DECODE-BRANCH TRANSFORMERS: DECOUPLING THE PRIMARY PREFILL PATH FROM ADDITIONAL DECODE COMPUTATION

Anonymous authors
Paper under double-blind review

ABSTRACT

As large language models serve ever more requests, cumulative inference cost is growing relative to the one-time cost of training. In common serving regimes the two inference phases place different demands on hardware: prompt prefill runs in parallel and tends to be compute-bound, whereas autoregressive decode is sequential and memory-traffic-bound. Conventional width or depth scaling raises both costs together, since every added layer is evaluated in both phases and enlarges the weights read at every decode step. We instead ask whether additional learned computation can be allocated to continuation prediction while preserving the prompt-wide primary computation and a single persistent key–value (KV) cache. We realize this separation with the *Decode-Branch Transformer*. Its primary path is a complete causal language model that alone processes the prompt and writes the KV cache. At inference, the decode branch is omitted at earlier prompt positions and activated at the final prompt position and each generated position, where it adds continuation-prediction computation without writing persistent state or influencing the primary path. The paths share all major attention, MLP, and output matrices and use separate token embeddings with lightweight coupling. Grouped decode reuses each loaded weight tile and primary-cache region across both paths, so the added arithmetic does not proportionally increase the dominant memory traffic or decode latency. Across matched-training-token comparisons, Decode-Branch attains lower validation loss across architectures and data configurations. A training-compute control further favors interacting trajectories over additional tokens. In MoE models, its structural separation makes the primary and branch expert fan-outs independent knobs for trading prompt cost, continuation cost, and predictive quality. We study two allocation regimes: fixing prefill expert computation while increasing decode computation, and fixing decode expert computation while reallocating the expert budget between the two paths. Together, these experiments expose a prefill–decode–quality trade-off and establish a structural opportunity for phase-specific expert allocation.

1 INTRODUCTION

The cost of serving large language models is determined by two different phases. During *prefill*, all prompt tokens are processed in parallel, allowing weights to be reused across a large batch of token states. During autoregressive *decode*, tokens are generated sequentially; each new token traverses the model weights and reads the growing KV cache. Prefill therefore tends toward arithmetic throughput limits, while decode at small-to-moderate batch sizes is often constrained by memory traffic. This mismatch is especially consequential for long-context inference and interactive agents, motivating architectures that vary continuation computation without proportionally changing prompt computation or persistent state.

Additional computation before prediction is already a useful capability-scaling resource. Chain-of-thought, self-consistency, and repeated sampling spend compute on intermediate or alternative trajectories (Wei et al., 2022; Wang et al., 2023; Brown et al., 2024); adaptive methods allocate it according to prompt difficulty (Snell et al., 2025); and pause tokens add hidden computation before an answer (Goyal et al., 2024; Kim et al., 2025). These methods commonly spend the extra

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.

budget through generated positions, samples, or serial latent steps. We instead ask whether it can be incorporated into one ordinary decode step.

Our phase-decoupled objective has four properties: (i) a complete primary causal path that alone processes the prompt; (ii) no persistent branch state, leaving the KV cache unchanged; (iii) additional learned computation at continuation-prediction steps; and (iv) substantial sharing of layer weights and cached state. Conventional scaling cannot satisfy these simultaneously because every added layer or parameter participates in both phases and is revisited at every generated token.

We instantiate this objective with the Decode-Branch Transformer. The primary path never depends on the branch, while the branch reads primary states and KV without writing persistent state or feeding back into the primary path. Both paths share the major attention, MLP, and output matrices, with separate token embeddings and lightweight learned coupling vectors. The branch is omitted at earlier prompt positions and activated at the final prompt position and each generated position. A mixture likelihood over the two next-token distributions gives both trajectories a direct training signal.

This sharing matters during memory-bound decode. Grouped execution lets one fetched weight tile and primary-cache region serve both paths, increasing arithmetic intensity without enlarging the unique shared weights or KV entries referenced by a step. In sparse MoE models, router replay further lets the branch reuse the primary selected experts with its own mixture weights. Alternatively, independent path fan-outs expose direct controls over computation: the primary fan-out determines prompt expert arithmetic, while the sum of primary and branch fan-outs determines continuation expert arithmetic.

We evaluate Decode-Branch through controlled NanoGPT data scaling, dense LLaMA-style model scaling, and sparse MoE configurations. Under matched token budgets it consistently lowers validation loss. Component ablations isolate the contributions of primary-to-branch coupling and mixture readout, and an approximate training-compute comparison supports allocating computation across interacting paths. Complementary MoE sweeps fix either prefill or decode expert computation, exposing the resulting prefill–decode–quality trade-off.

