POSITION: TRANSFORMER TRAINING GRAPHS SHOULD BE FIRST-CLASS DESIGN VARIABLES

Anonymous authors
Paper under double-blind review

ABSTRACT

Transformer depth can be changed through skipping, looping, reordering, and parallel execution, but studies differ in which graph they train, which pretrained weights they edit, and what resources they allow. An operation name therefore does not identify a comparable experiment. We argue that the complete *training graph*—its block calls, dependencies, parameter-sharing identities, merge rules, execution order, path-selection rule, and settings attached to individual block calls—should be treated as a shared experimental object. Claims about this object require graph-aware comparisons that separate whether training on the graph learns the intended computation, what quality its cost buys, how fixed weights respond to an execution edit, and what a multi-operation graph adds over simpler alternatives. In four shared runs of a controlled strict-order task, Serial has lower test error than Parallel-concat in all four but lower error than Looping in only one; the Looping comparison is not matched on estimated arithmetic. In language modeling, a wider shared-looping model has lower validation loss than a narrower serial model at similar parameter counts and matched training tokens and block calls, but width and sharing remain confounded and the looping design takes about 1.56 times as long to train. In the tested checkpoints, fixed-weight loop-count edits increase loss, while training with skipping lowers later skipping damage but misses the full-execution quality criterion. These experiments address task fit, quality and cost, and execution dependence, but none trains a multi-operation graph, so its value beyond simpler graphs remains untested. Together, these cases support a comparison contract that can select a proposed graph or retain a serial or simpler alternative.

1 INTRODUCTION

Transformers are usually trained as a fixed serial sequence of blocks, but the four operation families considered here can change their depth computation. *Skipping* omits block calls, *looping* repeats them, *reordering* changes their order, and *parallel execution* sends the same incoming block state through multiple paths before merging their outputs. Established examples include Universal Transformers for looping (Dehghani et al., 2019), LayerDrop for sampled skipping during training (Fan et al., 2020), and fixed-weight tests of reordering and parallel execution with pretrained weights held fixed (Sun et al., 2024). Together, these lines leave a common comparison problem: how should a model trained under one depth computation be compared with serial and structurally simpler alternatives under stated resource limits?

Architecture and execution are already design choices; Program-of-Layers, for example, combines skipping and looping when it selects input-dependent programs over frozen pretrained layers (Li et al., 2026). **We argue that researchers should specify the complete Transformer training graph and weight lifecycle for each graph claim, then compare its claimed benefit with serial and simpler alternatives under a declared resource allowance.** This places the decision-relevant comparison, rather than the operation name, at the center of the claim.

We use *depth graph* for the block calls, dependencies, parameter-sharing identities, and merge rules, and *training graph* for that structure together with its execution order, path-selection rule, and any settings attached to individual block calls, such as identifiers, normalization, or losses. A *graph hypothesis* states why a proposed training graph should help, such as repeated refinement,

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

independent subcomputations, or usable shorter paths. A *resource objective* states which capability or resource profile motivates the candidate, while a *resource allowance* states the constraint under which alternatives are compared.

The corresponding experiment compares the trained model with models trained on the serial graph, graphs using each component operation alone, and simpler alternatives under that allowance.

Figure 1 separates two decisions: compose depth operations into a candidate structure, then specify how that structure is trained.

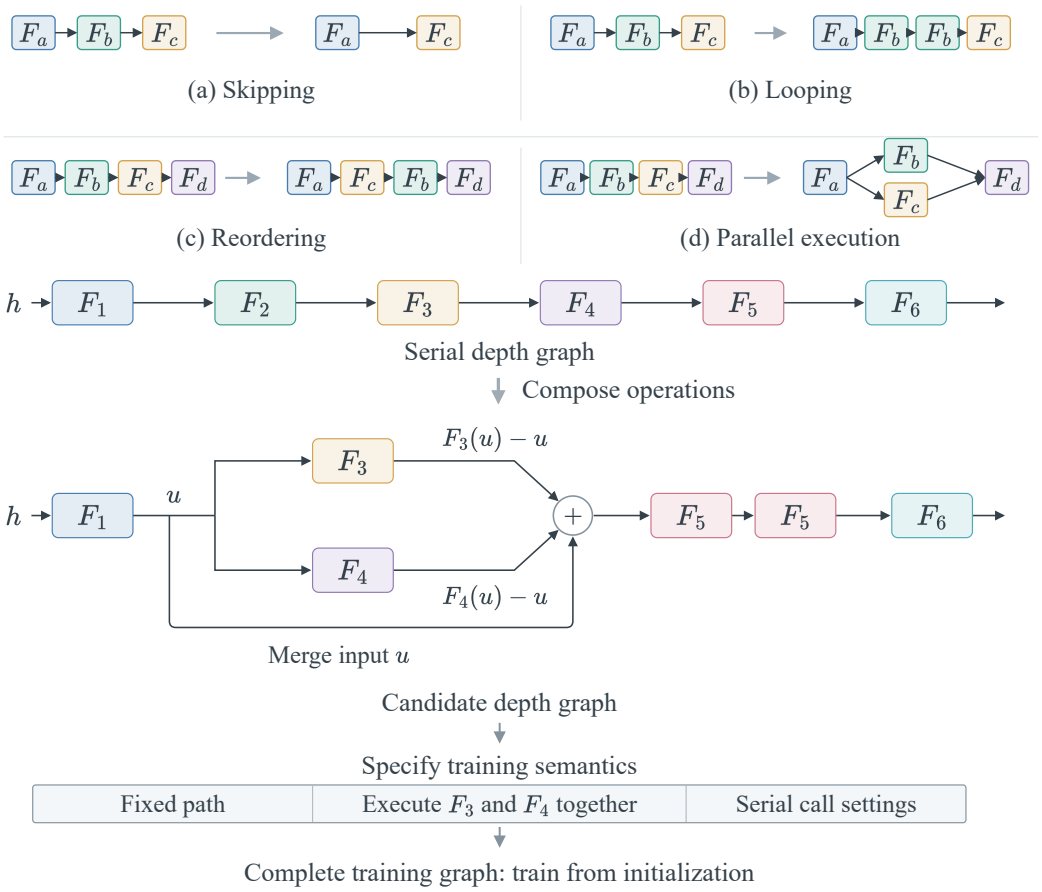

Figure 1: **From depth operations to a complete training graph.** The upper panels compare designs before training (gray arrows); dark arrows carry states. Panel (a) fixes an omitted call, unlike the sampled LayerDrop paths in Section 5.5; panel (d) leaves its merge rule open. For trainable blocks F_i and input state h, the middle candidate combines skipping, parallel execution, and looping, not reordering: with $u = F_1(h)$, it omits F_2, adds the F_3 and F_4 residual updates to u, calls F_5 twice with shared weights, then calls F_6. The lower row sets a fixed path, concurrent branches, and serial call settings for training from initialization. Colors distinguish block positions within each design, not transferred weights. We do not train this multi-operation example.

