, Neel Gupta, and Shikai Qiu for helpful discussions.

REFERENCES

- Yamini Bansal, Behrooz Ghorbani, Ankush Garg, Biao Zhang, Colin Cherry, Behnam Neyshabur, and Orhan Firat. Data scaling laws in nmt: The effect of noise and architecture. In *International Conference on Machine Learning*, pp. 1466–1482. PMLR, 2022.
- Blake Bordelon, Alexander Atanasov, and Cengiz Pehlevan. How feature learning can improve neural scaling laws. In *International Conference on Learning Representations*, volume 2025, pp. 51909–51939, 2025.
- Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.
- Guangyu Chen, Yu Zhang, Jianlin Su, Weixin Xu, Siyuan Pan, Yaoyu Wang, Yucheng Wang, Guanduo Chen, Bohong Yin, et al. Attention residuals. *arXiv preprint arXiv:2603.15031*, 2026.
- Tianqi Chen, Ian Goodfellow, and Jonathon Shlens. Net2net: Accelerating learning via knowledge transfer. *arXiv preprint arXiv:1511.05641*, 2015.
- Róbert Csordás, Christopher D Manning, and Chris Potts. Do language models use their depth efficiently? *Advances in Neural Information Processing Systems*, 38:160313–160362, 2026.
- Mostafa Dehghani, Stephan Gouws, Oriol Vinyals, Jakob Uszkoreit, and Łukasz Kaiser. Universal transformers. In *International Conference on Learning Representations*, 2019.
- Wenyu Du, Tongxu Luo, Zihan Qiu, Zeyu Huang, Yikang Shen, Reynold Cheng, Yike Guo, and Jie Fu. Stacking your transformers: A closer look at model growth for efficient llm pre-training. *Advances in Neural Information Processing Systems*, 37:10491–10540, 2024.
- Jonas Geiping, Sean McLeish, Neel Jain, John Kirchenbauer, Siddharth Singh, Brian Bartoldson, Bhavya Kailkhura, Abhinav Bhatele, and Tom Goldstein. Scaling up test-time compute with latent reasoning: A recurrent depth approach. *Advances in Neural Information Processing Systems*, 38: 41340–41391, 2025.
- Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.
- Joel Hestness, Sharan Narang, Newsha Ardalani, Gregory Diamos, Heewoo Jun, Hassan Kianinejad, Md Mostofa Ali Patwary, Yang Yang, and Yanqi Zhou. Deep learning scaling is predictable, empirically. *arXiv preprint arXiv:1712.00409*, 2017.
- Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Henigan, Eric Noland, Katie Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Jack W. Rae, Oriol Vinyals, and Laurent Sifre. Training compute-optimal large language models. In *Advances in Neural Information Processing Systems*, 2022.
- Alexia Jolicoeur-Martineau. Less is more: Recursive reasoning with tiny networks. *arXiv preprint arXiv:2510.04871*, 2025.
- Keller Jordan et al. modded-nanogpt: NanoGPT speedrun. <https://github.com/KellerJordan/modded-nanogpt>, 2024.
- Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.
- Andrej Karpathy. [jan 7 2026] nanochat miniseries v1 · karpathy nanochat, a. URL <https://github.com/karpathy/nanochat/discussions/420>.

- Andrej Karpathy. Karpathy/nanochat: The best ChatGPT that \$100 can buy., b. URL <https://github.com/karpathy/nanochat>.
- Konwoo Kim, Suhas Kotha, Percy Liang, and Tatsunori Hashimoto. Pre-training under infinite compute. In *International Conference on Learning Representations*, volume 2026, pp. 74596–74636, 2026.
- Jakub Krajewski, Jan Ludziejewski, Kamil Adamczewski, Maciej Pióro, Michał Krutul, Szymon Antoniak, Kamil Ciebiera, Krystian Król, Tomasz Odrzygóźdź, Piotr Sankowski, et al. Scaling laws for fine-grained mixture of experts. *arXiv preprint arXiv:2402.07871*, 2024.
- Yoav Levine, Noam Wies, Or Sharir, Hofit Bata, and Amnon Shashua. Limits to depth efficiencies of self-attention. *Advances in Neural Information Processing Systems*, 33:22640–22651, 2020.
- Jeffrey Li, Alex Fang, Georgios Smyrnis, Maor Ivgi, Matt Jordan, Samir Gadre, Hritik Bansal, Etash Guha, Sedrick Keh, Kushal Arora, et al. Datacomp-lm: In search of the next generation of training sets for language models. *Advances in Neural Information Processing Systems*, 37:14200–14282, 2024.
- Seng Pei Liew and Takuya Kato. Reusing overtrained language models saturates scaling. *arXiv preprint arXiv:2510.06548*, 2025.
- Jingyuan Liu, Jianlin Su, Xingcheng Yao, Zhejun Jiang, Guokun Lai, Yulun Du, Yidao Qin, Weixin Xu, Enzhe Lu, Junjie Yan, et al. Muon is scalable for llm training. *arXiv preprint arXiv:2502.16982*, 2025.
- Liyuan Liu, Xiaodong Liu, Jianfeng Gao, Weizhu Chen, and Jiawei Han. Understanding the difficulty of training transformers. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 5747–5763, 2020.
- Zechun Liu, Changsheng Zhao, Forrest Iandola, Chen Lai, Yuandong Tian, Igor Fedorov, Yunyang Xiong, Ernie Chang, Yangyang Shi, Raghuraman Krishnamoorthi, et al. Mobilellm: Optimizing sub-billion parameter language models for on-device use cases. *arXiv preprint arXiv:2402.14905*, 2024.
- Ilya Loshchilov, Cheng-Ping Hsieh, Simeng Sun, and Boris Ginsburg. ngpt: Normalized transformer with representation learning on the hypersphere. In *International Conference on Learning Representations*, volume 2025, pp. 74014–74038, 2025.
- Justin Lovelace, Christian Belardi, Srivatsa Kundurthy, Shriya Sudhakar, and Kilian Q Weinberger. Prescriptive scaling laws for data constrained training. *arXiv preprint arXiv:2605.01640*, 2026.
- Bruno Mlodozieniec, Pierre Ablin, Louis Béthune, Dan Busbridge, Michal Klein, Jason Ramapuram, et al. Completed hyperparameter transfer across modules, width, depth, batch and duration. In *International Conference on Learning Representations*, volume 2026, pp. 45300–45325, 2026.
- Niklas Muennighoff, Alexander M Rush, Boaz Barak, Teven Le Scao, Aleksandra Piktus, Nouamane Tazi, Sampo Pyysalo, Thomas Wolf, and Colin Raffel. Scaling data-constrained language models. In *Advances in Neural Information Processing Systems*, 2023.
- Preetum Nakkiran, Gal Kaplun, Dimitris Kalimeris, Tristan Yang, Benjamin L. Edelman, Fred Zhang, and Boaz Barak. Sgd on neural networks learns functions of increasing complexity. *Advances in Neural Information Processing Systems*, 32, 2019.
- Nostalgebrais. Interpreting gpt: The logit lens. URL <https://www.lesswrong.com/posts/AcKRB8wDpdAN6v6ru/interpreting-gpt-the-logit-lens>.
- Guilherme Penedo, Hynek Kydlíček, Anton Lozhkov, Margaret Mitchell, Colin Raffel, Leandro Von Werra, Thomas Wolf, et al. The fineweb datasets: Decanting the web for the finest text data at scale. *Advances in Neural Information Processing Systems*, 37:30811–30849, 2024.
- Andres Potapczynski, Shikai Qiu, Marc Finzi, Christopher Ferri, Zixi Chen, Micah Goldblum, C Bayan Bruss, Christopher De, and Andrew Wilson. Searching for efficient linear layers over a continuous space of structured matrices. *Advances in Neural Information Processing Systems*, 37:3857–3881, 2024.

- Hayden Prairie, Zachary Novack, Taylor Berg-Kirkpatrick, and Daniel Y Fu. Parcae: Scaling laws for stable looped language models. *arXiv preprint arXiv:2604.12946*, 2026.
- Shikai Qiu, Zixi Chen, Hoang Phan, Qi Lei, and Andrew Wilson. Hyperparameter transfer enables consistent gains of matrix-preconditioned optimizers across scales. *Advances in Neural Information Processing Systems*, 38:130867–130911, 2026.
- Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.
- Jack W Rae, Sebastian Borgeaud, Trevor Cai, Katie Millican, Jordan Hoffmann, Francis Song, John Aslanides, Sarah Henderson, Roman Ring, Susannah Young, et al. Scaling language models: Methods, analysis & insights from training gopher. *arXiv preprint arXiv:2112.11446*, 2021.
- Nasim Rahaman, Aristide Baratin, Devansh Arpit, Felix Draxler, Min Lin, Fred A. Hamprecht, Yoshua Bengio, and Aaron Courville. On the spectral bias of neural networks. In *Proceedings of the 36th International Conference on Machine Learning*, pp. 5301–5310, 2019.
- 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*, volume 2025, pp. 14855–14881, 2025.
- Kristian Schwethelm, Daniel Rueckert, and Georgios Kaissis. How much is one recurrence worth? iso-depth scaling laws for looped language models. *arXiv preprint arXiv:2604.21106*, 2026.
- Sheng Shen, Pete Walsh, Kurt Keutzer, Jesse Dodge, Matthew Peters, and Iz Beltagy. Staged training for transformer language models. In *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pp. 19893–19908. PMLR, 2022. URL <https://proceedings.mlr.press/v162/shen22f.html>.
- Wenfeng Sun, Xinyuan Song, Pengxiang Li, Lu Yin, Yefeng Zheng, and Shiwei Liu. The curse of depth in large language models. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025.
- Yi Tay, Mostafa Dehghani, Jinfeng Rao, William Fedus, Samira Abnar, Hyung Won Chung, Sharan Narang, Dani Yogatama, Ashish Vaswani, and Donald Metzler. Scale efficiently: Insights from pre-training and fine-tuning transformers. *arXiv preprint arXiv:2109.10686*, 2021.
- Akshay Vegesna, Samip Dahal, Chinmay Karkar, Bishwas Mandal, Shmuel Berman, and Zhiwei Xu. Slowrun: Language modeling with infinite compute, fixed data. <https://github.com/qlabs-eng/slowrun>, 2026.
- Pablo Villalobos, Anson Ho, Jaime Sevilla, Tamay Besiroglu, Lennart Heim, and Marius Hobbhahn. Will we run out of data? limits of llm scaling based on human-generated data, 2024. URL <https://arxiv.org/abs/2211.04325>.
- Guan Wang, Jin Li, Yuhao Sun, Xing Chen, Changling Liu, Yue Wu, Meng Lu, Sen Song, and Yasin Abbasi Yadkori. Hierarchical reasoning model, 2025. URL <https://arxiv.org/abs/2506.21734>.
- Guan Wang, Changling Liu, Chenyu Wang, Cai Zhou, Yuhao Sun, Yifei Wu, Shuai Zhen, Luca Scimeca, and Yasin Abbasi Yadkori. Hrm-text: Efficient pretraining beyond scaling. *arXiv preprint arXiv:2605.20613*, 2026a.
- Ruizhe Wang, Yucheng Ding, Xiao Liu, Yaoxiang Wang, Peng Cheng, Baining Guo, Zhengjun Zha, and Yeyun Gong. Beyond sunk costs: Boosting llm pre-training efficiency via orthogonal growth of mixture-of-experts. In *International Conference on Machine Learning*, 2026b. URL <https://arxiv.org/abs/2510.08008>.
- Shaowen Wang, Ge Zhang, Kairong Luo, Yuhao Wu, Shaofan Liu, Jiaheng Liu, Wenhao Huang, Shen Yan, and Jian Li. Smelt: Scaling laws for compute-matched moe looped transformers. *arXiv preprint arXiv:2609.01343*, 2026c.

- Kaiyue Wen, David Hall, Tengyu Ma, and Percy Liang. Fantastic pretraining optimizers and where to find them. In *International Conference on Learning Representations*, volume 2026, pp. 144731–144838, 2026.
- Greg Yang, Edward J. Hu, Igor Babuschkin, Szymon Sidor, Xiaodong Liu, David Farhi, Nick Ryder, Jakub Pachocki, Weizhu Chen, and Jianfeng Gao. Tensor programs v: Tuning large neural networks via zero-shot hyperparameter transfer, 2022. URL <https://arxiv.org/abs/2203.03466>.
- Liu Yang, Kangwook Lee, Robert Nowak, and Dimitris Papailiopoulos. Looped transformers are better at learning learning algorithms. In *International conference on learning representations*, volume 2024, pp. 42195–42214, 2024.

APPENDIX OUTLINE

The appendix provides the experimental details, derivations, and additional results supporting the main text.

Appendix A describes the common training setup, hardware, and architecture definitions for our compute-optimal experiments. It details the choices of block allocation, core-pass count, growth target, and growth timing, and derives the relationships between recurrence, compute, and tokens per parameter. It also gives the full recipe-fitting procedure: tuning a base model, selecting the token allocation and growth fraction, fitting the learning-rate scaling rule, and training the scaling ladders. The settings used for each architecture are listed in Table 9.

Appendix B presents additional compute-optimal scaling results and ablations. These isolate the components of the boundary operator, test block allocation, and compare efficiency in FLOPs and training time. Further experiments examine architecture-specific hyperparameter tuning, model shape, width-only scaling, random recurrence, test-time core passes, and optimizer choice, showing how these decisions affect constant-factor and exponent improvements.

Appendix C examines corpus transfer and downstream performance. It compares scaling on FineWeb and FineWeb-Edu, specifies the CORE accuracy and answer-NLL evaluation protocols, and analyzes downstream scaling and task-level differences at matched pretraining loss. It then describes the taskwise calibration used to forecast CORE accuracy and evaluate the held-out large-model extrapolation, together with the scope and limitations of these comparisons.

Appendix D extends the data-constrained experiments across data-repetition levels, weight decay, and architectural controls. It examines how repetition changes the preferred number of core passes, how weight decay and recurrence interact, and how hyperparameters transfer across recurrence counts. Comparisons of tied and untied models, with and without the boundary operator, distinguish the roles of weight sharing and additional depth.

A EXPERIMENT DETAILS FOR COMPUTE-OPTIMAL SCALING

We explain the training protocols we use for Section 4 and 5. The common setup is in Table 1.

Hardware. Most runs use one node with eight H100 GPUs. The largest $d26$ extrapolation run uses two such nodes.

Setting	Protocol
Context length	2048 tokens
Tokenizer	GPT-2 vocabulary of 50,257, padded to 50,304
Global batch	524,288 tokens
Shape rule	Model name $d\ell$ denotes nominal/reference depth ℓ and width $w = 128\ell$
Optimizer	Muon for matrix parameters; AdamW for embeddings and the language-model head.

Table 1: Fixed architecture and training protocol. Architecture-specific hyperparameters and token budgets are reported in the following subsections.

A.1 ARCHITECTURE

All models use the pre-norm decoder-only transformer, with RoPE, SwiGLU, and QK normalization. We use no biases or learned normalization gains, so trainable matrices are all two-dimensional. Additional RMS normalizations follow the token embedding and precede `lm_head`. Attention and MLP output projections and `lm_head` are initialized to zero; the token embedding is initialized normally, while the attention Q/K/V and MLP input matrices use uniform initialization.