Our contributions are:

- **Architecture.** We introduce an asymmetric shared-KV design in which the primary path alone processes the prompt and determines persistent state, while a continuation-only branch reads that state and shares the large layer matrices.
- **Empirical quality.** Across data scaling, dense model-size scaling, and sparse configurations, Decode-Branch consistently reduces matched-token validation loss; component and compute controls isolate the value of the additional interacting trajectory.
- **Phase-specific MoE allocation.** Independent primary and branch fan-outs make prefill and decode expert computation separately configurable, enabling workload-specific trade-offs with predictive quality.

2 Related work

Additional computation before prediction. Model and data scaling increase training and serving cost (Kaplan et al., 2020; Hoffmann et al., 2022), while test-time methods allocate computation before committing to a prediction. Chain-of-thought exposes intermediate reasoning (Wei et al., 2022); self-consistency and repeated sampling evaluate candidate trajectories (Wang et al., 2023; Brown et al., 2024); adaptive methods vary the budget with prompt difficulty (Snell et al., 2025); and pause tokens add learned hidden computation (Goyal et al., 2024; Kim et al., 2025). These methods generally add positions, samples, or serial latent steps rather than changing the internal computation of one decode step.

Parallel latent computation. Parallel Scaling transforms an input into shared-backbone streams (Chen et al., 2025); Hidden Decoding retains independently embedded stream KV (Liu et al., 2026); and State-Prediction Separation preserves prediction-token KV in a local window (Monea et al., 2026). Their additional streams participate in prompt processing or retain stream-specific history, so prompt computation, persistent state, or both grow with expanded computation.

Phase-decoupled computation. PHD-Transformer and Parallel Loop Transformer (PLT) are the closest precedents. PHD interleaves original and hidden-decoding token copies, retains only original-token KV for long-range attention, and uses structured attention during training (Wu et al., 2025b). PLT staggers loop states across tokens, sharing first-loop global KV while augmenting later loops with gated local state (Wu et al., 2025a). Decode-Branch instead keeps same-position trajectories that can be grouped under ordinary causal attention (Dao et al., 2022); the primary never reads the branch, while the branch receives same-layer primary coupling. PHD and PLT predict from the final copy or loop, whereas Decode-Branch trains a mixture of both distributions. We therefore include a controlled PHD-2 comparison.

Prediction and state sharing. Multi-token prediction adds heads for future tokens (Gloeckle et al., 2024), and speculative decoding uses draft and verification (Leviathan et al., 2023). Decode-Branch instead predicts the same next token through interacting trajectories. Its KV sharing is related to multi-query and grouped-query attention (Shazeer, 2019; Ainslie et al., 2023), but operates across full residual flows. Router replay similarly aligns selected experts across paths (Shazeer et al., 2017; Fedus et al., 2022).

3 Background

Given a sequence $x_{1:T}$, an autoregressive model factorizes $p(x_{1:T}) = \prod_{t=1}^{T} p(x_t \mid x_{<t})$. For residual states $H^\ell \in \mathbb{R}^{T\times d}$, a Transformer forms

$$Q^\ell = H^\ell W_Q^\ell, \qquad K^\ell = H^\ell W_K^\ell, \qquad V^\ell = H^\ell W_V^\ell, \tag{1}$$

and applies causal attention before the residual MLP update (Vaswani et al., 2017). During inference, prompt positions construct persistent KV once, whereas each generated position reads the accumulated cache; the implications for Decode-Branch are developed in Section 4.4.

An MoE layer replaces the dense MLP with E experts and a router (Shazeer et al., 2017; Fedus et al., 2022). For representation h, it selects $I(h) = \mathrm{TopK}(r(h), k)$ and returns

$$\mathrm{MoE}(h) = \sum_{e\in I(h)} w_e(h)\,\mathrm{Expert}_e(h). \tag{2}$$

Thus storage scales with the expert bank while per-token routed arithmetic scales with k. Some architectures also apply shared experts to every token; these remain separate from the top-k set. Prefill groups many tokens by expert, whereas lightly batched decode forms smaller expert batches and may incur expert-parallel communication.

4 Decoupling Prefill and Decode Computation with Decode-Branch

In this section, we present Decode-Branch as a concrete realization of phase-decoupled prefill and decode computation. Decode-Branch augments a standard primary trajectory with one branch residual-state trajectory while sharing the Transformer's large matrices. Let d be the model dimension and let $S_1^\ell, S_2^\ell \in \mathbb{R}^{T\times d}$ denote primary and branch residual states before layer ℓ. The two paths begin from distinct embedding tables,

$$S_1^0 = \mathrm{RMSNorm}(E_1[x]), \qquad S_2^0 = \mathrm{RMSNorm}(E_2[x]). \tag{3}$$

The independent embeddings are the only additional vocabulary-scale parameters. All subsequent dense projections are shared.