Our comparison contract names the training graph, alternatives, and allowance, then asks:

- *Task fit*: does training on this graph learn the intended computation?
- *Quality and cost*: what prediction quality does the declared training or deployment budget buy?
- *Execution dependence*: how does loss change when fixed weights run an execution different from the one used during training?
- *Composition value*: does a multi-operation graph add value beyond its constituents and simpler alternatives?

These questions are not interchangeable: changing a graph can also change parameter sharing, width, and executed work, while quality under the trained execution does not establish that an edited execution will work. The experiments below illustrate the first three questions; composition remains a proposed comparison rather than an observed gain.

We contribute a two-level specification separating block-call structure from the execution under which weights are optimized, and a comparison contract that makes weight lifecycle, alternatives, resource allowance, and four distinct questions explicit. On the same controlled runs, changing the alternative changes Serial's paired test-error conclusion; a wider looping language model has lower validation loss but longer training with width unmatched; and lower skip damage can coexist with failed full-execution quality. These cases show why graph quality, cost, and edit tolerance must be judged separately when choosing a graph.

2 Why Operation Names Do Not Define Comparable Training Graphs

The same depth operation can describe different experimental objects: weights trained from initialization on that graph, fixed pretrained weights run on an edited path, or a subnetwork inherited from joint training. These objects support different claims. We first compare executable multi-operation precedents with our training question, then locate what each operation and graph-search line already establishes.

2.1 Multi-Operation Programs and Weight Lifecycles

Program-of-Layers selects input-dependent programs that can skip or repeat frozen pretrained layers (Li et al., 2026). It establishes that skipping and looping can coexist in one executable program and that their placement may depend on the input. Transformer Layers as Painters also executes parallelized middle layers repeatedly on pretrained models and reports better performance than a single parallel pass in its tested variants (Sun et al., 2024). These studies establish that operations can be composed over existing block weights. They do not compare a multi-operation graph trained from initialization with serial, constituent, and simpler independently trained alternatives under a declared resource allowance.

The lifecycle distinction extends to fixed-weight diagnostics and inherited subnetworks. Editing one checkpoint measures its response to a changed execution; it does not measure how weights trained under that new execution would perform. A subnetwork inherited from weights jointly trained across candidate paths must likewise be judged as an inherited candidate. These distinctions matter before an operation is used to motivate a new training graph.

2.2 What the Four Operations Already Establish

Skipping can be sampled during training rather than fixed as one omitted block. Stochastic depth randomly bypasses residual blocks in convolutional networks during training (Huang et al., 2016); LayerDrop randomly omits Transformer layers during training and later selects shallower subnetworks (Fan et al., 2020); LayerSkip and Mixture-of-Depths further change prediction heads or token-level computation (Elhoushi et al., 2024; Raposo et al., 2024). These lines establish different ways to train or select shorter paths; they do not identify the value of a particular fixed-skipping training graph against a simpler independently trained alternative under one declared resource allowance.

Looping and cross-layer sharing make a different trade-off: they reuse transformations across depth. Universal Transformers, ALBERT, and Looped Transformers provide trained examples (Dehghani et al., 2019; Lan et al., 2020; Yang et al., 2023); Adaptive Computation Time learns when to stop repeating (Graves, 2016). Later recurrent-depth studies compare different parameter or computation budgets (Saunshi et al., 2025; Geiping et al., 2025; Wang et al., 2026). Sharing can reduce stored parameters while leaving executed work unchanged or increasing it. Parameter matching therefore cannot stand in for arithmetic or wall-clock matching.

Reordering changes which block receives another's state; fixed-weight swaps and reversals measure loss under altered dependencies (Sun et al., 2024). Parallel execution sends a shared state through

multiple branches and merges their outputs; parallel middle-layer averaging has also been tested as a fixed-weight edit (Sun et al., 2024). Such edits do not establish the quality of weights trained on the changed order or merge. Fewer sequential dependencies do not guarantee a speedup without a compatible workload and implementation. Across all four operations, the execution and comparison must be stated more precisely than the operation name.

2.3 Search, Inheritance, and the Remaining Comparison

Neural architecture search already treats network structure as a candidate space and distinguishes search space, strategy, and performance estimation (Elsken et al., 2019). Weight-sharing supernets can reduce search cost; Once-for-All trains a common network and specializes inherited subnetworks across structural and deployment constraints (Cai et al., 2020). These results show how candidates can be found and inherited, not that an inherited candidate equals an independently trained instance of the same graph.

Our question begins once a Transformer training graph and claim are named: which complete execution was optimized, which lifecycle produced its weights, which alternatives could overturn the preference, and which resource allowance governs the comparison? The next section makes the training graph itself recoverable before the paper defines how to test it.

3 Training Graphs as a Shared Experimental Object

An operation name is not enough to identify the model being trained. A *depth graph* records block calls, their dependencies, parameter-sharing identities, and merge rules. A *training graph* additionally specifies execution order, how a path is selected, and any identifier, normalization, or loss attached to an individual block call. The first object describes the computation's structure; the second identifies the complete computation under which weights are optimized. This section makes that distinction operational before asking whether a candidate graph is worth training.