Following Karpathy (b), $d\ell$ names a model by nominal/reference depth ℓ . Vanilla is an unsplit dense stack, so its stored and executed depths are both ℓ . The architectures are defined in Section 3. Parameters for different architectures and reference depths are shown in Table 2.

Depths	P/C/D	Exec. Depth	Width	Vanilla	Loop-Grow	Untied-2	Untied-Grow
$d6$	2/2/2	$8 \rightarrow 12$	768	120 M	120 M	130 M	160 M
$d8$	2/3/3	$11 \rightarrow 17$	1,024	210 M	210 M	240 M	320 M
$d10$	3/4/3	$14 \rightarrow 22$	1,280	330 M	330 M	410 M	580 M
$d12$	4/4/4	$16 \rightarrow 24$	1,536	490 M	490 M	610 M	830 M
$d14$	4/5/5	$19 \rightarrow 29$	1,792	730 M	730 M	920 M	1.3 B
$d16$	5/6/5	$22 \rightarrow 34$	2,048	1.0 B	1.0 B	1.3 B	2.0 B
$d18$	6/6/6	$24 \rightarrow 36$	2,304	1.4 B	1.4 B	1.8 B	2.5 B
$d20$	6/7/7	$27 \rightarrow 41$	2,560	1.8 B	1.8 B	2.4 B	3.5 B
$d22$	7/8/7	$30 \rightarrow 46$	2,816	2.4 B	2.4 B	3.2 B	4.7 B
$d24$	8/8/8	$32 \rightarrow 48$	3,072	3.0 B	3.0 B	3.9 B	5.7 B
$d26$	8/9/9	$35 \rightarrow 53$	3,328	3.8 B	3.8 B	5.0 B	7.4 B

Table 2: **Model depths, layer splits, executed depths, and stored parameters.** P/C/D gives the prelude/core/coda layer split. Executed depth shows the transition from $K = 2$ to $K = 4$; fixed Loop-2 and Untied-2 use the first value and Loop-Grow and Untied-Grow use the second. Vanilla has executed depths equal to physical depths. Operator-1 and Loop-2 have the same stored parameters as Vanilla, while Loop-2 has the same effective parameters as Untied-2. Loop-Grow shares the Vanilla stored parameters, whereas Untied-Grow allocates all four untied core copies from the start.

Compute budget estimation Plotted compute comes from the model FLOP estimator, which includes matrix multiplications and attention computation; expressions of the form $6TN_{\text{eff}}$ below are leading-order allocation identities.

Hardware consideration Operator-1 differs from Vanilla only with the boundary operator, adding one RMS Norm and vector additions. Untied-2 and Loop-2 have the same computational graph. The added arithmetic and memory communication is relatively small. For a runtime comparison, see Figure 15b.

Multipliers Suppose Θ is parameter and F is one layer (e.g. MLP, attention). Multiplier α for Θ is defined as $F(\alpha\Theta)$. Multiplier is not trainable. Multipliers follow an equivalence relationship with learning rate and initialization scales, known as ABC-parameterization (Yang et al., 2022). In this paper, we define two multipliers of interest. Output multiplier (OM) is the multiplier to the unembedding or readout layer. Residual multiplier (RM) is the shared weight multiplier to the MLP down projection and attention output projection.

A.2 COMPUTE-OPTIMAL DECISIONS FOR LOOPING

Looped models introduce four choices beyond the base Transformer recipe: how to allocate blocks across the prelude, core, and coda; how many times to apply the core; how to increase that count during training; and how to allocate tokens after growth. We organize the evidence around these decisions. The experimental controls differ across the sweeps: the fixed-token ladders compare model sizes and recurrence counts at matched compute, whereas the fixed-anchor sweeps trade training tokens for recurrence within each compute budget. Appendix A.3 derives the TPP relationships used to interpret the latter sweeps.

A.2.1 ALLOCATING BLOCKS ACROSS THE PRELUDE, CORE, AND CODA

We sweep the core size of tied, two-pass models at several stored depths, training each model for 1B tokens with the tuned $d8$ recipe. Because a larger core executes more blocks per token, we compare losses against a common compute frontier rather than only within a fixed stored depth (Figure 7, left). The preferred core fraction varies little with depth, supporting a family whose three regions grow in proportion. We use a simple shared allocation: divide the blocks as evenly as possible across the prelude, core, and coda, assigning remainders first to the core and then to the coda.

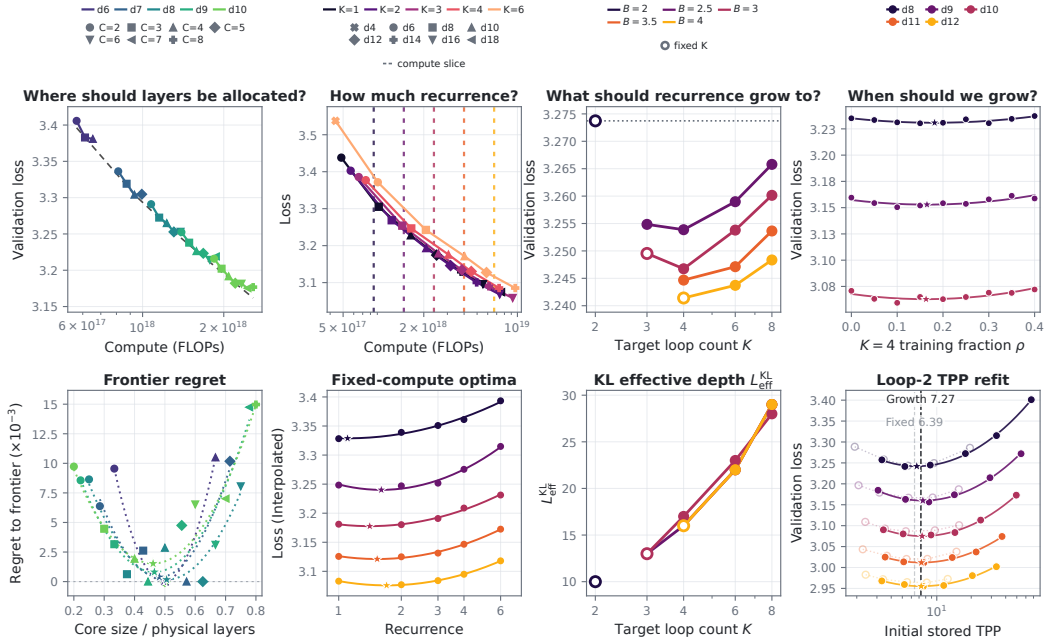


Figure 7: **Allocation, recurrence, and growth sweeps.** **Left:** loss versus compute for the 1B-token core-allocation sweep, followed by regret to the compute frontier versus core fraction; curves fit each depth’s regret and stars mark their minima. **Middle left:** fixed-token recurrence ladders and fixed-compute slices interpolated in log loss versus log compute. **Middle right:** loss and logit-KL effective depth versus the growth target, with filled points for growth from $K = 2$ and hollow points for fixed- K anchors. **Right:** Loop-2 growth-fraction sweeps at the depth-matched TPP-6 compute budgets, followed by the initial-stored-TPP refit; filled points use growth and hollow points are fixed controls. These panels summarize separate experiments with the controls described in the text.

A.2.2 CHOOSING THE NUMBER OF CORE PASSES

Fixed-token ladders favor one to two passes. For each core-pass count $K \in \{1, 2, 3, 4, 6\}$, we train a ladder of model sizes on 1B tokens with the Operator-1 base recipe. We interpolate each ladder’s loss at common compute budgets and fit loss against K . For both tied and untied models, the fitted optimum stays between one and two passes across the measured budgets; $K = 3, 4, 6$ give higher loss (Figure 8). We use $K = 2$ for the fixed-recurrence variants on this basis.

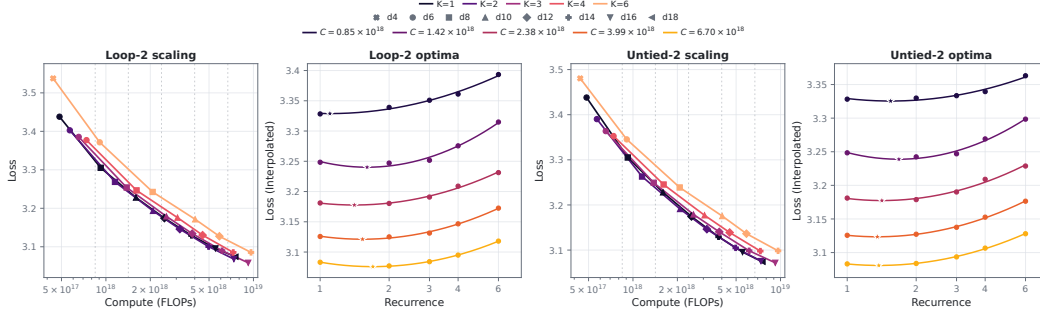


Figure 8: **Recurrence choice in fixed-token ladders.** Every model trains on 1B tokens with the Operator-1 base recipe. For each family, the loss–compute ladders are interpolated at common budgets, and quadratics in log recurrence and log loss locate the fitted optima (stars). The two families share the $K = 1$ ladder and are FLOPs-matched at each depth and recurrence.

At fixed anchor size, more compute favors more passes. We next hold the anchor depth fixed, sweep the same K values, and reduce training tokens as K increases to match compute. Across anchors $d8$ – $d12$, the fitted optimal count K^* increases with the budget (Figure 9, left), consistent with Prairie et al. (2026). Thus, at fixed anchor size, larger budgets favor allocating some compute to extra passes. This comparison does not test whether extra passes outperform increasing model size.

Figure 9 (right) summarizes the optima across anchors using baseline tokens per stored parameter, TPP_{K1} : the tokens affordable at $K = 1$ under the same budget, divided by that model’s stored parameter count. Define $N_{c,K}$ as the *compute-active parameter count*: the prelude, coda, and output-head matrix parameters counted once, plus the core matrix parameters counted K times, even when shared. This count excludes the input embedding lookup; $N_{c,1}$ is the count for the same anchor with one core pass. The expansion is $E_K = N_{c,K}/N_{c,1}$. At K^* , we fit

$$\begin{aligned} E_{K^*} &= 0.82 \text{TPP}_{K1}^{0.15} \quad (\text{tied}), \\ E_{K^*} &= 0.83 \text{TPP}_{K1}^{0.15} \quad (\text{untied}). \end{aligned} \tag{4}$$

The rightmost panels show that tokens per stored parameter at the optimum also increase with baseline TPP, but sublinearly: some of the additional budget goes to recurrence. Appendix A.3 gives these TPP fits and derives their relation to compute expansion.

A.2.3 CHOOSING THE GROWTH TARGET AND TRANSITION

Grow from two to four passes. Starting from the tied $d8$ model with a $2/3/3$ block split, we sweep growth targets $K \in \{3, 4, 6, 8\}$ over 1B training tokens. At each compute budget, the transition is chosen to match the budget; thus a larger target is used for a smaller fraction of the run. Growing to four passes gives the lowest loss at every tested budget (Figure 7, middle right). Larger targets increase logit-KL effective depth without consistently improving loss. We adopt the $2 \rightarrow 4$ schedule for the growth variants: Loop-Grow reuses its tied core more times, while Untied-Grow duplicates the trained untied cores. Deep Vanilla Grow applies the same duplication schedule to plain Transformer blocks with the same prelude–core–coda allocation.

Choose the transition at fixed depth and compute. Let ρ be the fraction of training tokens processed *after* growth, so a larger ρ means an earlier transition. With the target fixed at four passes, we sweep ρ at three anchor depths and six compute budgets for Loop-Grow, and at the same anchors

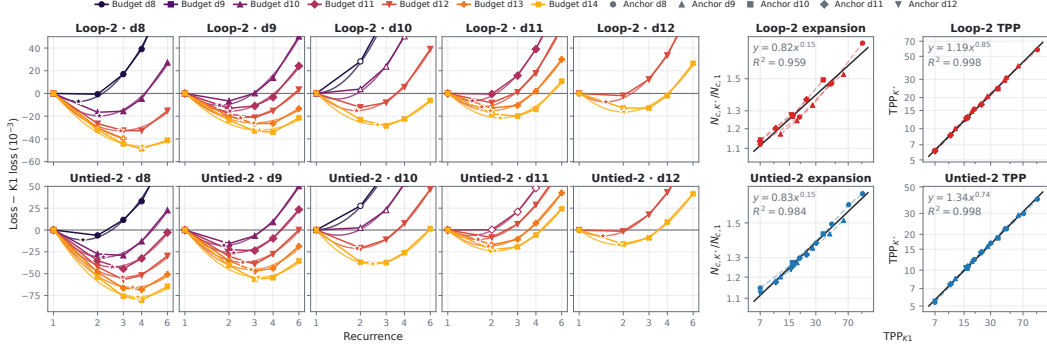


Figure 9: **Recurrence choice at fixed anchor size and compute.** The first row uses tied cores and the second untied cores. **Left:** loss relative to $K = 1$ versus recurrence, with anchor depth increasing across columns; each series holds compute fixed by adjusting tokens. Quadratic fits locate K^* . **Right:** compute-active expansion and tokens per stored parameter at the fitted optimum versus the $K = 1$ baseline TPP. Hollow points mark optima outside the measured recurrence grid and are excluded from the fitted laws.

with three budgets each for Untied-Grow. Tokens are adjusted to keep each depth–budget pair at fixed compute. Quadratic fits to validation loss locate ρ^* . Larger budgets generally favor earlier growth, but the minima are broad (Figures 10a and 10b, top rows).

Across depths and budgets, the token allocation at the fitted optimum follows an affine relationship,

$$\text{TPP}_{K_{\rho^*}} = a \text{TPP}_{K2} + b, \quad (5)$$

where both TPP quantities use the initial $K = 2$ stored parameter count: TPP_{K2} is the allocation without growth at the same compute, and $\text{TPP}_{K_{\rho^*}}$ is the allocation with the fitted transition. The coefficients are $(a, b) = (0.902, 0.216)$ for Loop-Grow and $(0.883, 0.103)$ for Untied-Grow, with $R^2 > 0.9999$ for both. As derived in Appendix A.3, this fit implies a compute expansion that rises and then plateaus with baseline TPP, motivating a nearly constant growth fraction at sufficiently high TPP. At the high-TPP end of the measured sweeps, the fitted fractions are approximately 0.23–0.24 for Loop-Grow and 0.26–0.32 for Untied-Grow. The exact conversion from expansion to ρ depends slightly on the model’s block allocation.

For Deep Vanilla Grow, we sweep one fixed-compute budget at each of $d8$, $d9$, and $d10$, using each anchor’s fixed Deep Vanilla TPP-6 budget. The fitted fractions are 0.556, 0.559, and 0.516, respectively, with similarly shallow minima near half of training (Figure 10c).