4.1 Asymmetric shared-KV attention

For normalized primary states $N_1 = \mathrm{RMSNorm}(S_1^\ell)$, shared projections produce

$$Q_1 = \mathrm{RoPE}(N_1 W_Q), \quad K_1 = \mathrm{RoPE}(N_1 W_K), \quad V_1 = N_1 W_V. \tag{4}$$

The primary attention is exactly causal self-attention,

$$A_1 = \mathrm{CausalAttn}(Q_1, K_1, V_1) W_O. \tag{5}$$

The decode branch produces only a new query, $Q_2 = \mathrm{RoPE}(N_2 W_Q)$, where $N_2 = \mathrm{RMSNorm}(S_2^\ell)$. Its coupled query and attention output are

$$\widetilde{Q}_2 = Q_2 + a^\ell \odot Q_1, \qquad A_2 = \mathrm{CausalAttn}(\widetilde{Q}_2, K_1, V_1) W_O. \tag{6}$$

where a^ℓ is a learned vector broadcast across tokens. Both attention computations use ordinary causal semantics and require no interleaved or method-specific attention mask, so each is compatible with standard causal attention kernels. Grouped or fused query execution lets the two paths share reads of the primary keys and values. Importantly, S_1 never reads S_2; the primary path is unchanged by whether a decode branch is evaluated.

After residual addition, $\bar{S}_i = S_i^\ell + A_i$, the shared MLP produces intermediate states. For a generic gated or non-gated MLP, write $H_i = f_{\mathrm{up}}(\mathrm{RMSNorm}(\bar{S}_i))$ for the activated intermediate representation and W_D for its output projection. We set $M_1 = H_1 W_D$ and $\widetilde{H}_2 = H_2 + b^\ell \odot H_1$, then use

$$M_2 = \widetilde{H}_2 W_D + c^\ell \odot M_1, \qquad S_i^{\ell+1} = \bar{S}_i + M_i. \tag{7}$$

The learned coupling vectors a^ℓ, b^ℓ, and c^ℓ strengthen information transfer from the primary path to the decode branch at the attention-query, MLP-intermediate, and MLP-output levels, respectively. This interaction remains asymmetric: it enriches the branch representation without introducing any dependence of the primary path on the decode branch.

4.2 Mixture next-token objective

A shared output matrix maps the final states to logits z_1, z_2, yielding distributions $p = \mathrm{softmax}(z_1)$ and $q = \mathrm{softmax}(z_2)$. For target token y_t, we minimize

$$\mathcal{L}_t = -\log(\alpha_t p_t(y_t) + (1 - \alpha_t) q_t(y_t)). \tag{8}$$

For all Decode-Branch experiments, we set

$$\alpha_t = \mathrm{clip}_{[0.5,1]}\left(\sum_v p_t(v)^2\right). \tag{9}$$

We use α_t as a confidence score based on the primary distribution's concentration. Before clipping, $\sum_v p_t(v)^2$ is exactly the probability that two independent samples from the primary distribution produce the same token—its collision probability; it is high when the distribution is concentrated and low when the primary path is uncertain. We compute α_t directly from the primary distribution and allow gradients to propagate through this dependence. Clipping at 0.5 guarantees that the mixture weight on the primary path is always at least one half, so the predictive distribution used at inference—and scored by the validation loss—remains primary-dominated. We use the same mixture likelihood for training and validation.

4.3 Combining Decode-Branch with MoE: router replay

To preserve the primary path's referenced expert-weight set across both paths, we introduce *router replay*. Let the primary router produce logits r_1 and top-k expert indices I_1. The branch router evaluates its own logits r_2 but gathers routing weights only at the primary indices:

$$I_1 = \mathrm{TopK}(r_1), \qquad w_2 = \mathrm{Normalize}\big(\mathrm{softmax}(r_2)[I_1]\big). \tag{10}$$

Both paths execute the same selected experts with potentially different combination weights. Router replay thus preserves a path-specific routing signal rather than copying the primary probabilities, while keeping the set of referenced expert weights unchanged across paths. Each selected expert is applied to both hidden states, so expert arithmetic increases. Grouping the two paths' inputs allows the selected expert weights to be loaded once and reused during decoding.

4.4 Training and inference cost analysis

A roofline view clarifies the phase asymmetry. If a phase performs F arithmetic operations while moving B bytes, its execution time is bounded by

$$T_{\mathrm{phase}} \gtrsim \max\left(\frac{F}{\Pi}, \frac{B}{\beta}\right), \tag{11}$$

where Π and β are effective compute throughput and memory bandwidth. Prefill exposes many token states per weight load and therefore tends toward the compute-bound side. Decode processes one new position per request and repeatedly streams weights and accumulated KV, so at small-to-moderate batch sizes it tends toward the memory-bound side. This distinction is the basis for adding shared computation specifically at continuation positions.

Let the prompt be $x_{1:S}$. Decode-Branch inference has three steps:

1. **Prompt-wide prefill.** The primary path processes positions $1, \ldots, S$ exactly as a one-flow Transformer and writes their keys and values. No branch computation is performed over the prompt.