3.1 Calls, Dependencies, and Execution Rules

In a serial Transformer, each block consumes the preceding block's output and has its own parameters. Calling a sequence of 32 shared blocks twice makes 64 block calls but uses only 32 distinct block-parameter sets. Call count and stored parameters therefore measure different aspects of the same execution. In a depth graph, each node is a block call, each directed edge states which output a later call consumes, and a parameter label identifies calls that share weights. Starting from a serial reference, *skipping* omits calls, *looping* repeats them, *reordering* changes their order, and *parallel execution* gives multiple calls a shared input and a specified merge. These four families organize the candidates considered here; they are not an exhaustive grammar for every possible Transformer computation.

The depth graph also states how branches merge. To turn that structure into a training graph, the researcher must specify how it is executed. A path-selection rule may be fixed, sampled during training, or input dependent; it is part of the specification, not a fifth operation. Repeated calls may need iteration identifiers, implemented as a learned iteration embedding in the looping model below, or boundary normalization. These choices can change the function and supervision even when two diagrams have the same connectivity. Their values must be stated before a training or evaluation result can be attributed to the complete graph rather than to an operation label.

3.2 From a Candidate to a Testable Comparison

The candidate derived from a six-block serial reference in Figure 1 makes six block calls using five distinct parameter sets: it omits F_2 and shares F_5 across two calls. Appendix A specifies its merge and remaining execution choices. This is a design illustration, not a trained multi-operation graph; the structural counts do not establish quality, measured speed, or an advantage over a single-operation alternative.

A candidate is worth testing when a graph hypothesis predicts a relevant capability or resource benefit and when a feasible alternative could change the decision. Repeated refinement may motivate looping;

independent subcomputations may motivate parallel execution; usable shorter paths may motivate training with skipping. Combining operations requires a joint hypothesis, not merely the ability to draw them together. Once the complete training execution is specified, the remaining question is what it buys relative to serial, constituent, and simpler graphs under a declared allowance. The next section defines those comparisons.

4 Principles for Evaluating Training Graphs

Specifying a complete training graph makes a claim testable, not true. A comparison must ask whether the trained graph learns the target computation, what quality its resource cost buys, whether fixed weights tolerate a specified execution edit, and whether combining operations adds value beyond simpler graphs. These questions require different measurements and controls; a gain on one axis does not answer the others.

4.1 Task Fit, Quality and Cost, and Execution Dependence

A graph hypothesis names a target computation and an alternative that might perform it at least as well. A controlled task can test that contrast after both complete graphs are trained under common data, optimization, and a declared allowance. Its task label alone cannot determine which graph wins. The comparison should report absolute quality under the execution of interest: test mean-squared error for the constructed tasks below, or held-out token-mean negative log-likelihood for the language-model studies.

Quality under a trained execution differs from sensitivity to an edit of that execution. For a fixed checkpoint θ, evaluation examples $\mathcal{D}$, declared reference execution $\mathcal{E}$, and edited execution $\mathcal{E}'$, let $\mathcal{L}(\theta, \mathcal{E}; \mathcal{D})$ be loss averaged over the stated evaluation unit. The damage from this particular edit is

$$\Delta(\theta; \mathcal{E} \to \mathcal{E}', \mathcal{D}) = \mathcal{L}(\theta, \mathcal{E}'; \mathcal{D}) - \mathcal{L}(\theta, \mathcal{E}; \mathcal{D}).$$

The two losses use the same checkpoint and examples; positive damage means the edit worsens prediction. Small damage does not establish high absolute quality, and damage under one edit does not establish tolerance to arbitrary paths. The fixed-weight diagnostics below use predicted tokens as their averaging unit. A model intended to run on several paths must also report its absolute quality on those paths, not only damage from its own reference.

Resource claims need their own measurements. Distinct parameter sets, block calls, arithmetic, sequential dependencies, wall-clock time, and peak memory are different quantities. Equal parameters or calls do not imply equal arithmetic or time; any speed claim must identify the implementation, hardware, precision, sequence length, and measured training or inference phase. Search and failed training runs belong in the cost of choosing a graph when the claim concerns net resource value.

4.2 Composition and a Decision Rule

To say what composition adds, compare a trained multi-operation graph with the serial reference, its constituent graphs, and simpler mixtures under the same declared decision rule. A quality-under-cost claim, for example, requires quality and measured cost for the best evaluated feasible alternative in that candidate set. Fewer parameters, calls, or dependencies by construction cannot substitute for a measured quality or runtime result. The finite comparison can change the choice within its candidate set; composition value requires training the proposed graph and its constituents under that same decision rule.

In practice, state the graph hypothesis, resource objective, feasible alternatives, and acceptance rule before final testing. Use validation or calibration data to choose checkpoints and paths, keep disjoint held-out examples for final quality and edit diagnostics, and report the resource named by the claim alongside any relevant search cost. This workflow may select the proposed graph, a constituent, the serial reference, or no winner when a required measurement is missing. The following experiments show why these outcomes must remain distinct.

5 Empirical Tests of the Comparison Contract

The experiments test three questions from the contract above. Controlled tasks ask whether a graph helps learn a specified computation relative to a named alternative. A paired language-model study compares final quality with measured training cost. Fixed-weight loop edits and a separate skipping-training study then distinguish damage under a changed execution from absolute quality. These are studies of constituent operations and complete models, not tests of a trained multi-operation composition. No experiment in this paper trains a reordered graph.

5.1 Experimental Setup

The controlled tasks use fixed teacher functions on 16-dimensional inputs: one repeats a transformation, one computes on two separate input halves, one requires two distinct steps in order, and one mixes one- and two-step outputs using an input-dependent weight. Students are feed-forward designs without self-attention. Serial is the reference, Looping repeats a shared stage, Parallel-concat joins independently computed halves, and Parallel-mix combines one- and two-step paths with an input-dependent weight. Within each teacher instance and seed, they use the same data split and 800-update allowance; parameter counts are within 5% of a common target. Arithmetic is not matched: Looping uses about 2.5 times the estimated active multiplies of Serial or either parallel design. We select each checkpoint by validation mean-squared error and compare test mean-squared error on paired teacher instances and training seeds. Appendix B gives the teacher constructions, exact splits, accounting, and initial full-matrix checks.