A.2.4 ALLOCATING TOKENS AND TESTING RECIPE SENSITIVITY

Growth increases average compute per token, so the optimal token allocation must be refitted. We repeat the iso-compute sweeps with growth in place (Figure 10d, left and middle). The figure uses $6T^2/C$, where T is training tokens and C is training compute; under the leading-order relation $C = 6TN_{c,\text{eff}}$, this coordinate equals tokens per training-averaged compute-active parameter. The mean optimum in this coordinate rises from 6.39 to 6.97 for tied growth and from 7.43 to 8.97 for untied growth. This coordinate differs from tokens per initial *stored* parameter, which we use to specify the ladder recipes in Table 9. Across the measured budgets, growth favors a smaller initial model trained on more tokens, although the loss curves are flat near their minima.

We test the cost of simplifying both the token allocation and the transition for Untied-Grow (Figure 10d, right). The ablation crosses initial stored TPP 8 versus 6 with the fitted per-size ρ versus a constant $\rho = 0.30$. The matched-loss compute multipliers remain close to one: reusing TPP 6 changes the estimated compute requirement by less than about 5%, and replacing fitted ρ with 0.30 changes validation loss by -0.0009 to $+0.0019$. Thus, within this ablation, precise per-size tuning has little benefit. The saturation of the fitted expansion and the broad loss minima support a simple recipe with a fixed growth fraction and a rounded TPP.

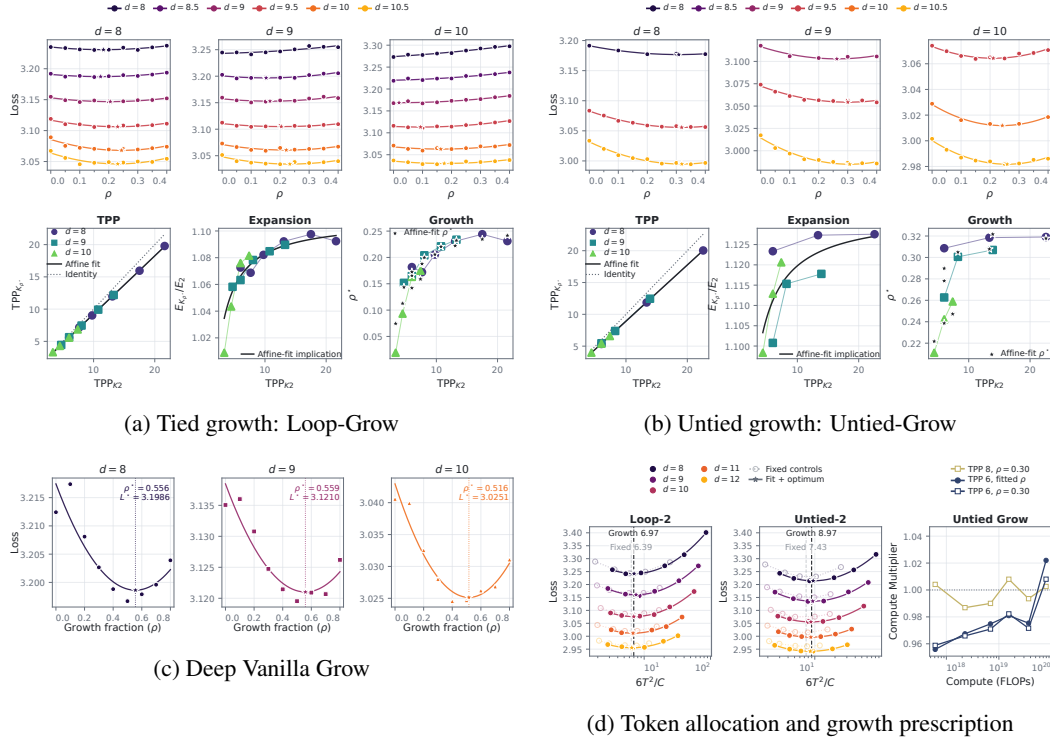


Figure 10: **Growth timing, token allocation, and recipe sensitivity.** (a,b) Top rows show validation loss versus the post-transition token fraction ρ at three anchor depths; colors identify fixed-compute budgets, curves are quadratic fits, and stars mark bracketed minima. Bottom rows show initial-stored-parameter TPP at the fitted transition, compute-active expansion, and optimal growth fraction versus fixed- $K = 2$ TPP. Colors and markers identify anchor depth; black curves and stars show the affine fit and its implications (Appendix A.3). (c) Deep Vanilla Grow timing sweeps at each anchor's fixed Deep Vanilla TPP-6 budget, with fitted minima marked by stars. (d) Left and middle: loss versus $6T^2/C$ for tied and untied growth, where T is tokens and C is compute. Filled points use growth, hollow points are fixed controls, stars mark fitted minima, and dashed lines mark mean optima. Right: the Untied-Grow prescription ablation at baseline depths $d6-d16$, even. The multiplier is the TPP-8, fitted- ρ baseline's compute divided by each alternative's compute at matched loss, using the two nearest measured points in log-compute/log-loss space.

A.3 TPP RELATIONS FOR RECURRENCE AND GROWTH

The sweeps in Appendix A.2 compare different recurrence counts at fixed training compute. This section derives how those comparisons change tokens per stored parameter. We distinguish stored parameters, which determine TPP, from compute-active parameters, which determine the leading-order compute cost. The identities below use $C = 6DN_c$; the plotted compute budgets use the model FLOP estimator, including attention, as described in Appendix A.1.

A.3.1 FIXED RECURRENCE

For recurrence K , let D_K be the number of training tokens and $N_{s,K}$ the stored parameter count. The compute-active count $N_{c,K}$, defined in Appendix A.2.2, counts weight-matrix parameters once per application, including all K core passes, and excludes the input embedding lookup. The subscript 1 denotes the same anchor model with one core pass. Define the compute-active and stored-parameter expansions relative to this baseline by

$$\text{TPP}_K = \frac{D_K}{N_{s,K}}, \quad E_K = \frac{N_{c,K}}{N_{c,1}}, \quad S_K = \frac{N_{s,K}}{N_{s,1}}. \quad (6)$$

At a fixed compute budget, $D_K N_{c,K} = D_1 N_{c,1}$, so $D_K = D_1/E_K$. Dividing by the stored parameter count gives

$$\text{TPP}_K = \frac{D_1/E_K}{S_K N_{s,1}} = \frac{\text{TPP}_{K1}}{E_K S_K}. \quad (7)$$

Increasing recurrence therefore reduces TPP through the increased compute per token and, for untied models, through the increased stored parameter count. Tied recurrence reuses one core, so $S_K = 1$. Untied recurrence stores a separate core for each pass; when block parameters dominate, S_K approaches E_K . With an approximately equal prelude-core-coda allocation, E_K approaches $(K+2)/3$ for either family. At finite size, stored and compute-active counts differ, including because embedding lookup contributes stored parameters without the same matrix-multiplication cost.

Equation 7 also holds at the fitted optimum K^* . Together with the expansion fits in Equation 4, it motivates a power-law relationship between optimal TPP and baseline TPP. The measured fits in Figure 9 are

$$\begin{aligned} \text{TPP}_{K^*} &= 1.19 \text{TPP}_{K1}^{0.85} \quad (\text{tied}), \\ \text{TPP}_{K^*} &= 1.34 \text{TPP}_{K1}^{0.74} \quad (\text{untied}). \end{aligned} \quad (8)$$

These are empirical fits; the finite-size stored-parameter expansion enters the untied relation through S_{K^*} .

A.3.2 GROWTH FROM TWO TO FOUR PASSES

Let ρ be the fraction of tokens processed after the transition from $K = 2$ to $K = 4$. The token-weighted average recurrence and compute-active parameter count are

$$\begin{aligned} K_\rho &= 2(1-\rho) + 4\rho, \\ N_{c,K_\rho} &= (1-\rho)N_{c,2} + \rho N_{c,4}, \\ C &= 6D_{K_\rho} N_{c,K_\rho}. \end{aligned} \quad (9)$$

Here N_{c,K_ρ} denotes a training average, rather than a model executing a fractional number of passes. For growth, both TPP coordinates use the *initial* stored parameter count $N_{s,2}$:

$$\text{TPP}_{K_\rho} = \frac{D_{K_\rho}}{N_{s,2}}, \quad \text{TPP}_{K2} = \frac{D_{K2}}{N_{s,2}}. \quad (10)$$

The fixed- $K = 2$ baseline spends the same compute as the growth run, so $D_{K_\rho} N_{c,K_\rho} = D_{K2} N_{c,2}$. Using $E_K = N_{c,K}/N_{c,1}$ gives

$$\text{TPP}_{K_\rho} = \frac{E_2}{E_{K_\rho}} \text{TPP}_{K2}, \quad \frac{E_{K_\rho}}{E_2} = 1 + \rho \left(\frac{E_4}{E_2} - 1 \right). \quad (11)$$

There is no extra stored-parameter expansion factor here because both TPP quantities use the same initial denominator, including for untied growth.

At the fitted transition, the affine law in Equation 5 implies

$$\begin{aligned}\frac{E_{K_{\rho^*}}}{E_2} &= \frac{\text{TPP}_{K_2}}{a \text{TPP}_{K_2} + b}, \\ \rho^* &= \frac{E_{K_{\rho^*}}/E_2 - 1}{E_4/E_2 - 1}.\end{aligned}\tag{12}$$

When b is small relative to $a \text{TPP}_{K_2}$, the optimal compute expansion approaches $1/a$. This yields the rise and plateau in the bottom rows of Figures 10a and 10b and explains why the fitted growth fraction becomes weakly dependent on TPP. Equal baseline TPP predicts equal compute expansion, but need not predict exactly equal ρ^* : the conversion also depends on E_4/E_2 , which varies with the integer block allocation and non-core compute. When the three regions approach equal proportions and block compute dominates, $E_4/E_2 \rightarrow 3/2$, giving the limiting prescription $\rho^* \rightarrow 2(1/a - 1)$.

A.4 COMPUTE-OPTIMAL RECIPE: STAGED SWEEPS AND FITS

The recipe follows the four stages of Section 4.1: tune base hyperparameters at $d8$ (Stage 1), fit tokens per stored parameter (Stage 2), choose a fixed growth fraction ρ for growth variants (Stage 3), and fit the learning-rate scaling rule (Stage 4). Fixed-recurrence models skip Stage 3. The block allocation, recurrence count, and growth target are chosen as described in Appendix A.2. For growth, we recommend retaining the Stage 2 TPP and calibrating ρ once: the sensitivity tests in Appendix A.2.4 show little benefit from retuning TPP or refitting the transition across sizes.

A.4.1 STAGE 1: TUNE A BASE MODEL

We first tune every hyperparameter at base size, $d8$, on a fixed budget of 1B tokens. We sweep one hyperparameter at a time and hold the local optimum into next hyperparameter sweep, similar to (Wen et al., 2026). We show the hyperparameter grids in Table 3. Partial tuning trajectories are shown in Figure 11.

Table 4 lists the tuned losses of compute-matched arms and justifies our architecture choices. The final mixing operator ρ , which is missing in previous looped architectures, improves the loss by a fair amount. In addition, we find that untying the weights have a benefits over tying.

Tuning each architecture separately matters: transferring Vanilla’s tuned recipe to the looped variants costs $7\text{--}26 \times 10^{-3}$ in loss (Table 6), emphasizing the importance of separate tuning for each architecture.

Architecture	ϕ (core boundary)	ρ (before coda)	Tuned loss	Runtime (min)
Deep Vanilla	h	h	3.2772	6.49
Loop-2	$\text{BO}(h, e)$	$\text{BO}(h, e)$	3.2704	6.54
Loop-2-no-coda-inj	$\text{BO}(h, e)$	$\text{Norm}(h)$	3.2912	6.45
Untied-2	$\text{BO}(h, e)$	$\text{BO}(h, e)$	3.2563	6.54
Untied-2-no-coda-inj	$\text{BO}(h, e)$	$\text{Norm}(h)$	3.2731	6.53
Deep Vanilla + norm	$\text{Norm}(h)$	$\text{Norm}(h)$	3.2781	6.48
Deep Vanilla + injection	$h + \alpha e$	$h + \alpha e$	3.2690	6.52

Table 4: **Boundary-operator ablations at the base size.** Architectures and boundary maps follow Equation 2 and Equation 3. Each model is tuned independently and trained on 1B tokens at width 1024 and executed depth 11. Untied-2 achieves the lowest loss; removing either BO component or the coda injection increases loss. Runtime is the corresponding run’s logged training time in minutes on eight H100 GPUs, excluding evaluation and initial compilation/warmup.

A.4.2 STAGE 2: FIT THE OPTIMAL TOKENS PER PARAMETER

With the base recipe frozen, we fit the split between model size and training tokens at fixed compute (Hoffmann et al., 2022). We express this split as tokens per stored parameter, $\text{TPP} = T/N$. Compute depends instead on the compute-active count N_c , which counts a shared core once per application (Appendix A.2.2). At the same anchor size and recurrence, Loop-2 and Untied-2 have the

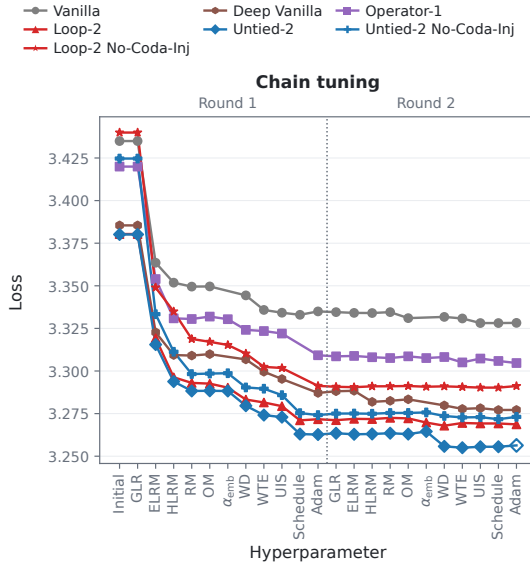


Figure 11: **Chain tuning at $d8$ on 1B tokens.** Each curve is one architecture’s tuning chain. We adopt the new hyperparameters if it’s 10^{-3} better than the current best. Loss continuously drops with more tuning. ELRM, HLRM, WD, Schedule, WTE, Adam bring most drops. The second round still improves loss at some HP because the optimum changes as other HPs change. The loss plateaus near the end of second round. The hollow marker denotes that more than one discrete HPs improve the performance and stacking both can potentially improve performance more.

HP	Definition	Initial HP
$\times \{1/16, 1/8, \dots, 8\}$		
GLR	global learning rate	0.04
RM	residual multiplier	0.5
OM	output multiplier	1
α_{emb}	injection weight	1
$\times \{1/8, 1/4, \dots, 16\}$		
WTE init.	embedding init. scale	0.08
UIS	input-matrix init. scale	0.25
$\times \{1, 2, \dots, 128\}$		
ELRM	embedding LR multiplier	0.02
HLRM	head LR multiplier	0.02
$\{0\} \cup \times \{1/32, \dots, 2\}$		
WD	weight decay	0.1
<i>Discrete</i>		
Schedule	warmup $\{0, 5, 10, 20\}$; warmdown $\{.2, .6, .8, 1\}$	40; .4
Adam	$\beta_1 \{.9, .95\}$; $\beta_2 \{.90, .98, .99\}$; $\epsilon \{10^{-8}, 10^{-6}\}$	$(.8, .95)$; 10^{-10}

Table 3: **Tuning Grid for Chain Tuning.** There’s nine sweeps (eight for Vanilla) in each round. Each sweep has eight parallel runs. For numerical values, grid is centered at the optimal values and span 2x grid in the first round and $\sqrt{2}$ x grid in the second round. For discrete values, there are two or three HPs. We change one HP in each run. If more than one HPs are better than the baseline, we apply both HPs in the next sweep. Discrete sweeps use the same candidates in both rounds.