2. **Continuation boundary.** The branch is evaluated once at the final prompt token using $E_2[x_S]$, retained primary coupling intermediates, and the completed primary KV cache. The mixture predicts x_{S+1}.

3. **Autoregressive continuation.** At each later position, the primary path appends one set of KV, both paths form queries over the shared cache, and their mixture predicts the next token.

Prefill. The prompt-wide primary graph, FLOPs, and KV construction match the one-flow Transformer. Online prefill retains only the final position's coupling intermediates Q_1, H_1, M_1 for the boundary evaluation, which adds one token-equivalent branch operation rather than prompt-length branch computation.

Decode. The branch generates queries but no persistent KV entries. Grouped execution lets shared matrices and primary KV serve both states. Although backbone arithmetic approximately doubles, unique shared weights and persistent cache remain near those of the primary model. When execution remains memory-bound, particularly for single-device serving without inter-device communication, the branch can exploit underutilized arithmetic and incur substantially less additional latency than width or depth scaling. This advantage fades near the compute roof.

More explicitly, the dominant off-chip traffic of a conventional decode step can be viewed as $B_{\text{decode}} = B_{\text{weight}} + B_{\text{KV}}$. Decode-Branch does not introduce a second backbone or KV history: grouped matrix operations reuse each loaded parameter tile, and both attention queries read the same primary cache region. Thus the number of *unique* weight and KV bytes referenced remains close to that of the primary model even though each byte supports more arithmetic. This distinction is central to the design. Adding independently stored layers or width increases B_{weight} directly, while retaining branch-specific KV would increase B_{KV} with context length. Decode-Branch instead trades available arithmetic throughput for continuation computation without enlarging the dominant backbone-weight traffic or persistent KV footprint.

Prediction and training. Every continuation prediction uses the mixture in Equation 8, matching the reported validation objective. The prompt-side saving is an inference property: during training, every position predicts a successor, so both paths run over the full sequence. Decode-Branch consequently uses approximately twice the dominant shared-backbone FLOPs and activation memory. Its additional parameters are a second embedding table and per-layer coupling vectors; the latter contribute less than 0.1% of non-embedding parameters, while the elementwise couplings and mixture add little arithmetic.

4.5 Experiments

We first test whether Decode-Branch improves predictive quality under controlled data scaling, then isolate its components and evaluate dense and sparse LLaMA-style model scaling. Phase-specific expert allocation is studied separately in Section 5. Full optimizer settings appear in Appendix A.

Unless otherwise stated, each matched comparison uses the same primary/base architecture and training-token budget. Decode-Branch adds a second embedding table and evaluates two paths during training, so these comparisons are not matched in total parameter count or training FLOPs.

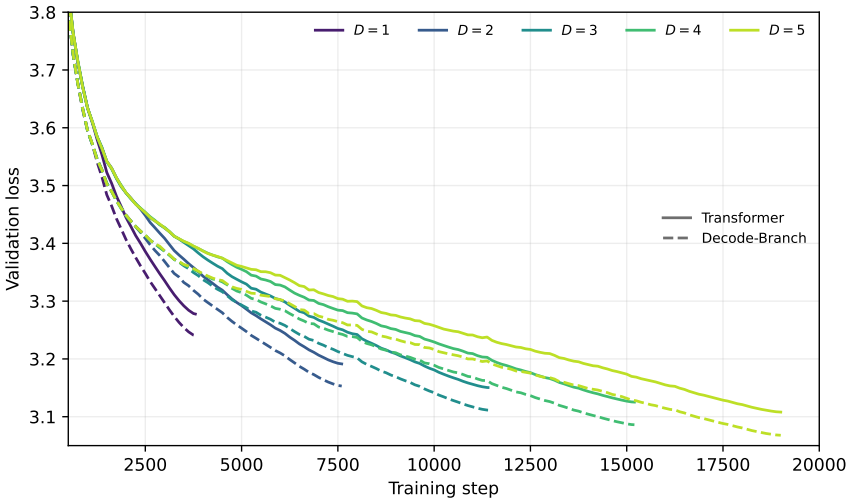


Figure 1: NanoGPT validation loss versus training step at five matched token budgets. Color identifies D; solid and dashed curves denote the Transformer and Decode-Branch.

4.5.1 NanoGPT data scaling and training-compute comparison.

We follow a modded-NanoGPT Track 3 configuration (Jordan & modded-NanoGPT contributors, 2024): 12 layers, width 768, head dimension 128, vocabulary size 50,257, sequence length 1,024, and a global batch of 512 sequences. On FineWeb, 3,800 optimizer steps correspond to approximately 2B tokens. A multiplier $D \in \{1, 2, 3, 4, 5, 10, 20\}$ sets the run length to $3{,}800D$ steps, so the Transformer and Decode-Branch see identical tokens at each D. Dense matrices use Muon (Jordan et al., 2024); embeddings, output heads, and vectors use AdamW.