The language-model studies use a common deduplicated Pile stream and GPT-NeoX tokenizer within each paired training comparison. Quality is held-out negative log-likelihood (NLL), averaged over predicted tokens; lower is better. The quality–cost comparison has three paired seeds and about 200 million training tokens per run; the separate skipping-training comparison has two paired seeds and about 50 million tokens per run. Data order and optimizer schedule are matched within each pair, not across these different studies. A further fixed-weight diagnostic uses independently trained checkpoints and shorter evaluation sequences. Appendix C records configurations, exact token counts, selection rules, and the non-comparable evaluation protocols.

5.2 Controlled Paired Conclusions Depend on the Named Alternative

Even when a student graph appears to match the teacher's computation, its value depends on the alternative it must beat. We therefore compare designated student graphs within the same teacher instance and training seed. For each pair, the log-error advantage is the alternative's $\log_{10}$ test error minus the tested graph's, after adding 10^{-12} to each error; positive values mean the tested graph has lower test error. The targeted test names one alternative per task and requires lower test error in at least seven of nine pairs (three instances by three seeds). Strict order has a separate tolerance check: Serial's test error must be at most 1.10 times the lower of the Serial and Looping errors in at least seven pairs. Passing this boundary does not count as a win against Looping.

Table 1: **Controlled comparisons across targets and named alternatives.** Each row has nine pairs (three teacher instances by three seeds). Wins count lower test error; Boundary counts strict-order pairs in which Serial's error is within 10% of the lower of the two errors. The prespecified thresholds are 7/9 wins and, for strict order, 7/9 boundary passes. Positive median log-error advantage means lower error for the tested graph. Parameter counts are within 5%, but arithmetic is not matched: Looping uses about 2.5 times the estimated active multiplies of the other designs.

Task	Tested graph	Alternative	Wins	Boundary	Median advantage
Repeated	Looping	Serial	9/9	–	0.0535
Separable	Parallel-concat	Looping	9/9	–	0.3981
Conditional	Parallel-mix	Serial	9/9	–	0.1405
Strict order	Serial	Looping	4/9	9/9	−0.0056

In their specified contrasts, Looping and both parallel designs win nine of nine pairs; on strict order, Serial wins four of nine against Looping but passes the separate 10% tolerance check in all nine (Table 1).

The initial matrix names Parallel-concat instead of Looping as the strict-order alternative. On the four teacher-instance–seed cells shared by both comparisons, Serial has lower test error than Parallel-concat in all four but lower error than Looping in only one. This descriptive same-cell contrast holds the teacher, data, seed, and Serial run fixed while changing the trained alternative; the later nine-pair test reuses those runs and is not an independent replication (Appendix B). The named alternative therefore changes the paired conclusion even for the same target and runs. These constructed tasks test named correspondences, not whether they transfer to natural-language training.

5.3 A Wider Shared-Looping Model Trades Training Time for NLL

The natural-language comparison asks what a complete looping design buys at a similar stored-parameter budget. Dense-64 has 64 distinct blocks of width 1,280; Looping-32x2 has 32 distinct blocks of width 1,792 and executes them twice, with iteration and boundary components specified in Appendix C.1. Both make 64 block calls and train on the same approximately 200 million tokens per paired seed. Their widths and parameter-sharing patterns differ, so equal calls do not match arithmetic.

Table 2: **Final quality at matched training tokens and block calls.** Values are validation NLL, mean $\pm$ sample standard deviation over paired seeds 501–503; lower is better. Parameters are approximately matched, but widths and arithmetic per call differ.

Model	Distinct blocks	Width	Parameters (billions)	Validation NLL
Dense-64	64	1,280	1.7449	4.0371 ± 0.0603
Looping-32x2	32	1,792	1.7381	$\mathbf{3.9107} \pm 0.0552$

Looping-32x2 has lower final NLL in all three pairs; the paired Dense-64-minus-Looping-32x2 reduction is 0.1264 ± 0.0053 (mean $\pm$ sample standard deviation). That gain costs training time: Looping-32x2 takes 24.66 rather than 15.76 hours per run on the measured A6000, about 1.56 times as long, and reserves slightly more memory (Appendix Table 4). It is not a speed gain or a quality result at equal time or arithmetic. The three training trajectories appear in Appendix Figure 3.

An auxiliary study trains a new set of Looping-32x2 checkpoints, independent of Table 2, alongside Dense-32 (32 distinct blocks) at the same width of 1,792. The auxiliary Looping-32x2 checkpoints have lower trained-execution NLL than Dense-32 in all three paired seeds (0.0246 ± 0.0104 for the Dense-32-minus-Looping-32x2 difference). Its checkpoints and 512-token evaluation sequences differ from those in Table 2, so the NLL values cannot be pooled. Extra calls, sharing, and loop-specific components remain confounded; neither comparison isolates looping as the cause of the quality ordering. Appendix C.2 gives the complete auxiliary protocol.

5.4 Changing a Trained Loop Count Raises Loss

The equal-width study also supplies a fixed-weight diagnostic. Dense-32 was trained for one traversal of its 32 blocks, and Looping-32x2 for two. We evaluate each checkpoint on its trained execution, then repeat the sequence once more for Dense-32 or remove the second traversal for Looping-32x2. The checkpoint and held-out tokens stay fixed within each edit; the two families have different loop-specific components and opposite edits. Appendix C.3 gives their exact execution rules and reference losses.

On 16 held-out 512-token sequences, all three seeds in each family have positive edited-minus-reference NLL damage: 0.2957 ± 0.0200 when a pass is added to Dense-32, and 1.2191 ± 0.0791 when the trained second pass is removed from Looping-32x2 (Appendix Table 5). These opposite changes both worsen prediction relative to the execution used during training. Their unmatched edits do not identify one common strength of adaptation, nor does fixed-weight damage show which graph would be best if independently trained under the edited execution. The next test asks whether training across shorter paths lowers damage without sacrificing quality when all blocks run.