Architecture	GLR	ELRM	HLRM	RM	OM	α_{emb}	WD	WTE	UIS	WU	WDR	β_1	β_2	ϵ
Vanilla	0.04	0.453	0.113	0.25	0.5	—	0.071	0.007	0.063	40	0.6	0.8	0.95	10^{-10}
Deep Vanilla	0.04	0.16	0.057	0.5	1	—	0.1	0.005	0.5	5	0.8	0.8	0.99	10^{-8}
Operator-1	0.04	0.905	0.08	0.5	1	1	0.05	0.113	0.354	0	0.8	0.8	0.98	10^{-10}
Loop-2	0.04	0.32	0.113	0.25	1	0.707	0.05	0.02	0.044	40	1	0.8	0.95	10^{-10}
Untied-2	0.04	0.16	0.16	0.25	1	1	0.071	0.01	0.354	40	1	0.8	0.99	10^{-8}

Table 5: **Base-tuned hyperparameters by architecture.** Recipes are selected by two rounds of chain tuning at the base size on 1B tokens. Dashes indicate inapplicable hyperparameters.

Architecture	Vanilla-recipe loss	Δ vs. Vanilla (10^{-3})	Own-recipe loss	Transfer regret (10^{-3})
Vanilla	3.3275	+0.0	3.3279	−0.3
Deep Vanilla	3.2869	−40.6	3.2772	+9.8
Operator-1	3.3135	−14.1	3.3057	+7.8
Loop-2	3.2777	−49.9	3.2704	+7.2
Untied-2	3.2697	−57.9	3.2563	+13.3

Table 6: **Transfer probe from the Vanilla recipe.** Each available transfer probe is trained once at $d8$ on 1B tokens with Vanilla’s tuned recipe (injection variants keep the default α_{emb}) and compared with its own tuned recipe. Transfer regret is the transferred-recipe loss minus the own-recipe loss, so positive values measure the benefit of variant-specific tuning; deltas and regrets use unrounded losses. Dashes indicate unavailable transfer measurements for Deep Vanilla, whose own-recipe loss is reported at matched executed depth 11 and width 1024. Vanilla’s -0.3×10^{-3} is below the adoption threshold and reflects run-to-run noise.

same compute per token, but Untied-2 stores more parameters; its stored-parameter TPP is therefore lower at matched compute. We use stored-parameter TPP throughout this stage.

Iso-compute sweeps. We name each budget by the reference model size whose training at TPP 7 defines its compute cost. We use five budgets, $d8$ through $d12$. At each budget, we train five model sizes around the anchor and adjust the token count to match compute. We fit a quadratic in log loss against log TPP and take its vertex as the budget’s optimal TPP (Figure 12). The vertex losses describe the fitted frontier under the base recipe. We defer scaling-law comparisons to the completed ladders, after fitting growth timing and learning-rate scaling.

Is the optimal TPP stable across scale? Hoffmann et al. (2022) found a roughly constant optimal TPP across compute, and we test the same property for each architecture. Table 7 reports the mean of the five vertices together with a scale-shift test. No variant rejects a scale-invariant TPP at $\alpha = 0.05$. We adopt the rounded mean in Table 7: 5 for Vanilla and 6 for Operator-1, Loop-2, and Untied-2. Every looped variant prefers a slightly higher TPP than Vanilla, indicating better parameter efficiency.

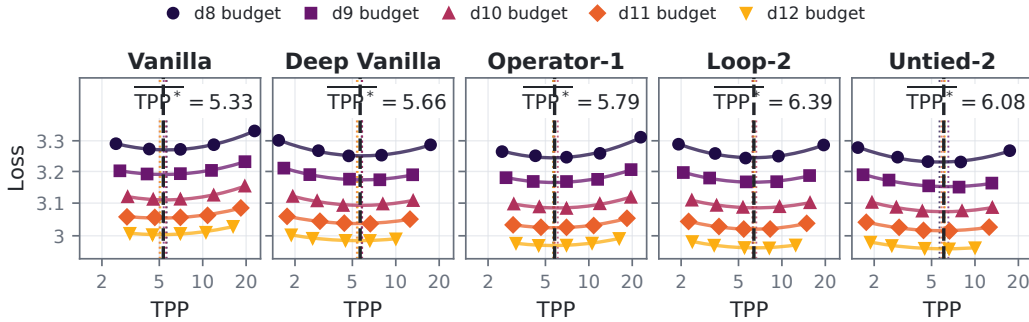


Figure 12: **Iso-compute TPP fits and fitted scaling laws.** Loss against tokens per parameter for Vanilla, Loop-2, and Untied-2 and a scaling law from fitted vertices. Each color is one fixed compute budget, the cost of the $d8$ – $d12$ model at TPP 7; curves are quadratic fits of log loss against log TPP, dotted lines mark the per-budget vertices, and the dashed line is their mean. Per-budget vertices for the three architectures are plotted in the last subfigure as a scaling law.

Architecture	Mean TPP*	Rounded	p	L_{d8}^*	L_{d10}^*	L_{d12}^*
Vanilla	5.33	5	0.087	3.270	3.110	3.003
Deep Vanilla	5.66	6	0.109	3.251	3.093	2.985
Operator-1	5.79	6	0.923	3.244	3.085	2.971
Loop-2	6.39	6	0.588	3.244	3.085	2.964
Untied-2	6.08	6	0.760	3.230	3.073	2.961

Table 7: **Compute-optimal TPP by architecture.** Mean TPP* averages the five iso-compute vertices; rounded TPP is the value used in the scaling ladders. The loss columns report the fitted vertex loss at budgets $d8$, $d10$, and $d12$; the best loss in each budget is bold. The scale-shift test fits $\log \text{TPP}^* = a + \beta \log C$ over all five budgets and reports the two-sided p -value of $H_0: \beta = 0$. No variant rejects a scale-invariant TPP at $\alpha = 0.05$.

A.4.3 STAGE 3: CHOOSE A FIXED GROWTH FRACTION

For the $2 \rightarrow 4$ growth schedule, let ρ denote the fraction of training tokens processed *after* growth; the transition therefore occurs after fraction $1 - \rho$. We recommend reusing the Stage 2 TPP to set the reference compute budget and choosing one fixed ρ for the ladder. At each of three small-model anchors ($d8$ – $d10$), sweep ρ while adjusting tokens to preserve that anchor’s compute budget, then fit a quadratic to validation loss versus ρ (Figure 13). Average the three fitted optima and round to one decimal place. The broad minima support this simple choice without a growth-specific TPP refit or per-size transition rule.

We use this recipe for Deep Vanilla Grow: the mean optimum 0.544 gives $\rho = 0.5$, with the Deep Vanilla TPP-6 reference budget. The reported Untied Grow and Loop Grow ladders instead use fitted per-size fractions from Appendix A.2.3 and refitted reference TPP values of 8 and 7, respectively (Table 9). We expect little benefit from these refinements: the Untied Grow ablation finds less than about 5% change in matched-loss compute when reusing TPP 6, and loss changes of -0.0009 to $+0.0019$ when replacing the fitted fractions with $\rho = 0.30$ (Figure 10d, right).

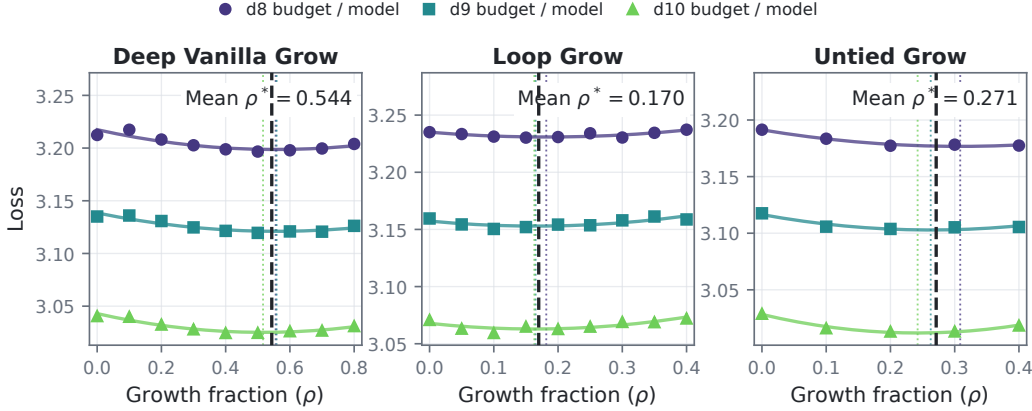


Figure 13: **Choosing a fixed growth fraction.** Left to right: Deep Vanilla Grow, Loop Grow, and Untied Grow. Each series shows validation loss versus the post-growth token fraction ρ for a $d8$, $d9$, or $d10$ model at its corresponding fixed-compute budget; tokens are adjusted as ρ varies. Points are measured runs and curves are quadratic fits. Colored dotted lines mark the fitted optima, and black dashed lines mark their arithmetic means: 0.544, 0.170, and 0.271, respectively. The panels use different axis ranges.

A.4.4 STAGE 4: FIT THE LEARNING-RATE RULE

With token allocation and growth timing fixed, we determine which base hyperparameters need to change with model size. We first screen Vanilla with one-dimensional sweeps over $d8$ – $d12$, using the Stage 2 token allocation at each size (Figure 14a). All hyperparameters are swept at $d8$, $d10$, and $d12$; the intermediate sizes cover the global learning rate (GLR), head learning-rate ratio, output and residual multipliers, weight decay, and warmdown ratio. Scalar settings use $\{1/4, 1/2, 1, 2, 4\}$ times their base values, while schedule and Adam settings use discrete grids. The preferred GLR falls from 0.04 at $d8$ – $d10$ to 0.02 at $d11$ – $d12$, whereas several other settings have nearly flat loss curves.

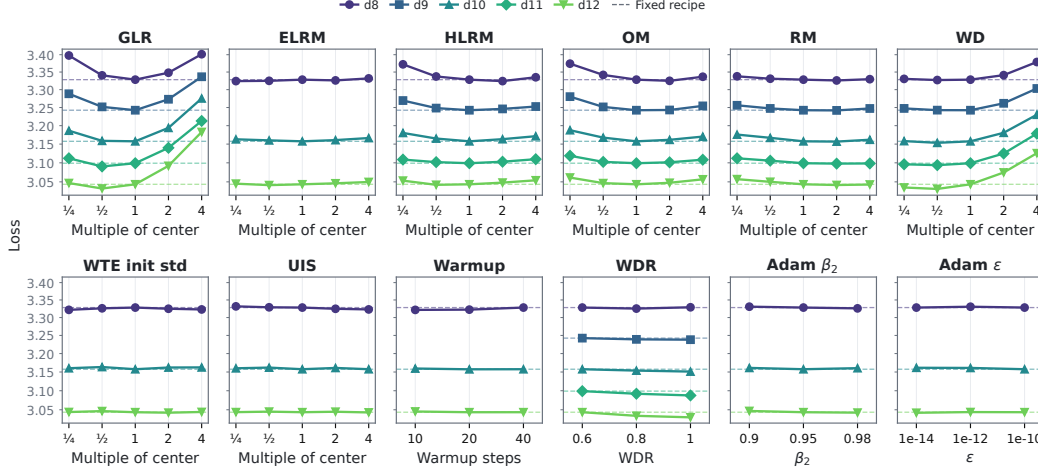
We then sweep the global learning rate (GLR), output multiplier (OM), residual multiplier (RM), and weight decay (WD) at $d8$ – $d10$ for the architecture families (Figure 14b). For each hyperparameter and size, we fit a quadratic to log validation loss against the log multiplier and take its minimum as the estimated optimum. Regressing the log optima against log stored parameter count gives the drift exponent β in Table 8. The table also reports the loss penalty for a factor-of-two change and the full loss range of each sweep. Among these four hyperparameters, GLR has the largest sensitivity under the constant recipe. We therefore use the anchored learning-rate rule

$$\text{GLR}(N) = \text{GLR}_{d8} \left(\frac{N}{N_{d8}} \right)^\beta, \quad (13)$$

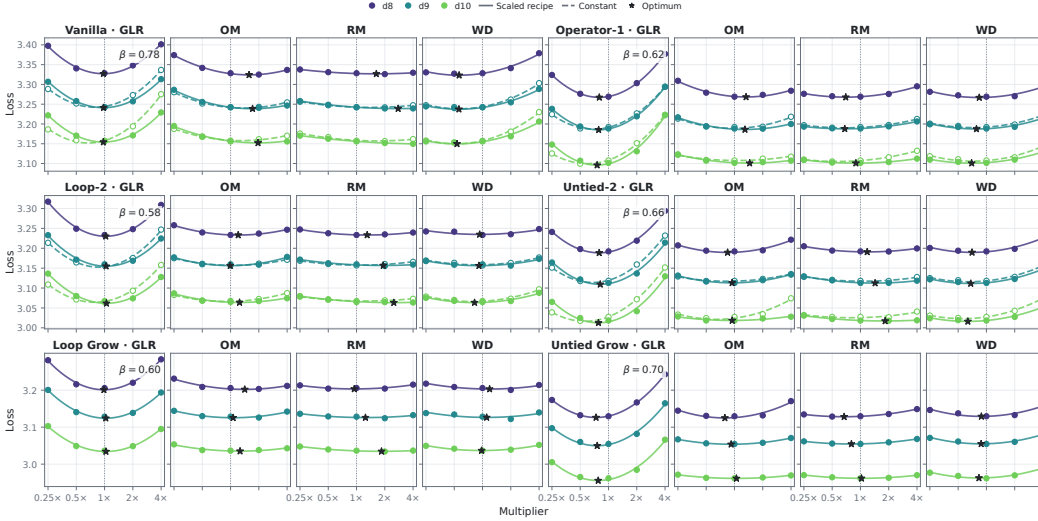
where N is the initial stored parameter count and N_{d8} is its value for that architecture’s base model. The exponent is negative, so the learning rate decreases as models grow. For growth variants, we center the learning-rate sweeps on the corresponding fixed-recurrence rule and measure the remaining size dependence. The exponents and base learning rates used in the reported ladders are listed in Table 9; the exponents are specified to one decimal place.

We repeat the hyperparameter sweeps around the scaled recipe to check transfer across sizes (Figure 14b and Table 8). Scaling GLR also changes the preferred values of the other hyperparameters,

so drift measured under a constant recipe need not imply that an additional scaling rule is useful. In the Vanilla ladder ablation, adding output-multiplier or weight-decay scaling changes the matched-loss compute multiplier by at most about 3% relative to GLR scaling alone (Figure 16a). We therefore use the fitted GLR rule and keep the other hyperparameters at their base-tuned values. This gives a simple recipe for comparing architectures without carrying a separate scaling law for every hyperparameter.