Figure 1 shows the five $D \leq 5$ trajectories. Decode-Branch finishes below the Transformer at every budget, and the separation persists as training data grows.

To characterize this trend, we fit each family's endpoints to

$$\ell(D) = L + AD^{-\gamma}, \tag{12}$$

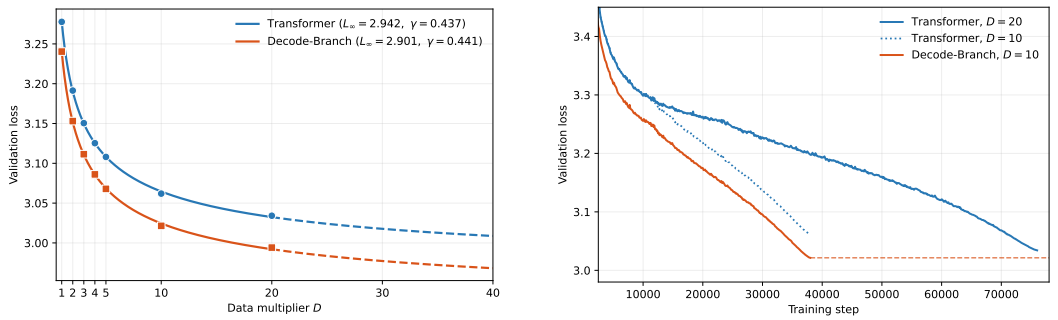
the fixed-model-size slice of the Chinchilla law (Hoffmann et al., 2022). The left panel of Figure 2 fits all seven budgets and extrapolates from $D = 20$ to $D = 40$. Decode-Branch lowers the fitted asymptote from 2.9416 to 2.9013.

One Decode-Branch step evaluates the shared backbone on two states, using approximately twice the dominant training FLOPs. In Figure 2 (right), the $D = 10$ Transformer is the same-step reference; for approximate compute matching, we compare the $D = 10$ Decode-Branch with the $D = 20$ Transformer, which is slightly more expensive because branch attention omits a second set of keys and values. Decode-Branch reaches lower loss, favoring interacting trajectories over additional tokens.

4.5.2 Component ablations and PHD comparison.

PHD-2 is the closest architectural baseline. It interleaves an original and hidden-decoding copy of each token; a hidden copy attends to causal original copies and itself but not earlier hidden copies, and only hidden copies produce next-token logits. Interleaving expands length T to $2T$, with token t occupying positions $2t$ and $2t + 1$; we retain these doubled coordinates to match the original PHD construction.

Our minimal Decode-Branch ablation removes the $a/b/c$ couplings and, like PHD-2, predicts only from the branch. It retains separate embeddings, same-position coordinates, and primary-to-branch shared-KV attention. These choices isolate the architectural difference from the later coupling and readout improvements. Figure 3 shows that this variant matches PHD-2 at smaller budgets and improves over it as data grows. Adding the coupling vectors yields a small consistent gain; restoring the probability mixture performs best. We restrict the comparison to PHD-2 because it matches the two representations and dominant training arithmetic. More PHD copies would also increase



(a) Data scaling and fitted curves. **(b)** Approximate training-compute match.

Figure 2: NanoGPT data scaling and approximate training-compute comparison. Left: three-parameter fits over seven measured budgets, with dashed extrapolation beyond $D = 20$. Right: the $D = 20$ Transformer trains for 76,000 steps, while the $D = 10$ Decode-Branch trains for 38,000 steps with two path evaluations per step.

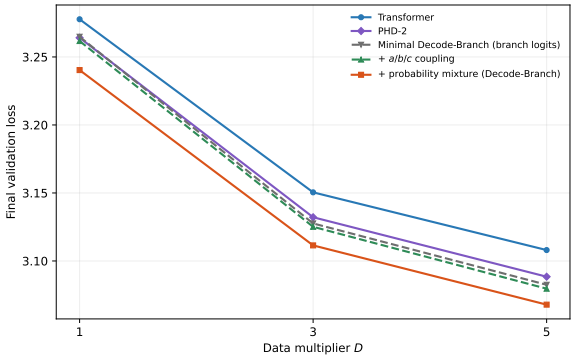


Figure 3: NanoGPT component ablations at three matched token budgets. Minimal Decode-Branch uses shared-KV branch computation and a branch-only readout; successive variants add asymmetric coupling and the primary–branch probability mixture.

sequence expansion, activation memory, and FLOPs, changing the compute allocation rather than providing an aligned architectural comparison.