5.5 Training with Sampled Skipping Lowers Damage but Misses the Quality Criterion

We compare three omission rates of a separate 68-block LayerDrop design with one paired 68-block serial baseline per seed, reused across rates. Unlike the fixed omission in Figure 1(a), LayerDrop samples shorter paths during training. Evaluation asks whether a specified seven-block omission damages the trained checkpoint and whether its full-execution NLL remains close to the serial reference. Each checkpoint selects its own path on calibration data and is tested on disjoint examples, so the measured damage concerns each checkpoint's selected path rather than arbitrary edits. The four prespecified paths and exact split are in Appendix C.4.

For a paired seed, let d_{dense} and d_{drop} be the serial and LayerDrop checkpoints' respective edited-minus-full token-mean NLL on the same held-out sequences. The percentage reduction relative to dense damage is

$$R = 100\left(1 - \frac{d_{\text{drop}}}{d_{\text{dense}}}\right).$$

Both dense-baseline damages are positive. We average the two seed-level percentages for each rate, not a ratio of averaged losses. The prespecified joint target requires positive reduction in both seeds, mean reduction of at least 20%, and full-execution validation NLL no more than 0.02 above each paired dense baseline.

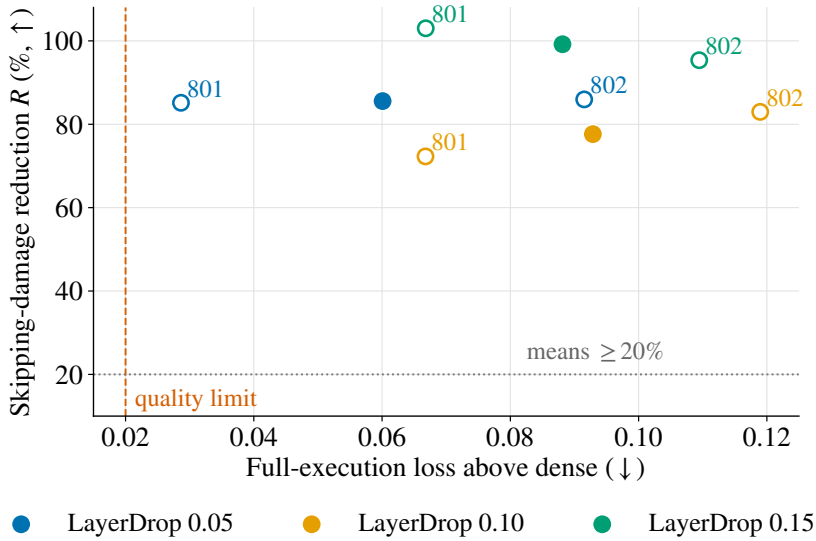

Figure 2: **Training with skipping lowers damage but misses the prespecified quality limit.** Hollow points are individual seeds and filled points are per-rate means. The horizontal axis is each LayerDrop checkpoint's full-execution *validation* NLL minus its paired dense baseline's validation NLL (lower is better); the vertical axis is R from separate held-out skip-path evaluations (higher is better). Numbers identify seeds 801 and 802, and colors identify rates. The vertical 0.02 validation-NLL limit is checked for each hollow seed point; the horizontal 20% damage-reduction target applies to filled per-rate means. The two axes use different evaluation sets.

Every rate lowers damage in both seeds, with per-rate means of 78–99%, but every rate exceeds the 0.02 full-execution loss limit in both seeds. None meets the joint target (Figure 2). One point exceeds 100% because its selected path slightly lowers that checkpoint's loss; it still fails the full-execution limit. These 434-million-parameter screening runs show a quality–damage trade-off under selected paths, not general edit tolerance or a larger-scale pretraining result.

Together, the studies show that a controlled ranking can change when the named alternative changes, while a quality gain can carry a higher training cost. They also show why absolute quality must be reported separately from damage under another execution. The observations are limited to the stated

tasks, seeds, and budgets; they illustrate the comparison contract rather than rank training graphs in general.

6 Alternative Views and Scope

A serious alternative is that ordinary architecture descriptions and end-task quality–cost tables already suffice; demanding complete graph specifications and graph-aware controls could add cost without changing model choice. This view is reasonable when a report already identifies the trained execution, the lifecycle of its weights, feasible alternatives, and each claimed resource dimension. No new reporting format is needed in that case. The missing distinction matters when those facts change the inference: on the same controlled runs, Serial's test-error comparison changes with the named alternative, and a fixed-weight edit does not estimate the quality of weights independently trained on the edited graph.

Another view is to keep serial or a single-operation graph as the default even when a non-serial graph improves one metric. Serial has an established systems stack; our wider looping model lowers validation loss but takes longer to train, and the cost of searching, discarded runs, and systems adaptation can further change the choice. We therefore advocate the complete comparison only when a graph claim could change a research or deployment decision. The declared rule can retain serial or a simpler graph when it meets the objective at lower total cost.

7 Future Research Directions

The objections above leave a selective agenda: researchers should invest in a new training graph when evidence could change the choice over serial and simpler graphs. Three actions make that comparison practical.

1. **Build comparable evidence.** Benchmark builders should train serial, single-operation, and multi-operation graphs under declared data, optimization, and resource allowances. Controlled tasks can first test a graph hypothesis; natural-language training can test whether it transfers. Report the complete graph and path rule, sharing and merge semantics, selection procedure, absolute quality, relevant costs, and failures.
2. **Price candidate selection.** Model designers should propose a small set from task dependencies and resource objectives, then test whether a ranking from short runs survives changes in data, scale, and optimization. Count search, discarded runs, and systems adaptation in the resource comparison. Retain serial or a simpler graph when the proposed graph does not meet its declared decision rule.
3. **Test intended executions and call boundaries.** Declare the full, shorter, or repeated paths a model is meant to support and report quality on each, plus worst-path or upper-quantile per-example damage when failures concentrate. Our fixed-weight edits and selected skipping paths do not establish reliability under arbitrary paths. To investigate a failure, compare states emitted and consumed at corresponding call edges; intervene on one boundary while holding the checkpoint, input, and downstream execution fixed. A useful account must predict held-out behavior and carry to independent runs, tasks, scales, or operation mixtures.