(a) Vanilla hyperparameter sensitivity across scale



(b) Architecture-specific sweeps under constant and scaled recipes

Figure 14: **Hyperparameter sensitivity and transfer across scale.** (a) One-dimensional Vanilla sweeps at $d8$ – $d12$; dashed lines show the fixed-recipe loss. Scalar settings use $\{1/4, 1/2, 1, 2, 4\}$ times the base value; schedule and Adam settings use the displayed grids. ELRM and HLRM are embedding and head learning-rate ratios, WTE init std is the embedding initialization standard deviation, UIS is the uniform initialization scale, and WDR is the warmdown ratio. (b) Sweeps of global learning rate (GLR), output multiplier (OM), residual multiplier (RM), and weight decay (WD) at $d8$ – $d10$. Rows pair Vanilla with Operator-1, Loop-2 with Untied-2, and Loop-Grow with Untied-Grow. Solid and dashed curves are quadratic fits for scaled and constant recipes; stars mark scaled-recipe optima. Positive β annotations use $\text{GLR} \propto N^{-\beta}$, opposite to the signed convention in Equation 13. Fixed-recurrence panels report constant-recipe fits; growth panels show inherited rule magnitudes.

Variant	Recipe	GLR		OM		RM		WD	
		β	regret / range	β	regret / range	β	regret / range	β	regret / range
Vanilla	constant	-0.78 ± 0.06	18.7 / 150.0	-0.37 ± 0.10	3.5 / 17.6	$+0.06 \pm 0.37$	2.3 / 15.2	-0.17 ± 0.11	9.4 / 94.0
	+GLR ($\beta=-0.8$)	-0.00 ± 0.01	15.0 / 73.3	$+0.46 \pm 0.05$	5.9 / 39.8	$+2.73 \pm 0.23$	1.4 / 20.7	-0.11 ± 0.05	4.9 / 53.3
Operator-1	constant	-0.62 ± 0.02	17.9 / 122.4	-0.43 ± 0.55	3.0 / 14.4	-0.41 ± 0.15	3.0 / 26.9	-0.32 ± 0.20	5.6 / 37.8
	+GLR ($\beta=-0.6$)	-0.11 ± 0.02	17.4 / 121.6	$+0.19 \pm 0.17$	3.8 / 21.5	$+0.53 \pm 0.35$	1.4 / 10.8	-0.39 ± 0.05	5.4 / 28.6
Loop-2	constant	-0.58 ± 0.08	16.8 / 110.1	-0.60 ± 0.04	2.3 / 14.9	-0.09 ± 0.38	2.1 / 11.6	-0.40 ± 0.06	7.0 / 41.1
	+GLR ($\beta=-0.6$)	$+0.03 \pm 0.00$	12.2 / 71.5	$+0.08 \pm 0.51$	3.0 / 21.9	$+1.36 \pm 0.19$	3.4 / 16.1	-0.26 ± 0.10	2.7 / 23.2
Loop-Grow	+GLR ($\beta=-0.6$)	$+0.10 \pm 0.05$	12.2 / 66.3	-0.23 ± 0.55	1.4 / 16.9	$+1.40 \pm 0.12$	0.4 / 13.6	-0.41 ± 0.06	3.9 / 15.5
Untied-2	constant	-0.66 ± 0.02	17.3 / 134.1	-0.53 ± 0.26	1.9 / 49.7	-0.73 ± 0.11	$-0.4 / 14.2$	-0.43 ± 0.02	6.2 / 50.4
	+GLR ($\beta=-0.7$)	-0.04 ± 0.07	14.8 / 111.2	$+0.20 \pm 0.13$	2.7 / 10.1	$+0.84 \pm 0.01$	1.7 / 12.8	-0.20 ± 0.03	5.1 / 35.0
Untied-Grow	+GLR ($\beta=-0.7$)	$+0.10 \pm 0.01$	13.1 / 104.3	$+0.53 \pm 0.06$	1.6 / 9.8	$+0.81 \pm 0.02$	2.0 / 8.5	-0.12 ± 0.04	7.2 / 28.2

Table 8: **Fitted hyperparameter drift and sensitivity.** GLR, OM, RM, and WD denote global learning rate, output multiplier, residual multiplier, and weight decay. Each β is fitted from the optima of quadratic slices at $d8$ – $d10$, with one regression standard error; bold entries have $|\beta| \geq 3$ standard errors. For constant recipes, the optimal value scales as N^β . For +GLR recipes, the GLR column measures residual drift relative to the applied exponent shown in the recipe column; the other columns still measure drift relative to constant base values. Regret is the mean loss of the $0.5\times$ and $2\times$ settings minus the center loss; range is the full loss span of the slice. Both are in units of 10^{-3} at the largest fully swept size. The $d8$ anchor is shared across constant and scaled recipes. Growth variants are swept around their fixed-recurrence GLR rule. These diagnostic fits are distinct from the deployed settings in Table 9.

A.4.5 TRAIN THE LADDERS WITH THE FITTED RECIPE

Once the recipe is fixed, we train the scaling ladder. For a fixed-recurrence model, we count stored parameters N and set the token budget to $T = \text{TPP} \times N$. For a growth variant, the recommended recipe retains the Stage 2 allocation prescription and uses the fixed Stage 3 growth fraction, with tokens adjusted to preserve the prescribed compute budget across the two phases. We then apply the Stage 4 learning-rate rule and keep the other base hyperparameters unchanged. Table 9 lists the settings used for the reported ladders.

The ladders span $d6$ – $d20$ for Vanilla (120M–1.8B stored parameters) and $d6$ – $d18$ for the looped variants (120M–1.4B for K2 and 130M–1.8B for Dep; Table 2). Because the looped variants execute more blocks per token and train on more tokens per parameter, every ladder covers a similar range of compute, ending near 10^{20} FLOPs at its largest size despite the different stored-parameter counts. We compare at equal compute and fit Equation 1 to each ladder. First, we fit the irreducible loss E from the Vanilla ladder with Huber loss minimization (Hoffmann et al., 2022). For the remaining ladders, we regress log reducible loss against log compute and report the exponent’s regression standard error.

Architecture	GLR exponent β	Token allocation	Measured models
Vanilla	-0.8	TPP 5	$d6$ – $d20$, even
Deep Vanilla	-0.7	TPP 6	$d6$ – $d18$, even
Operator-1	-0.6	TPP 6	$d6$ – $d20$, even
Loop-2	-0.6	TPP 6	$d6$ – $d18$, even
Untied-2	-0.6	TPP 6	$d6$ – $d18$, even
Deep Vanilla Grow	-0.8	$t_{\text{ref}} = 6$	$d6$ – $d18$, even
Untied Grow	-0.6	$t_{\text{ref}} = 8$	$d6$ – $d18$, even
Loop Grow	-0.5	$t_{\text{ref}} = 7$	$d6$ – $d18$, even

Table 9: **Recipes used for the reported ladders.** Exponents apply in Equation 13 with $\text{GLR}_{d8} = 0.04$ for every family except Deep Vanilla Grow, whose fitted $d8$ anchor is 0.036. TPP is T/N over stored parameters; for the Grow variants, $t_{\text{ref}} = T_0/N_{\text{ref}}$ sets the pre-growth compute budget, with tokens recomputed after choosing the transition. Deep Vanilla Grow uses the Deep Vanilla TPP-6 reference budget and $\rho = 0.5$. The Loop Grow and Untied Grow experiments include refitted growth allocations; the sensitivity results motivate the simpler prescription of reusing the fixed-recurrence allocation and a constant ρ (Appendix A.4.3). All other hyperparameters keep their base-tuned values.

B ADDITIONAL RESULTS FOR COMPUTE-OPTIMAL SCALINGS

Figures 15 and 16 test how the scaling results depend on architecture, training recipe, model shape, and optimizer. Figure 15 examines the boundary operator, runtime efficiency, and constant-recipe baselines; Figure 16 separates the effects of tuning, depth, and recurrence. We compare the compute needed to reach the same loss, using interpolation within the measured ladders. Each ablation’s reference is specified in its caption, so multiplier magnitudes should not be compared directly across panels.

B.1 ARCHITECTURE CHOICES AND RUNTIME EFFICIENCY

The full boundary operator retains the largest gain. Figure 15a (left) compares Untied-2 with plain Deep Vanilla and three operator ablations. Near 10^{20} FLOPs, Untied-2 reaches roughly $1.34\times$ the compute efficiency of Vanilla, while using only normalization or injection, or omitting coda injection, reduces the multiplier to about $1.17\text{--}1.19\times$. Deep Vanilla remains near $1.08\times$. Thus, extra executed depth alone does not recover the full gain, and each of these operator choices contributes over the measured range.

Block allocation matters as models grow. Removing coda injection also reduces Loop-2’s gain (Figure 15a, middle). Fixing its prelude and coda at two and three blocks while scaling only the core is worse still: the multiplier falls below one at the largest budgets. Operator-1 shows a similar limitation with a fixed $1/C/2$ allocation: its early advantage peaks and then declines, while the proportional allocation continues improving (right). These comparisons support scaling the prelude, core, and coda together.

The gains carry over to training time. Figure 15b replaces FLOPs with recorded optimization time. The growth variants retain increasing time savings at matched loss, with Untied Grow giving the largest gain. The benefit therefore survives implementation costs on our hardware.

B.2 RECIPE TUNING CHANGES THE APPARENT SCALING ADVANTAGE

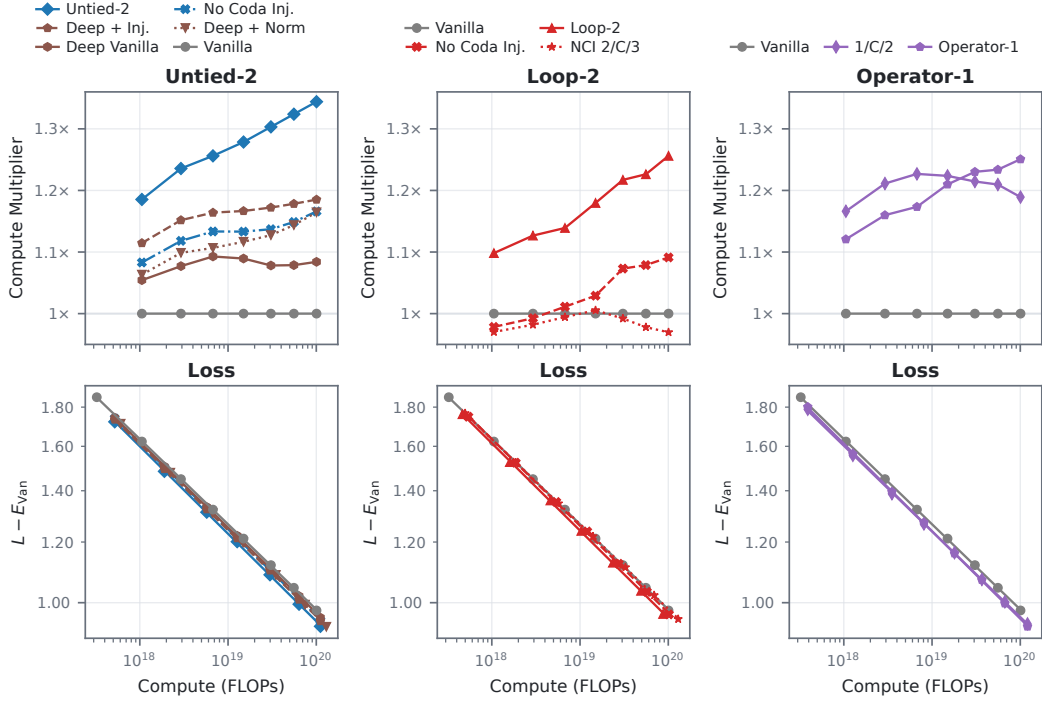
Constant base recipes favor the looped and grown models over Vanilla (Figure 15c). This is consistent with the ordering of β magnitudes in Table 8. Vanilla has the most negative β , and Loop-2 has the least negative values of β . As a result, compute multipliers under constant recipes are better than those under scaled ones, and in the long run, Loop-Grow is the best constant recipe. This emphasizes the importance of hyperparameter optimality in scaling (Qiu et al., 2026; Mlodozieniec et al., 2026). Figure 16a reinforces this theme: scaling Vanilla’s learning rate substantially improves its ladder. Adding output-multiplier or weight-decay scaling changes the multiplier by at most about 3% relative to GLR scaling alone, whereas the tested muP output-multiplier rule without GLR scaling loses efficiency with scale. This supports the GLR-only scaling prescription in Appendix A.4.4.

Tuning must also be architecture-specific. Simply ablating Operator-1’s recipe with Vanilla’s base tuned hyperparameters causes the exponent improvements to become constant ones (Figure 16b). Transferring a baseline recipe can therefore hide an architecture’s scaling improvement even when it remains better at individual budgets.

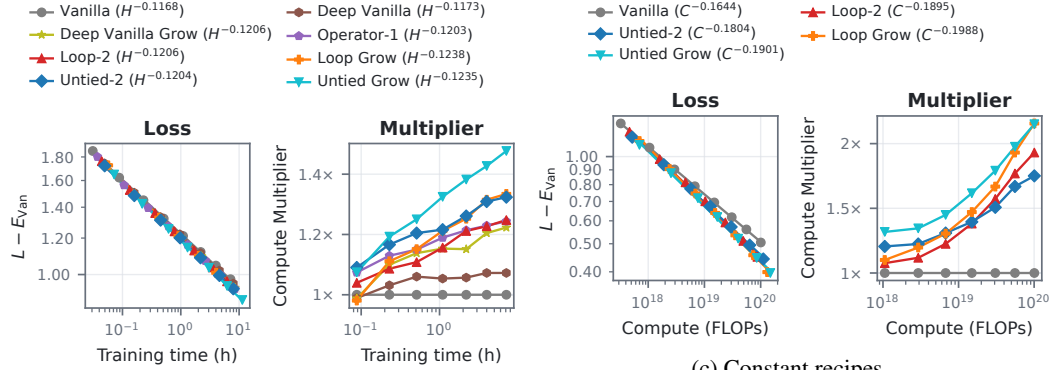
B.3 MODEL SHAPE AND THE DIRECTION OF SCALING

Making a model deeper at a given width does not consistently improve its compute efficiency (Figure 16c). Deep Vanilla gives a modest gain over Vanilla, but the still-deeper 1:64 depth-to-width ladders (Deeper series) do not consistently outperform their corresponding default shapes. In particular, Deeper Operator-1 loses the increasing advantage of Operator-1 over the measured range.

Holding executed depth fixed provides a more direct control (Figure 16e). At depth 11, width-only Untied-2 remains more efficient than Deep Vanilla, but Untied-2 now has a better constant, not exponent, than Deep Vanilla. This supports that boundary operators become more useful as depths increase and the hypothesis that compute-optimal scaling exponents improve because of increased computational depths..