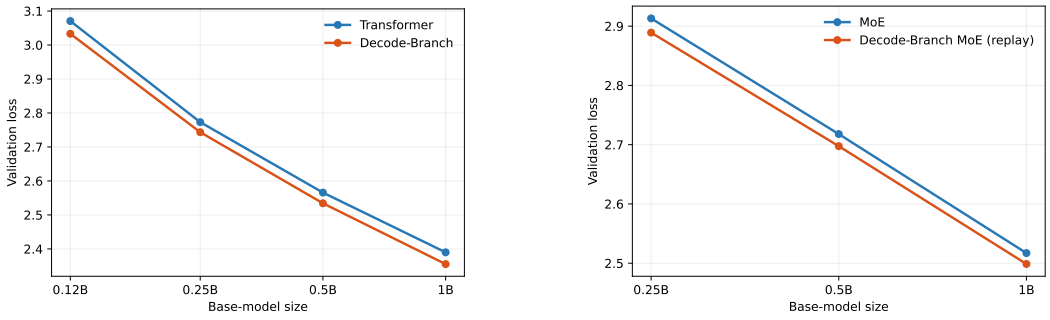
4.5.3 Dense and sparse LLaMA-style scaling.

We next scale LLaMA-style backbones with RMSNorm, rotary embeddings, gated MLPs, and multi-head attention. The dense and sparse LLaMA-style models are trained on FineWeb-Edu (Penedo et al., 2024) with sequence length 1,024 and a global batch of 512 sequences. Dense models use training data equal to 80 times the base parameter count. Sparse Qwen-style MoE models use routed and shared experts, top-4 router replay, and data equal to 40 times the base parameter count. At every tested scale from 0.12B to 1B, Decode-Branch improves over its matched baseline (Figure 4), showing that the gain is not specific to NanoGPT, dense MLPs, or one model size.

Router replay makes the branch reuse primary expert indices while retaining path-specific mixture weights. Figure 5 compares it with independent branch routing. Both variants improve over standard MoE; independent routing provides only a modest additional gain, so replay preserves most of the improvement while keeping the referenced expert set unchanged.

5 Decoupling prefill and decode expert budgets

Decode-Branch with MoE removes the symmetry that gives every stream the same expert budget. Let the primary and branch paths activate independently routed top-k_1 and top-k_2 experts. Earlier prompt positions evaluate only the primary path, whereas continuation positions evaluate both. Ignoring



(a) Dense model-size scaling.

(b) Sparse MoE model-size scaling.

Figure 4: LLaMA-style model-size scaling. Left: dense models trained on 80 times the base parameter count. Right: sparse MoE models trained on 40 times the base parameter count with router replay.

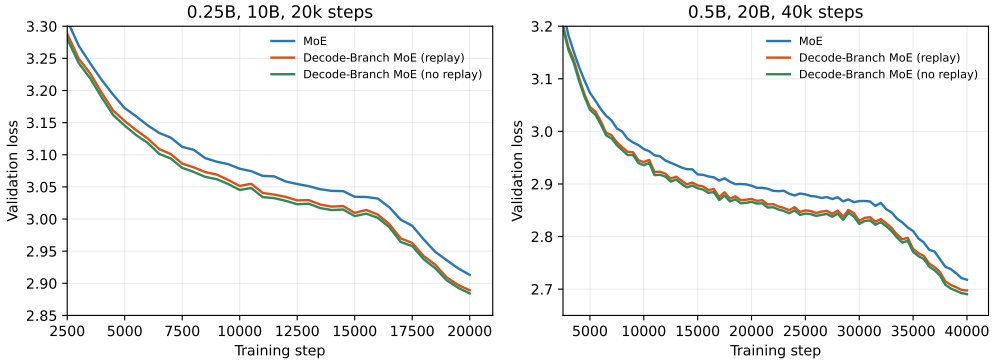


Figure 5: Sparse LLaMA-style MoE training with router replay and independent branch routing.

shared experts, attention, and dense projections, routed-expert arithmetic scales as

$$C_{\text{prefill}} \propto k_1, \qquad C_{\text{decode}} \propto k_1 + k_2. \tag{13}$$

Together with validation loss, these quantities define a prefill–decode–quality surface. We study two complementary slices using independent routing.

Unless otherwise specified, the expert-allocation experiments use the same FineWeb-Edu data, sequence length, global batch size, model configurations, training-token budgets, and optimization settings as the sparse LLaMA-style scaling experiments.

The slices reflect different deployment constraints. A prefill-constrained system can hold primary fan-out fixed and spend available decode arithmetic on the branch, as in lightly batched or fragmented generation with memory-bound headroom. A throughput-oriented system can instead fix the per-token decode budget and lower k_1, moving expert applications from every prompt position to continuation-only computation. Because training evaluates both paths at every position, this latter slice also fixes dominant routed-expert arithmetic. The slices identify independently adjustable budget components.

5.1 Fixed prefill budget: increasing decode experts

The first regime fixes $k_1 = 4$ and increases $k_2 \in \{2, 4, 8, 16\}$. Prompt expert arithmetic and the primary KV cache remain unchanged, while continuation prediction receives additional expert computation. Figure 6 shows that validation loss decreases monotonically with branch fan-out at both 0.25B and 0.5B. Thus extra continuation-side expert computation translates directly into better predictive quality when prompt computation is fixed.