8 Conclusion

A Transformer training-graph claim becomes interpretable when it names the complete trained execution, the lifecycle of its weights, alternative graphs, and a resource allowance. On the same controlled runs, the named alternative changes whether Serial has lower test error; a wider looping model lowers validation NLL but trains longer with width and arithmetic unmatched; and skipping training lowers selected-path damage without meeting the full-execution quality criterion. These outcomes show why trained quality, resource cost, and fixed-weight edit tolerance must be evaluated separately. Composition remains a claim to test under this contract. The declared decision rule can select a proposed graph, serial, or a simpler alternative.

AI USE STATEMENT

We used generative AI tools to assist with literature organization, experimental and figure-generation code, data analysis, and manuscript drafting and revision. The authors reviewed all AI-assisted text and code, verified reported measurements against saved result files, and compiled and visually inspected the final paper. We take responsibility for the complete content of this work, including text, claims, code, and artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

The anonymous repository with the manuscript and available supplementary artifacts is at `https://anonymous.4open.science/r/position-pretrain-72BB/README.md`.

The submitted supplementary directory `data/` contains a `README.md`. The README index maps each experiment to saved result tables, task definitions, available training configurations, evaluation scripts, and prespecified decision records used in the paper. The main text states the comparison budgets, model-selection rules, repeated units, and aggregation conventions needed to interpret the reported results. For the LayerDrop study, the archive retains configuration hashes but not the historical hashed configuration files.

REFERENCES

Han Cai, Chuang Gan, Tianzhe Wang, Zhekai Zhang, and Song Han. Once-for-all: Train one network and specialize it for efficient deployment. In *ICLR*, 2020. URL https://arxiv.org/abs/1908.09791.

Mostafa Dehghani, Stephan Gouws, Oriol Vinyals, Jakob Uszkoreit, and Łukasz Kaiser. Universal transformers. In *ICLR*, 2019.

Mostafa Elhoushi, Akshat Shrivastava, Diana Liskovich, Basil Hosmer, Bram Wasti, Liangzhen Lai, Anas Mahmoud, Bilge Acun, Saurabh Agarwal, Ahmed Roman, Ahmed A Aly, Beidi Chen, and Carole-Jean Wu. Layerskip: Enabling early exit inference and self-speculative decoding. In *ACL*, 2024.

Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey. *Journal of Machine Learning Research*, 20(55):1–21, 2019. URL https://arxiv.org/abs/1808.05377.

Angela Fan, Edouard Grave, and Armand Joulin. Reducing transformer depth on demand with structured dropout. In *ICLR*, 2020.

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

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

Gao Huang, Yu Sun, Zhuang Liu, Daniel Sedra, and Kilian Q. Weinberger. Deep networks with stochastic depth. In *ECCV*, 2016. URL https://arxiv.org/abs/1603.09382.

Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. ALBERT: A lite BERT for self-supervised learning of language representations. In *ICLR*, 2020.

Ziyue Li, Yang Li, and Tianyi Zhou. Skip a layer or loop it? learning program-of-layers in LLMs, 2026. URL https://arxiv.org/abs/2606.06574.

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

Nikunj Saunshi, Nishanth Dikkala, Zhiyuan Li, Sanjiv Kumar, and Sashank J. Reddi. Reasoning with latent thoughts: On the power of looped transformers. In *ICLR*, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/2676109d49d1eb26d6bc584a8f556305-Paper-Conference.pdf.

Qi Sun, Marc Pickett, Aakash Kumar Nain, and Llion Jones. Transformer layers as painters, 2024. URL https://arxiv.org/abs/2407.09298.

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, 2026. URL https://arxiv.org/abs/2609.01343.

Liu Yang, Kangwook Lee, Robert Nowak, and Dimitris Papailiopoulos. Looped transformers are better at learning learning algorithms, 2023. URL https://arxiv.org/abs/2311.12424.

A Worked Specification of a Multi-Operation Graph

This appendix expands the candidate introduced in Section 3. It illustrates how a merge rule and parameter identities determine a complete training computation; it does not report a trained multi-operation model.

Let a and b index two trainable block positions in a serial design, let F_a and F_b be their mappings, and let h be an incoming token-state matrix whose shape each block preserves. Their serial execution is $F_b(F_a(h))$. A parallel merge that adds the two residual updates instead computes

$$P_{a,b}(h) = h + \big(F_a(h) - h\big) + \big(F_b(h) - h\big).$$

This rule differs from averaging $F_a(h)$ and $F_b(h)$ even though both computations have two shared-input branches.

For a six-block serial reference, one fixed-path candidate skips F_2, runs F_3 and F_4 in parallel, and calls F_5 twice:

$$F_6(F_5(F_5(P_{3,4}(F_1(h))))) \,.$$

Both branches consume the same state and finish before the stated merge. The two calls to F_5 share one trainable parameter set. Every training input follows this path; no call-specific identifier, normalization change, or auxiliary loss is added, and the other block and attention conventions follow the serial reference. These position labels specify a graph to train from initialization, not an edit of weights pretrained on the serial graph.

The candidate makes six block calls: one each to F_1, F_3, F_4, and F_6, and two to F_5. It uses five distinct block-parameter sets; skipping F_2 and merging F_3 with F_4 change the serial dependency path. These structural counts do not establish training quality, measured speed, or preference over a simpler graph.

B Controlled Tasks and Full-Matrix Checks

This appendix gives the construction and initial checks behind Section 5.2. The controlled tasks test whether complete student graphs make different predictions for known target computations. They do not identify a universally best graph for a task name.

B.1 Teachers, Students, and Matched Runs

Each fixed teacher receives a 16-dimensional Gaussian vector and uses linear maps followed by hyperbolic tangent nonlinearities. The repeated teacher applies the same transformation three times. The separable teacher transforms the first and last eight coordinates independently and concatenates their outputs. The strict-order teacher applies two distinct transformations in a fixed sequence. The conditional teacher computes outputs after the first and second transformations and mixes them: an input-dependent scalar sigmoid weight multiplies the two-transformation output, and its complement

multiplies the first. Each teacher instance supplies 2,048 training, 1,024 validation, and 2,048 test examples.