(a) Architecture ablations



(b) Major variants by runtime

(c) Constant recipes

Figure 15: **Architecture, runtime, and constant-recipe ladders.** Points are measurements; loss curves fit power laws above a shared Vanilla-fitted floor. **(a)** Columns compare Untied-2 boundary-operator ablations, Loop-2 coda and allocation ablations, and Operator-1 block allocations. Rows show compute multipliers and reducible loss. NCI denotes no coda injection; 2/C/3 and 1/C/2 fix the prelude and coda depths while scaling the core. **(b)** Major variants against recorded optimization time, excluding evaluation and checkpoint overhead; the multiplier is Vanilla time divided by each variant’s time at matched loss. **(c)** Ladders with constant base recipes; the multiplier uses constant-recipe Vanilla as the reference. Multipliers interpolate between measurements in log compute or log time without extrapolation.

B.4 RANDOM RECURRENCE AND TEST-TIME PASSES

We match the Loop Grow recipe but sample K uniformly from $\{2, 3, 4, 5, 6\}$ at each optimizer step after growth, preserving mean recurrence four. Compute accounting uses the realized recurrence counts, and evaluation uses $K = 4$. This random-recurrence ladder is slightly worse than fixed Loop Grow at matched compute (Figure 16d).

Random training does improve tolerance to extra test-time passes (Figure 17). At $k = 8$, loss relative to $k = 4$ changes by -0.0003 to $+0.0028$, compared with increases of 0.008 – 0.042 for fixed Loop Grow. Five of seven random-recurrence checkpoints have a shallow minimum at $k = 5$, but the improvement is below 0.001 loss and does not continue with further passes. In this setting, random recurrence chiefly reduces the penalty for extra passes rather than providing sustained test-time scaling.

B.5 OPTIMIZER CHOICE AFFECTS BOTH THE CONSTANT AND THE EXPONENT

Muon has a constant improvement over Adam under width-only scaling, consistent with Qiu et al. (2026), but in coupled width/depth scaling, we find that compute efficiency gains *shrink* with larger scales (Figure 16f). This suggests that Muon optimizer may not be as effective as Adam for deeper networks. This discrepancy between width and joint scaling motivates investigation on the interaction of architectures and optimizers.

C CORPUS TRANSFER AND DOWNSTREAM EVALUATION

We test whether the compute-optimal gains carry over to a different pretraining corpus and to downstream tasks. We first compare Vanilla and Untied-Grow on FineWeb and FineWeb-Edu, then define the evaluation protocol and examine the eight-architecture downstream ladders. Finally, we describe the taskwise calibration used to extrapolate CORE accuracy beyond the measured compute range.

C.1 TRANSFER FROM FINEWEB TO FINEWEB-EDU

We repeat the Vanilla and Untied-Grow ladders on FineWeb-Edu with the same architectures and training recipes. Figure 18a separates the architectural advantage within each corpus from the effect of changing the corpus. On both datasets, Untied-Grow’s loss-matched compute multiplier increases with scale. Its fitted loss exponent exceeds Vanilla’s by a similar amount: 0.1168 versus 0.1113 on FineWeb, and 0.1146 versus 0.1095 on FineWeb-Edu, using a separate Vanilla-fitted irreducible loss for each corpus. Downstream gains also persist, although accuracy-based multipliers fluctuate more than loss-based multipliers.

For Untied-Grow itself, FineWeb-Edu improves downstream performance at matched training compute (Figure 18b). The fitted CORE curves project that FineWeb requires about 1.5 – $1.6\times$ as much compute to reach the displayed GPT-3 reference scores. These are extrapolated corpus comparisons; the multipliers in Figure 18a instead interpolate between measured architectures within each corpus. Appendix C.4 specifies the forecasting procedure. The held-out $d26$ run is excluded from these fits.

C.2 DOWNSTREAM EVALUATION PROTOCOL

We follow the 22-task DCLM CORE benchmark (Li et al., 2024) as implemented in Karpathy (b), evaluating all 91,037 examples at each checkpoint. Full CORE accuracy and answer negative log-likelihood (NLL) are the primary metrics. We also report a secondary accuracy score on a fixed subset of 17 tasks.

Full CORE accuracy. For multiple-choice and shared-ending tasks, the model chooses the candidate with the lowest mean token loss. For language-modeling tasks, an answer is correct only if every reference token is predicted correctly. Let a_j be task j ’s raw accuracy and b_j its random-guess accuracy. We chance-center each task and average with equal task weight:

$$c_j = \frac{a_j - b_j}{1 - b_j}, \quad \text{CORE}_{22} = \frac{1}{22} \sum_{j=1}^{22} c_j. \quad (14)$$

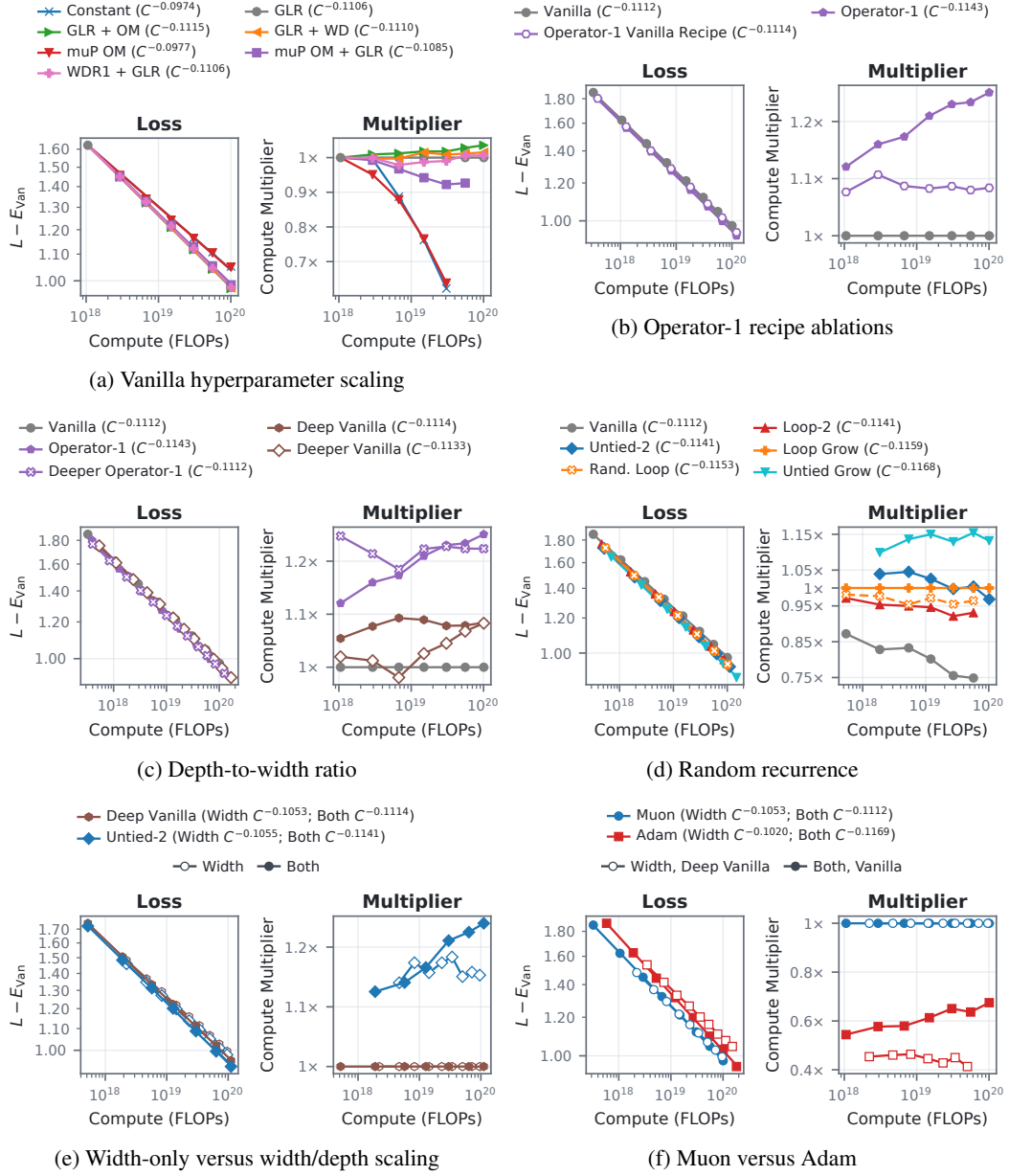


Figure 16: **Recipe, shape, recurrence, and optimizer ablations.** Each subfigure shows reducible loss and a loss-matched compute multiplier, interpolated without extrapolation. **(a)** Vanilla recipes share a fitted $d8$ anchor; multipliers are relative to constant + GLR. OM, WD, and WDR denote output multiplier, weight decay, and warmdown ratio. **(b)** Ablating Operator-1’s recipe by using Vanilla’s based tuned hyperparameters. **(c)** Shape comparisons; Deeper Vanilla and Deeper Operator-1 use depth-to-width ratio 1:64. Panels (b,c) use Vanilla as the reference. **(d)** Random recurrence samples $K \in \{2, 3, 4, 5, 6\}$ after growth and evaluates at $K = 4$; compute uses the realized recurrence counts, with fixed Loop Grow as the reference. **(e,f)** Open markers indicate width-only scaling at executed depth 11; filled markers indicate coupled width/depth scaling. Each regime uses its own reference: Deep Vanilla with Muon in (e), and the corresponding Muon ladder in (f). Each optimizer uses its fitted token budget.

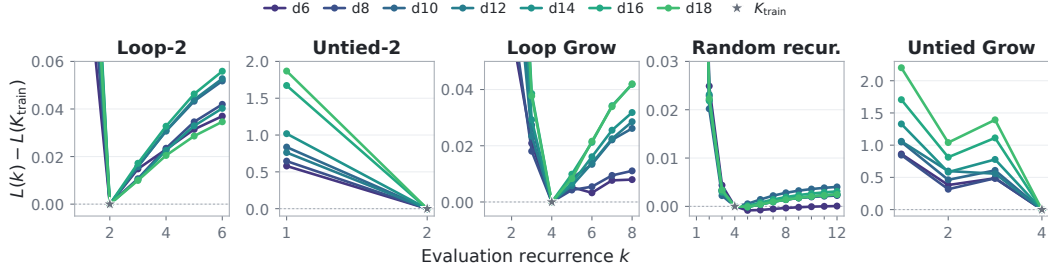
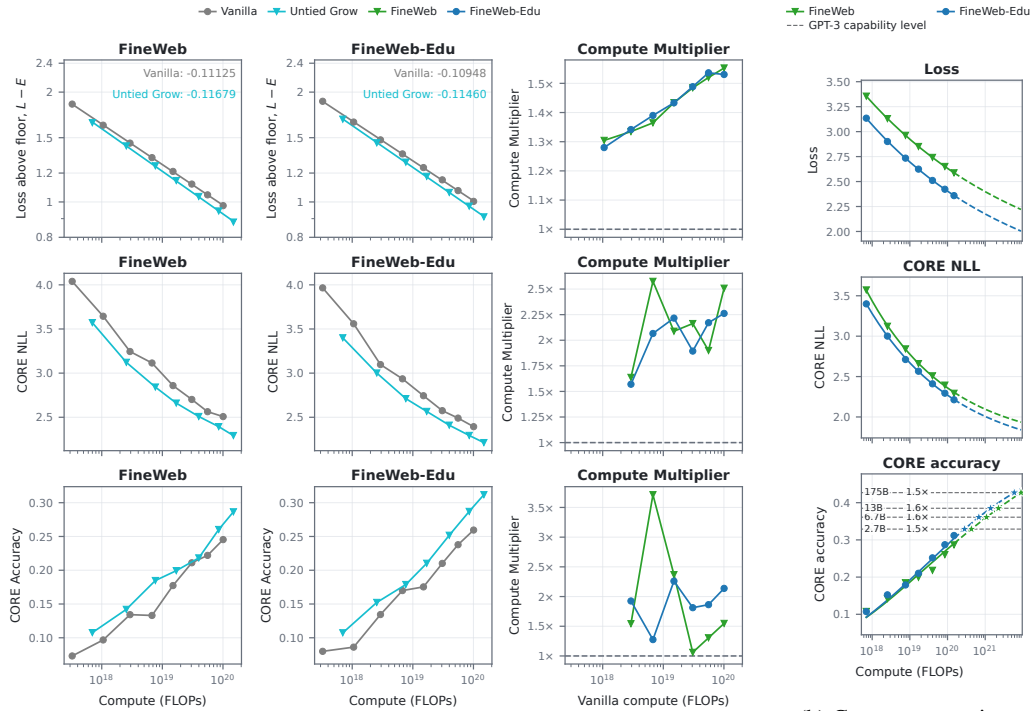


Figure 17: **Sensitivity to evaluation recurrence.** Loss differences $L(k) - L(K_{\text{train}})$ for the final-ladder checkpoints, with color indicating model size and stars marking the reference recurrence: $K_{\text{train}} = 2$ for Loop-2 and Untied-2, and 4 for the growth variants. Tied models reuse their core for additional passes; untied models can only run their allocated cores. Fixed Loop-2 and Loop Grow are best at their trained recurrence. Random-recurrence training produces much flatter curves, with small gains at $k = 5$ for five of seven checkpoints but no sustained improvement as more passes are added. Panels use different vertical scales.



(a) Architecture gains on FineWeb and FineWeb-Edu

(b) Corpus comparison

Figure 18: **Transfer from FineWeb to FineWeb-Edu.** (a) Vanilla and Untied-Grow on each corpus: rows show loss above the Vanilla-fitted floor, CORE NLL, and CORE accuracy. The final column shows Vanilla-to-Untied-Grow compute multipliers at matched metrics within each corpus, interpolated without extrapolation. (b) Untied-Grow corpus comparison: loss, CORE NLL, and CORE accuracy from top to bottom. Solid curves cover measured domains and dashed curves show extrapolation. Horizontal references mark GPT-3 CORE scores; multipliers compare FineWeb compute with FineWeb-Edu compute at matched accuracy. The held-out $d26$ run is excluded from both subfigures.

Zero denotes chance, one denotes perfect accuracy, and negative values denote below-chance performance. Table 10 lists the tasks and scoring types.

Task name	Evaluation type	Filter
HellaSwag (zero-shot)	MC	-
ARC-Easy	MC	-
ARC-Challenge	MC	-
COPA	MC	-
CommonsenseQA	MC	Kendall ($\tau = 0.000$)
PIQA	MC	-
OpenBookQA	MC	-
HellaSwag (10-shot)	MC	-
AGI Eval LSAT-AR	MC	-
BoolQ	MC	Below chance (-4.62 pp); Kendall ($\tau = 0.286$)
BIG-bench Language Identification	MC	Kendall ($\tau = 0.143$)
Winograd	SE	-
WinoGrande	SE	-
Jeopardy	LM	-
BIG-bench QA Wikidata	LM	-
LAMBADA OpenAI	LM	-
BIG-bench Dyck Languages	LM	-
BIG-bench CS Algorithms	LM	Kendall ($\tau = 0.500$)
BIG-bench Operators	LM	-
BIG-bench Repeat Copy Logic	LM	At chance ($+0.00$ pp); Kendall ($\tau = 0.423$)
SQuAD	LM	-
CoQA	LM	-

Table 10: DCLM CORE tasks and evaluation types. MC denotes multiple choice, SE shared ending, and LM language modeling. A dash in the Filter column denotes a retained task; other entries list the failed criteria: the accuracy difference from random chance in percentage points (pp) and/or Kendall’s τ_b across Vanilla checkpoints. Retention requires an accuracy gap of at least $+2$ pp and $\tau_b \geq 0.50$. Values are rounded; the unrounded τ_b for BIG-bench CS Algorithms is marginally below 0.50.