5.2 Fixed decode budget: reallocating experts

The second regime fixes $K = k_1 + k_2$, so every point performs the same number of routed-expert applications per decode token. Decreasing k_1 reduces prompt arithmetic while increasing k_2 preserves

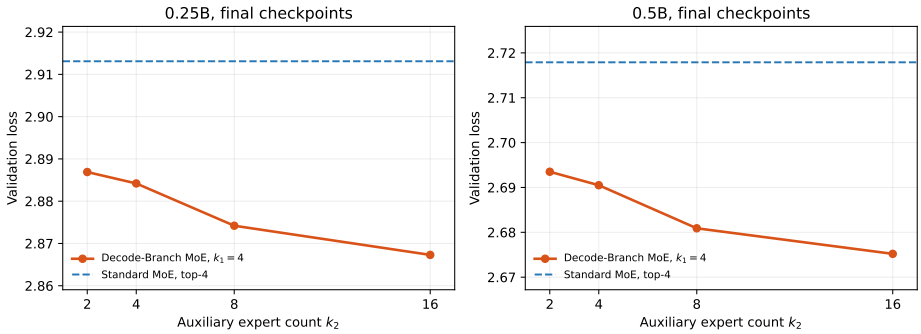


Figure 6: Fixed-prefill allocation at 0.25B and 0.5B. All Decode-Branch runs use $k_1 = 4$ with independently routed $k_2 \in \{2, 4, 8, 16\}$; standard top-4 MoE is the baseline.

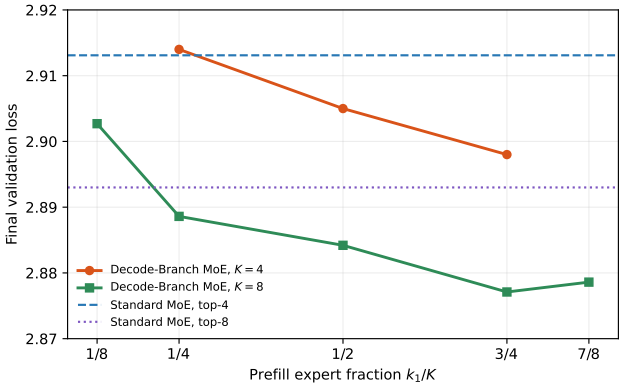


Figure 7: Fixed-decode allocation at 0.25B for $K = k_1 + k_2 \in \{4, 8\}$, plotted against the retained prompt expert fraction k_1/K. Standard top-4 and top-8 MoE are the corresponding baselines.

the decode budget. We parameterize the slice by k_1/K, exactly the fraction of baseline routed experts retained at each prompt position. Because training evaluates both paths everywhere, fixing K also holds dominant routed-expert training arithmetic constant.

Figure 7 evaluates $K \in \{4, 8\}$. The half-prefill allocation improves over standard top-K MoE in both slices. With only one quarter of baseline prompt expert arithmetic, the smaller budget remains competitive and the larger budget improves over its baseline. Quality generally improves as more experts move to the primary path because its representation also supplies the branch KV and coupling signal, but the trend need not be monotonic: if k_2 becomes too small, the branch contribution weakens and the model approaches the single-flow case. At $K = 8$, performance is therefore best near a three-quarter primary fraction and degrades slightly when nearly all computation moves to either path, showing that the optimum can be interior.

These slices match routed-expert applications rather than every arithmetic operation. They retain branch attention and shared-expert evaluation because both reuse primary KV or already referenced parameters in the intended memory-bound execution regime. Appendix C.3 gives a stricter control in which the branch receives gated primary attention and shared-expert outputs, matching those dominant evaluations as well as the routed-expert budget.

6 Conclusion

Decode-Branch decouples prefill and decode computation using a continuation trajectory that shares weights and reads primary KV without writing persistent state. Across data, dense, and sparse scaling, it improves matched-token loss; ablations isolate coupling and mixture readout. This separation creates a controllable frontier between prompt-side work, continuation capacity, and predictive quality. Independent MoE fan-outs make that frontier available for workload-specific expert allocation.

REFERENCES

Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebron, and Sumit Sanghai. GQA: Training generalized multi-query transformer models from multi-head checkpoints. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 4895–4901. Association for Computational Linguistics, 2023. doi: 10.18653/v1/2023.emnlp-main.298. URL https://aclanthology.org/2023.emnlp-main.298/.

Bradley Brown, Jordan Juravsky, Ryan Ehrlich, Ronald Clark, Quoc V. Le, Christopher Ré, and Azalia Mirhoseini. Large language monkeys: Scaling inference compute with repeated sampling, 2024. URL https://arxiv.org/abs/2407.21787.