Students are feed-forward networks without self-attention. Serial is the reference. Looping calls one shared stage three times. Parallel-concat executes two paths on the input halves and concatenates their outputs; Parallel-mix computes both one- and two-step paths and combines them with an input-dependent weight. These merges are not fifth and sixth depth operations. Within a task and teacher instance, the student designs retain the same data split and 800-update training allowance. The frozen configuration uses AdamW with learning rate 10^{-3}, weight decay 10^{-4}, batch size 128, and validation every 40 updates. Parameter counts are 11,953 for Serial, 12,048 for Looping, 12,022 for Parallel-concat, and 11,970 for Parallel-mix, all within 5% of the 11,953-parameter target. The designs do not have matched arithmetic: estimated active multiplies per example are 11,730 for Serial and Parallel-concat, 11,746 for Parallel-mix, and 29,516 for Looping.

We select each run's checkpoint by lowest validation mean-squared error and report test mean-squared error averaged over output coordinates and examples. The initial matrix trains every combination of four tasks and four student graphs using two teacher instances and two training seeds. This yields 64 trained runs; the 16 designated pairs below count selected comparisons, not task–graph combinations. The later targeted comparisons extend to three instances and three seeds; they reuse the initial instance–seed training runs and are not an independent replication. The archived rows and accounting are indexed by `data/README.md`.

B.2 Initial Designated Contrasts

The initial matrix designates one tested graph and one alternative per task. The tested graph has lower test error in all four paired instance–seed runs for each designated contrast (Table 3).

Table 3: Initial designated contrasts. Each task has two teacher instances and two training seeds, giving four paired test-error comparisons. These contrasts precede the nine-pair comparisons in Table 1; the later test reuses the initial instance–seed training runs, but names different alternatives for two tasks.

Task	Tested graph	Alternative	Lower test error
Strict order	Serial	Parallel-concat	4/4
Repeated	Looping	Parallel-concat	4/4
Separable	Parallel-concat	Looping	4/4
Conditional	Parallel-mix	Serial	4/4

The observed mean log-error advantage across the 16 designated pairs is 0.8787; under 10,000 graph-label permutations within each task instance and seed, its one-sided p-value is below 10^{-4} (plus-one correction). The 16/16 result concerns the designated alternatives only. In the targeted nine-pair comparisons, the Repeated task names Serial rather than Parallel-concat as the alternative, and the Strict order task names Looping rather than Parallel-concat. On the four reused strict-order instance–seed cells, Serial has lower test error than Parallel-concat in all four and than Looping in one. This descriptive overlap changes the named alternative without adding an independent replication. Serial fails the prespecified seven-win check against Looping (4/9), although it passes the separate 10% tolerance check in all nine pairs. The archived Stage B configuration records these targeted comparison cells and excludes the initial instance–seed coordinates from the added runs. Its configuration, paired rows, decision record, and nine strict-order pairs are indexed in `data/README.md`.

C Language-Model Training and Execution Protocols

This appendix supplies the implementation and extended evidence behind the language-model comparisons in Section 5. The three studies have different checkpoints and evaluation sequences; their losses must not be pooled into one result.

C.1 Matched-Token Quality, Cost, and Trajectories

The Dense-64 and Looping-32x2 comparison uses a common deduplicated Pile stream, the GPT-NeoX tokenizer, a held-out validation split, and three paired seeds. Runs within each pair share data order and optimizer schedule and train for 200,015,872 tokens. Both use 2,048-token sequences, AdamW, bfloat16 precision, activation checkpointing, and one NVIDIA RTX A6000 per run. Dense-64 has 64 distinct blocks of width 1,280. Looping-32x2 has 32 distinct blocks of width 1,792 and makes two passes, adding a learned iteration embedding and a learned scalar for residual updates on each pass. After both passes it adds the sequence input through an outer residual connection and applies boundary normalization. The two designs make 64 block calls per forward evaluation and have similar parameter counts, but differ in width, sharing, and per-call arithmetic.

Figure 3 shows that Looping-32x2 has lower validation negative log-likelihood (NLL) throughout the later part of all three paired training trajectories. Table 2 in the main text reports final NLL, and Table 4 below reports measured training cost. These trajectories do not make the unmatched arithmetic a matched-compute comparison.

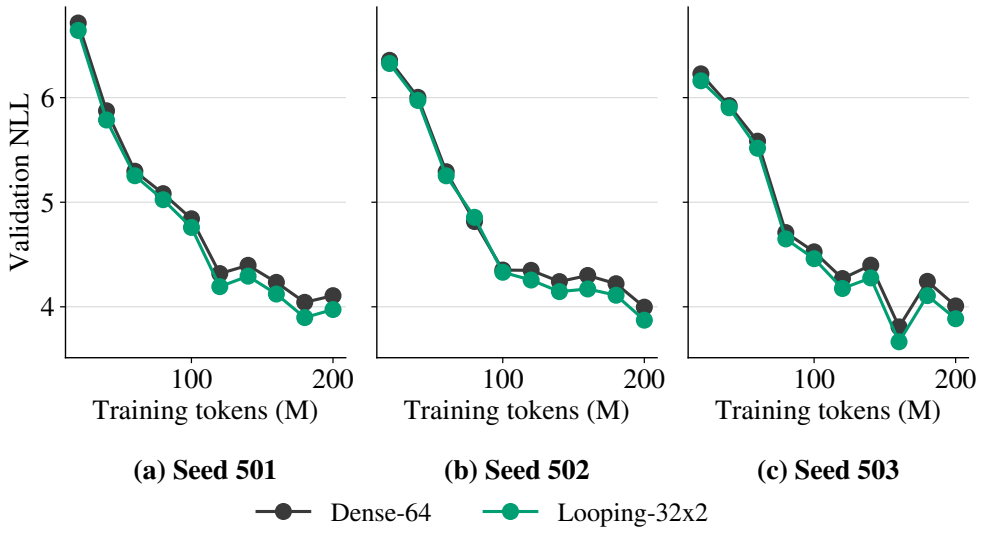

Figure 3: **Paired validation trajectories for the wider looping design.** Panels (a)–(c) show seeds 501–503. Each panel uses a paired data stream; the horizontal axis is cumulative training tokens in millions and the shared vertical axis is validation NLL, where lower is better. The designs have similar parameter counts and equal block calls, but different widths and sharing patterns, so their arithmetic per call differs.