Answer NLL. Answer NLL measures the probability assigned to the correct reference answer, providing a continuous comparison even when two models select the same option (Grattafiori et al., 2024). For example i of task j , with prompt x_{ij} and answer tokens $y_{ij1:T_{ij}}$, define

$$\ell_{ij}^{\text{ans}} = -\frac{1}{T_{ij}} \sum_{t=1}^{T_{ij}} \log p(y_{ijt} \mid x_{ij}, y_{ij,<t}),$$

$$\text{NLL}_{22}^{\text{ans}} = \frac{1}{22} \sum_{j=1}^{22} \left(\frac{1}{n_j} \sum_{i=1}^{n_j} \ell_{ij}^{\text{ans}} \right). \quad (15)$$

Here n_j is the number of examples in task j . Prompt tokens are excluded; token averaging prevents longer answers from receiving larger losses merely because of their length. We average examples within each task and then average tasks equally. Lower values are better; the figures abbreviate this metric as CORE NLL.

Frozen Vanilla-only filter. Some tasks provide little signal at the scales studied, so we additionally report mean centered accuracy on a subset selected using only the eight final-GLR Vanilla checkpoints. Inspired by the evaluation-task selection criteria of Penedo et al. (2024), a task is retained if the largest-token checkpoint’s raw accuracy is at least two percentage points above chance and Kendall’s τ_b between training amount and accuracy is at least 0.50. This selects 17 tasks (Table 10). The subset stays fixed across architectures, recipes, and corpora. It is a post-hoc sensitivity analysis, reported alongside the two full-suite metrics; every evaluation still scores all 22 tasks.

Replicates and reproducibility. We use evaluation seeds 0, 1, and 2, which change the few-shot demonstrations rather than the benchmark examples. Deterministic zero-shot results may be reused across replicates. We average replicates within each task before averaging tasks.

C.3 DOWNSTREAM SCALING AND TASK-LEVEL RESIDUALS

Figure 19a evaluates 58 checkpoints across eight architectures. The looped and grown variants generally improve downstream compute efficiency, especially on answer NLL and filtered CORE accuracy. Full CORE accuracy is less smooth across checkpoints, and its matched-score compute estimates fluctuate accordingly. Each multiplier uses interpolation between measurements without extrapolation.

Much of the downstream improvement tracks pretraining loss, but architecture-dependent residuals remain (Figure 19b). A single linear fit pools all checkpoints with equal weight; Untied-Grow generally has lower CORE NLL than this fit predicts. Figure 19c uses a different reference to resolve the task contributions: a separate Vanilla-only NLL-versus-validation-loss line for each task. Bars average measured NLL minus that prediction for each architecture, over all checkpoints or the lower half of the pooled validation-loss range. Negative residuals indicate better answer prediction than the Vanilla trend. Differences are uneven across tasks, with prominent gains on Dyck Languages and several reading-comprehension tasks. These residuals depend on the linear reference, which is also evaluated beyond Vanilla’s measured loss range for some checkpoints.

C.4 FORECASTING DOWNSTREAM PERFORMANCE

To forecast the compute needed for a target CORE score (Grattafiori et al., 2024), we fit a taskwise pipeline: compute to answer NLL, answer NLL to centered accuracy, and task accuracy to the full CORE score. Neither validation loss nor the aggregate CORE NLL is an intermediate in this forecast.

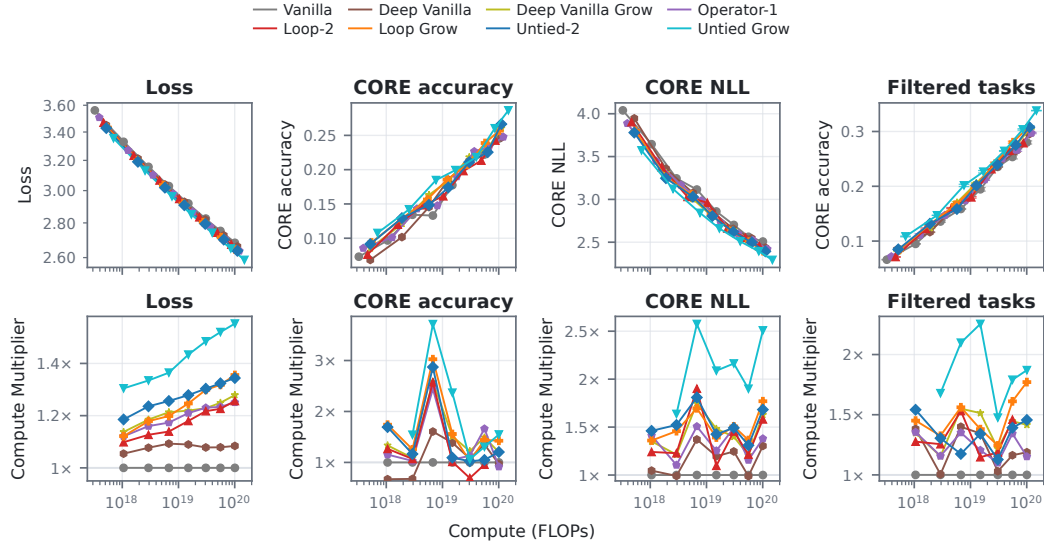
1. **Compute to task NLL.** For series a (an architecture or corpus) and each of the 17 selected tasks t , fit $B_{a,t}(C) = E_{a,t} + A_{a,t}(C/10^{18})^{-\alpha_{a,t}}$ with Huber loss. Each series has its own task-NLL scaling laws.
2. **Task NLL to accuracy.** Fit one sigmoid per task, $\hat{c}_t(B) = [1 + \exp(s_t B - b_t)]^{-1}$, shared across the series being compared. The target is the task’s centered accuracy.
3. **Aggregation.** Average the 17 predicted task scores and fit a shared, no-intercept coefficient r to recover full CORE accuracy: $\widehat{\text{CORE}}_{22,a}(C) = \frac{r}{17} \sum_t \hat{c}_t(B_{a,t}(C))$.

For the FineWeb-Edu architecture comparison, calibration pools eight Vanilla and seven Untied-Grow checkpoints, giving $r = 0.86912$. Figure 20 shows the task sigmoids and the filtered-to-full conversion. The corpus comparison fits a separate shared calibration to seven FineWeb and seven FineWeb-Edu Untied-Grow checkpoints, giving $r = 0.87104$. Only task selection is inherited from the FineWeb Vanilla filter; both calibrations use the measurements in their respective comparisons.

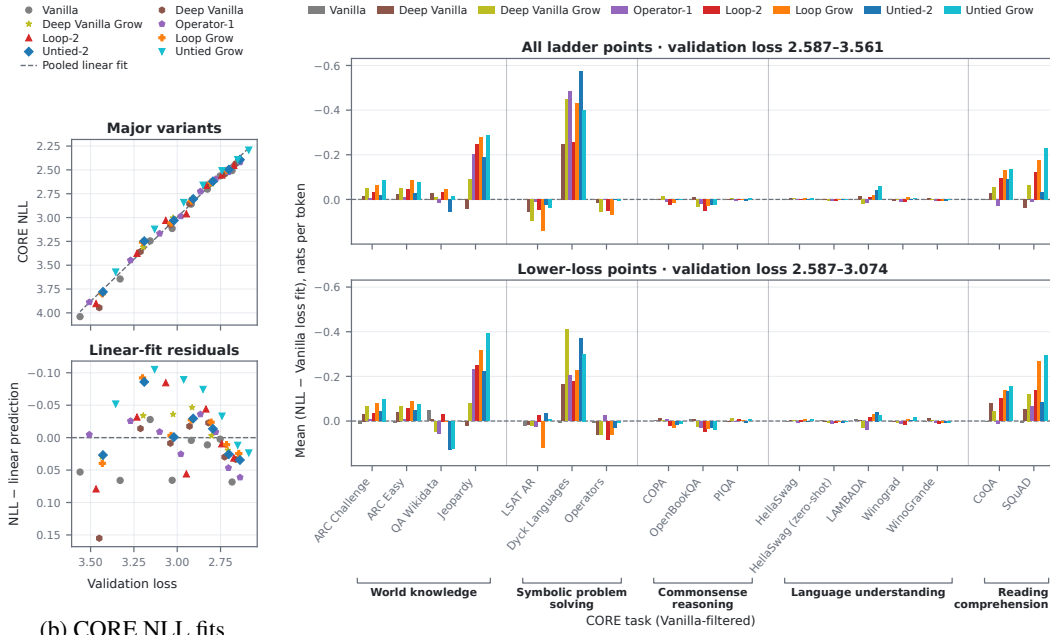
The held-out $d26$ model is excluded from every fit. At its compute of 1.225×10^{21} FLOPs, the architecture-comparison pipeline predicts CORE accuracy 0.3837, versus the observed 0.3865 ± 0.0015 . The error bar is one standard deviation across evaluation seeds, not uncertainty in the fitted extrapolation. Matching the GPT-3 reference scores (Karpathy, a) with these curves gives projected Vanilla-to-Untied-Grow compute ratios of about $2.5\text{--}3.5 \times$ (Figure 4); these projections assume that the small-scale relationships continue beyond the measured ladders.

C.5 SCOPE OF THE COMPARISONS

The ladders use a fixed batch size and common hardware, so their fitted recipes do not address joint optimization of batch size and model scale. The filtered score and task residuals supplement the full-suite metrics, while extrapolated CORE predictions additionally depend on the fitted task-NLL laws and calibration curves.



(a) Downstream scaling ladders



(b) CORE NLL fits

(c) Mean task residuals by architecture

Figure 19: **Downstream performance across architectures and tasks.** (a) Loss and downstream metrics for eight architectures; the lower row shows compute multipliers relative to Vanilla, interpolated between measurements without extrapolation. (b) CORE NLL versus validation loss (top) and residuals from one linear fit pooled across all architectures (bottom), with both axes reversed. (c) Mean task NLL residuals relative to a Vanilla-only linear fit against validation loss, grouped by CORE category, for all ladder points (top) and the lower-loss subset (bottom). Negative residuals indicate lower NLL and point upward.

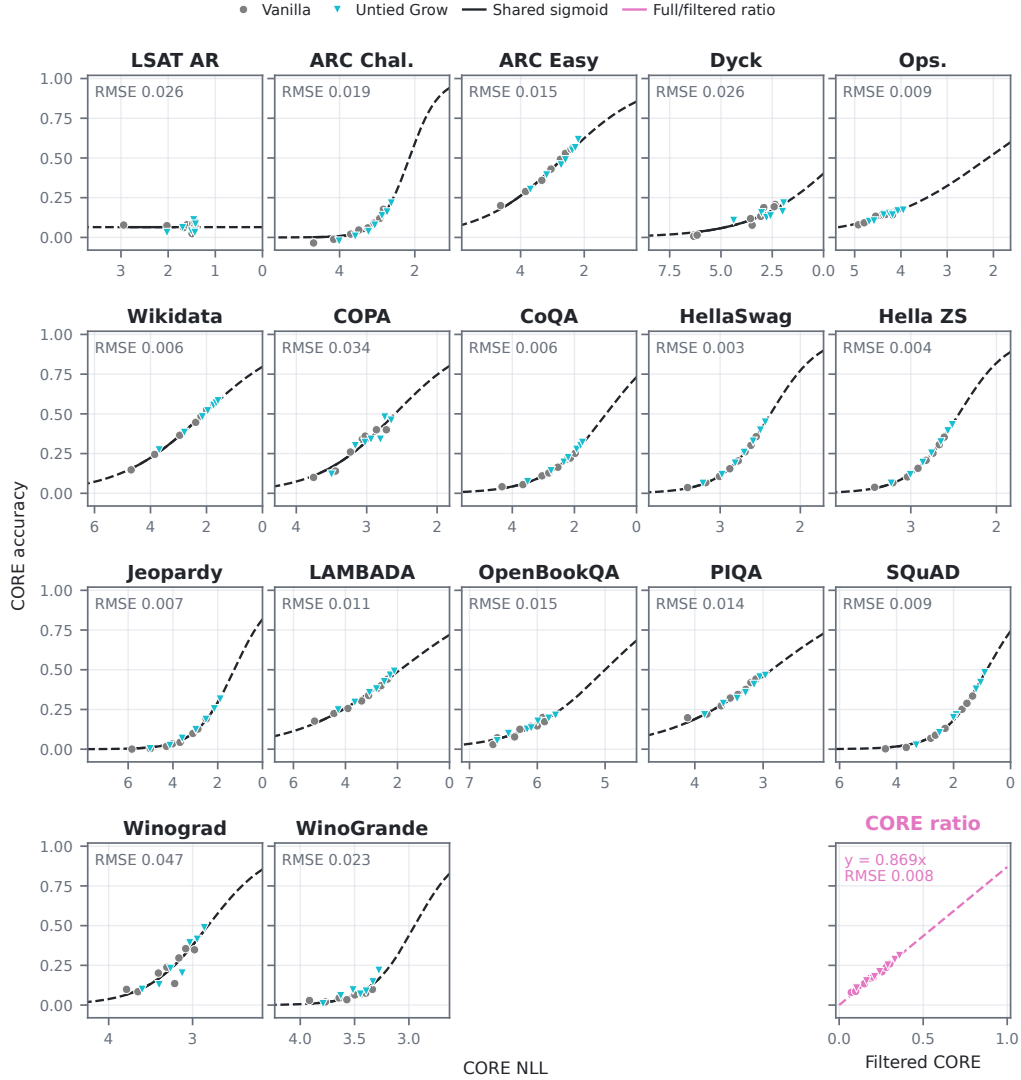


Figure 20: FineWeb-Edu calibration for the architecture comparison. The first 17 panels show centered task accuracy against task CORE NLL for the Vanilla-selected tasks. Each solid segment is the shared Huber sigmoid over all FineWeb-Edu Vanilla and Untied-Grow measurements; dashed segments extend the fitted curve beyond the measured NLL range. The final purple panel shows the no-intercept conversion from the 17-task filtered score to full CORE across the same 15 measurements.

D ADDITIONAL RESULTS FOR DATA-CONSTRAINED SCALING

We extend Section 5 across repetition levels, weight decay, and architecture controls. At fixed token exposure, we compare spending additional compute on stored model size or extra core passes.

D.1 DATA REPETITION CHANGES THE PREFERRED RECURRENCE

Figure 21 compares approximately 1B token exposures from fresh data, a 250M-token pool repeated four times, and a 100M-token pool repeated ten times. The first three columns use weight decay 0.8 and otherwise fixed base hyperparameters. Under fresh data and four epochs, the preferred recurrence remains near one to two passes across the measured budgets. With ten epochs, increasing model size at low recurrence eventually worsens loss, while higher recurrence postpones this upturn; the best recurrence therefore rises more strongly with compute.