The reported hours and throughput describe training rather than deployment latency. Peak memory is framework-reserved memory in GiB (2^{30} bytes).

Table 4: **Measured cost for the paired language-model comparison.** Values are arithmetic means over seeds 501–503 on one NVIDIA RTX A6000 per run. Throughput and hours refer to training, not deployment.

Model	Tokens/second	Hours/run	Peak reserved (GiB)
Dense-64	3,538	15.76	34.43
Looping-32x2	2,253	24.66	35.72

The source curves, paired summary, and training configurations are indexed by `data/README.md`.

C.2 Equal-Width Auxiliary Comparison

An auxiliary study asks whether the quality ordering persists when width is the same. Its Dense-32 and Looping-32x2 checkpoints are a separate set trained independently of the primary comparison. Both designs have width 1,792, 32 distinct blocks, approximately 1.74 billion parameters, and an approximately 200-million-token training budget. Dense-32 executes its block sequence once; Looping-32x2 executes it twice with its loop-specific components. On 16 held-out 512-token sequences, Looping-32x2 has lower trained-execution NLL in all three paired seeds: the mean Dense-32-minus-Looping-32x2 difference is 0.0246 ± 0.0104.

These checkpoints are independent of the primary Dense-64 comparison and use shorter evaluation sequences, so their NLL values are not numerically comparable with Table 2. Matching width removes that one difference but not the extra calls, sharing, or loop-specific components. Neither comparison isolates looping as the cause of the quality ordering. The paired results and configurations are indexed by `data/README.md`.

C.3 Fixed-Weight Loop-Count Edits

The diagnostic in Section 5.4 reuses the independently trained Dense-32 and Looping-32x2 checkpoints of the equal-width study. Dense-32 was trained for one traversal of its 32-block sequence (32 calls); Looping-32x2 was trained for two (64 calls). Editing Dense-32 repeats its block sequence without adding parameters. Editing Looping-32x2 removes the second traversal. Its first pass retains the learned iteration embedding, the scalar that scales its residual update, and the outer residual connection and normalization at the sequence boundary. The Dense-32 edit does not gain these loop-specific components.

Each checkpoint is evaluated before and after its edit on the same 16 held-out 512-token Pile sequences. It contributes one edited-minus-trained-execution token-mean NLL difference, with positive damage indicating worse prediction. Table 5 averages these differences over three seeds within each family. The two families use opposite edits and are not matched in arithmetic; their magnitudes cannot rank a common strength of adaptation.

Table 5: **Fixed-weight loop-count edits increase loss in every checkpoint.** Reference NLL is measured under each checkpoint's trained traversal count; damage is edited minus reference NLL on the same 16 held-out 512-token sequences. Values are mean $\pm$ sample standard deviation over three seeds. These independently trained checkpoints and shorter evaluation sequences make reference NLL incomparable with Table 2.

Training design	Reference NLL	Passes: trained $\rightarrow$ edited	NLL damage	Positive seeds
Dense-32	4.2767 ± 0.0079	$1 \rightarrow 2$	0.2957 ± 0.0200	3/3
Looping-32x2	4.2521 ± 0.0035	$2 \rightarrow 1$	1.2191 ± 0.0791	3/3

C.4 Skipping-Training Path Selection

The separate LayerDrop study uses a 68-block, width-608 backbone with approximately 434 million parameters. Each omission probability $p \in \{0.05, 0.10, 0.15\}$ is compared with the same per-seed 68-block serial baseline across rates. The two paired seeds per rate use a common data order, tokenizer, and optimizer schedule, with approximately 50,003,968 training tokens per run. Runs use 2,048-token sequences, AdamW, bfloat16 precision, activation checkpointing, and one NVIDIA RTX A6000.

Four seven-block skipping patterns were prespecified: approximately evenly spaced interior blocks, a consecutive middle group, the last seven blocks, and a seeded random group. Each omits about 10.3% of the 68 blocks. For each checkpoint, the pattern with the smallest edited-minus-full NLL on 16 calibration sequences is selected, then its damage is evaluated on 32 separate held-out 512-token sequences. Serial and LayerDrop checkpoints each select their own pattern. The comparison therefore evaluates this selection procedure, not one identical edit applied to both models.

Table 6 reports both quantities for every rate and seed. Damage is edited-minus-full token-mean NLL on the selected path and separate held-out sequences. The quality gap is full-execution validation NLL minus the paired dense model's validation NLL, so the two kinds of columns use different evaluation sets.

Table 6: **Selected-path damage and full-execution quality by rate and seed.** Smaller damage and a smaller validation gap are better; R is the percentage reduction relative to paired dense damage. The quality criterion requires a validation gap no greater than 0.02 in each seed. Values are rounded from `data/gp2_004_paired_seed_table.csv`.

Rate	Seed	Dense damage	LayerDrop damage	R (%)	Validation gap
0.05	801	0.0115	0.0017	85.2	0.0286
0.05	802	0.0272	0.0038	86.0	0.0915
0.10	801	0.0115	0.0032	72.3	0.0668
0.10	802	0.0272	0.0046	83.0	0.1190
0.15	801	0.0115	−0.0003	103.0	0.0668
0.15	802	0.0272	0.0013	95.4	0.1095

For each paired seed, the relative damage reduction R uses the two checkpoints' edited-minus-full token-mean NLL on the same held-out sequences, as defined in Section 5.5. The two seed percentages are then averaged for each rate; this is not the ratio of two seed-averaged losses. Both dense-baseline damages are positive. The prespecified joint criterion additionally requires final full-execution validation NLL no more than 0.02 above the paired dense model in each seed. The preregistration, selection script, and paired results are indexed by `data/README.md`; the historical hashed LayerDrop configuration files are not included in the submitted data directory.