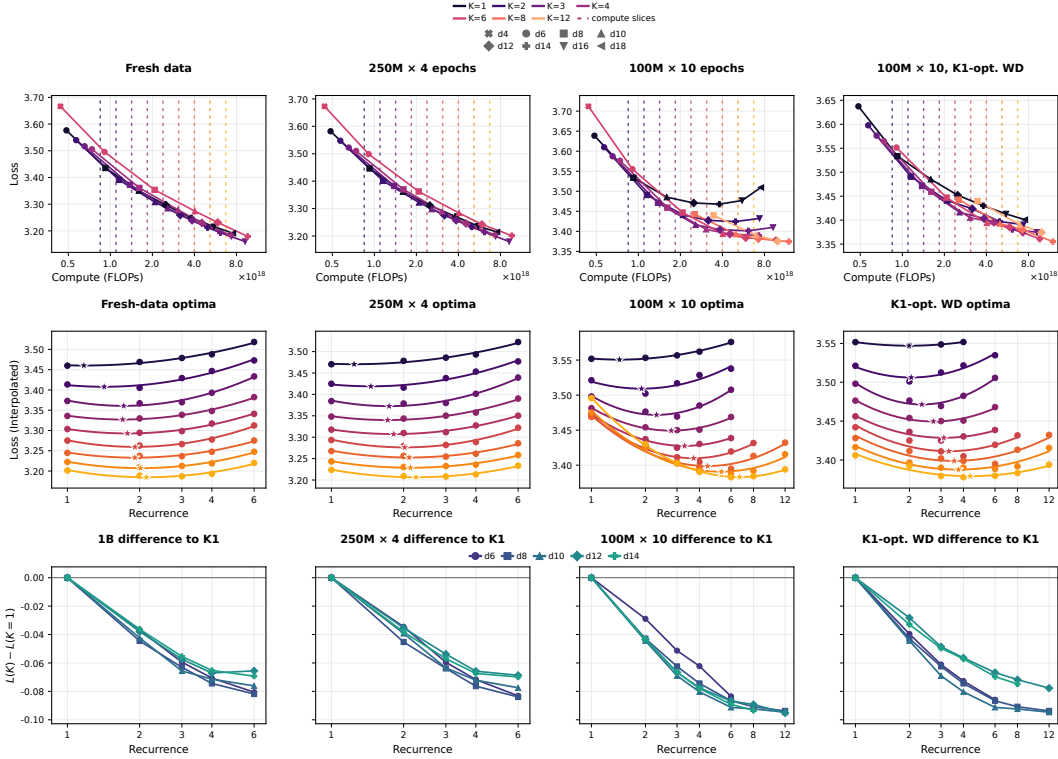


Figure 21: **Recurrence across data-repetition regimes.** Columns compare fresh data, 250M tokens repeated four times, and 100M tokens repeated ten times at fixed weight decay 0.8, followed by the ten-epoch regime with each size’s $K = 1$ -selected weight decay. Rows show loss versus compute, matched-compute recurrence fits, and loss differences from $K = 1$ at fixed stored depth. Colors identify recurrence in the top row, compute slices in the middle row, and depth in the bottom row. Filled stars mark interior fitted optima; hollow stars mark the best measured recurrence when the fit is censored. The ten-epoch columns include $K = 8, 12$ only where their measured compute ranges bracket the comparison budget.

The fourth column repeats the ten-epoch comparison with weight decay selected at $K = 1$ for each model size and then held fixed across recurrence. This reduces overfitting and the shift toward larger K , but does not remove the shift. The bottom row measures $L(K) - L(1)$ at fixed stored depth: extra passes continue to reduce loss even when increasing stored size becomes less useful. These fixed-depth comparisons spend more compute as K grows; the middle row makes the equal-compute comparison.

At each compute slice, we interpolate within measured ranges and fit a quadratic in log recurrence and log loss. Filled stars mark interior minima; hollow stars mark the best measured recurrence when

no interior optimum is identified. The ten-epoch sweeps include $K = 8, 12$ where measurements bracket the budget; the other regimes end at $K = 6$.

D.2 WEIGHT DECAY AND RECURRENCE ARE COMPLEMENTARY

Figure 22 crosses the ten-epoch protocol with weight decay in $\{0.05, 0.2, 0.4, 0.8, 1.2, 1.6\}$. Weak regularization produces the strongest upturn in loss as stored size grows and favors high recurrence at large budgets. Stronger weight decay reduces that upturn and moderates the preferred recurrence. Extra passes still improve loss at fixed stored depth, although the gains are smaller. Thus, recurrence gains are largest when regularization is insufficient, but are not eliminated by tuning weight decay.

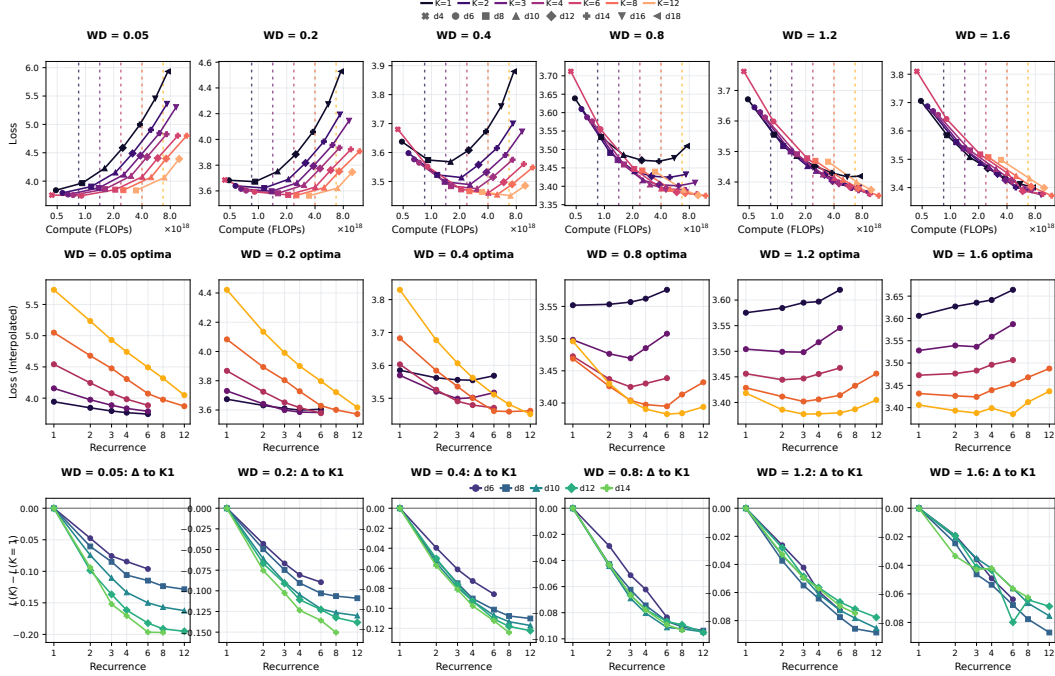


Figure 22: **Weight decay moderates the preference for recurrence.** Each column uses one of six weight-decay values under the 100M-token, ten-epoch protocol. Rows show loss versus compute, matched-compute recurrence comparisons, and $L(K) - L(1)$ at fixed stored depth. Low weight decay produces stronger overfitting with model size and a greater preference for additional passes. Compute-slice colors match between the first two rows; the bottom row uses depth colors. Comparisons interpolate within measured ranges, including $K = 8, 12$ where available. Panels use independent vertical scales.

D.3 HYPERPARAMETER TRANSFER ACROSS RECURRENCE

The preferred weight decay changes more with stored depth than with tied recurrence (Figure 23a). On the shared grid, selecting weight decay at $K = 1$ gives 0.4 at $d6$, 0.8 at $d8$ – $d10$, 1.2 at $d12$ – $d14$, and 1.6 at $d16$ – $d18$. Reusing these values across K yields the fourth column of Figure 21.

A separate fresh-data sweep at $d8$ tests transfer of the global learning rate, weight decay, output multiplier, residual multiplier, and injection scale across $K = 2, 4, 8$ (Figure 23b). Tied models retain broadly similar optima, while untied models shift more, particularly toward smaller learning-rate and residual multipliers at higher recurrence. Tokens are fixed within each family at approximately 1.235B for tied models and 1.466B for untied models. Compute increases with recurrence; for untied models, tokens per stored parameter also decrease.

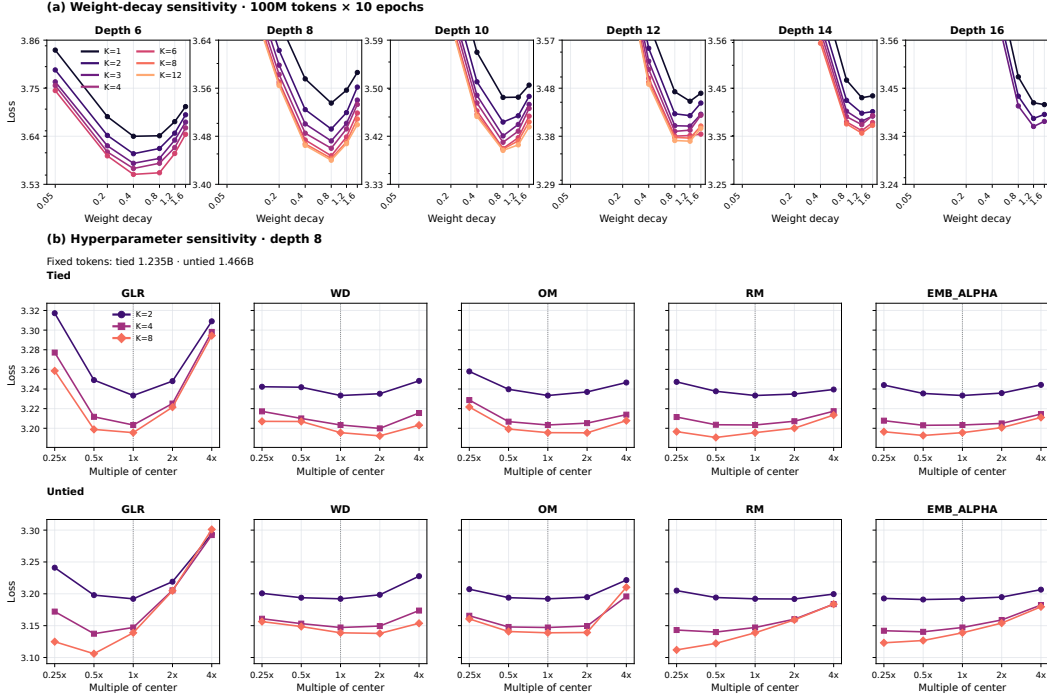


Figure 23: **Hyperparameter sensitivity to stored depth and recurrence.** (a) Weight-decay sweeps under the 100M-token, ten-epoch protocol, with one panel per stored depth. The preferred weight decay shifts with depth but varies relatively little across tied recurrence. (b) Fresh-data sweeps at $d8$ with two, four, and eight passes, using fixed token counts within each family: 1.235B for tied models and 1.466B for untied models. Columns vary GLR, weight decay, output multiplier, residual multiplier, and injection scale by $\{1/4, 1/2, 1, 2, 4\}$ times their base values. Dotted lines mark the base recipe. Untied recurrence causes larger shifts in several optima.

D.4 ARCHITECTURE CONTROLS: SHARING WEIGHTS AND ADDING DEPTH

Figure 24 compares three controls under the 100M-token, ten-epoch protocol at weight decay 0.8. Tied Vanilla repeats plain Transformer cores with shared weights. Untied Vanilla uses independent copies and is equivalent to a deeper plain Transformer. Both use the frozen Vanilla recipe without the boundary operator, share the $K = 1$ ladder, and match FLOPs at each depth and recurrence. The third control varies the number of untied cores with the boundary operator.

The preferred recurrence generally increases with budget in all three families, so repetition can favor depth even without weight sharing. Both plain-Transformer controls remain above the tied boundary-operator frontier. The untied operator model is slightly better at the two smallest reference budgets but worse at the three larger ones: tied looping’s advantage emerges as adding depth through new parameters becomes less effective.

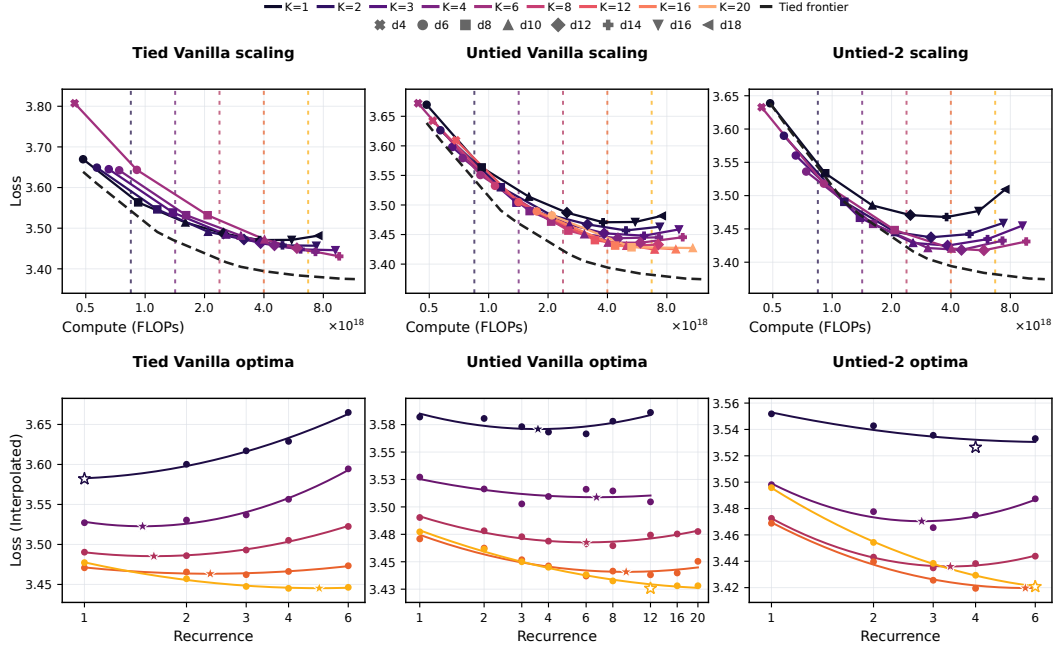


Figure 24: **Architecture controls under data repetition.** Columns show Tied Vanilla, Untied Vanilla, and the untied family with the boundary operator, all at weight decay 0.8 on 100M tokens for ten epochs. Top: loss versus compute, with the tied boundary-operator frontier repeated as a dashed reference. Bottom: matched-compute recurrence comparisons. Filled stars mark interior fitted optima; hollow stars mark censored fits at the best measured recurrence. Both plain-Transformer controls remain above the reference frontier. The untied operator family is slightly better at the two smallest reference budgets but worse at the three larger budgets.