Mouxiang Chen, Binyuan Hui, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Jianling Sun, Junyang Lin, and Zhongxin Liu. Parallel scaling law for language models. In *Advances in Neural Information Processing Systems*, 2025. URL https://arxiv.org/abs/2505.10475.

Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In *Advances in Neural Information Processing Systems*, volume 35, 2022. URL https://proceedings.neurips.cc/paper/2022/hash/67d57c32e20fd0a7a302cb81d36e40d5-Abstract-Conference.html.

William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research*, 23(120):1–39, 2022. URL https://www.jmlr.org/papers/v23/21-0998.html.

Fabian Gloeckle, Badr Youbi Idrissi, Baptiste Rozière, David Lopez-Paz, and Gabriel Synnaeve. Better & faster large language models via multi-token prediction. In *Proceedings of the 41st International Conference on Machine Learning*, 2024. URL https://proceedings.mlr.press/v235/gloeckle24a.html.

Sachin Goyal, Ziwei Ji, Ankit Singh Rawat, Aditya Krishna Menon, Sanjiv Kumar, and Vaishnavh Nagarajan. Think before you speak: Training language models with pause tokens. In *International Conference on Learning Representations*, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/76917808731dae9e6d62c2a7a6afb542-Abstract-Conference.html.

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katie Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Oriol Vinyals, Jack W. Rae, and Laurent Sifre. Training compute-optimal large language models. In *Advances in Neural Information Processing Systems*, volume 35, 2022. URL https://proceedings.neurips.cc/paper_files/paper/2022/hash/c1e2faff6f588870935f114ebe04a3e5-Abstract-Conference.html.

Keller Jordan and modded-NanoGPT contributors. Modded-NanoGPT: NanoGPT speedrun and optimization track. https://github.com/KellerJordan/modded-nanogpt, 2024. GitHub repository.

Keller Jordan, Yuchen Jin, Vlado Boza, You Jiacheng, Franz Cesista, Laker Newhouse, and Jeremy Bernstein. Muon: An optimizer for hidden layers in neural networks. https://github.com/KellerJordan/Muon, 2024. GitHub repository.

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, 2020. URL https://arxiv.org/abs/2001.08361.

Eunki Kim, Sangryul Kim, and James Thorne. Learning to insert [PAUSE] tokens for better reasoning. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 23760–23777. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.findings-acl.1217. URL https://aclanthology.org/2025.findings-acl.1217/.

Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pp. 19274–19286. PMLR, 2023. URL https://proceedings.mlr.press/v202/leviathan23a.html.

Aiwei Liu, Cheng Shi, Chuhan Wu, Ci Lei, Di Lu, Donald He, Fan Zhang, Fanhao Kong, Feifei Zhang, Guan Wang, et al. Hidden decoding at scale: Latent computation scaling for large language models, 2026. URL https://arxiv.org/abs/2607.08186.

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, 2025. URL https://arxiv.org/abs/2502.16982.

Giovanni Monea, Nathan Godey, Kianté Brantley, and Yoav Artzi. The state-prediction separation hypothesis, 2026. URL https://arxiv.org/abs/2607.01218.

Guilherme Penedo, Hynek Kydlíček, Loubna Ben Allal, Anton Lozhkov, Margaret Mitchell, Colin Raffel, Leandro von Werra, and Thomas Wolf. The FineWeb datasets: Decanting the web for the finest text data at scale, 2024. URL https://arxiv.org/abs/2406.17557.

Noam Shazeer. Fast transformer decoding: One write-head is all you need, 2019. URL https://arxiv.org/abs/1911.02150.

Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc V. Le, Geoffrey E. Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. In *International Conference on Learning Representations*, 2017. URL https://arxiv.org/abs/1701.06538.

Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling LLM test-time compute optimally can be more effective than scaling parameters for reasoning. In *International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=4FWAwZtd2n.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30, 2017. URL https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V. Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. In *International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=1PL1NIMMrw.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*, volume 35, 2022. URL https://proceedings.neurips.cc/paper_files/paper/2022/hash/9d5609613524ecf4f15af0f7b31abca4-Abstract.html.

Bohong Wu, Mengzhao Chen, Xiang Luo, Shen Yan, Qifan Yu, Fan Xia, Tianqi Zhang, Hongrui Zhan, Zheng Zhong, Xun Zhou, Siyuan Qiao, and Xingyan Bin. Parallel loop transformer for efficient test-time computation scaling, 2025a. URL https://arxiv.org/abs/2510.24824.

Bohong Wu, Shen Yan, Sijun Zhang, Jianqiao Lu, Yutao Zeng, Ya Wang, and Xun Zhou. Efficient pretraining length scaling, 2025b. URL https://arxiv.org/abs/2504.14992.

A Optimization settings

Within every comparison, the standard and Decode-Branch models use the same optimizer assignment and learning-rate schedule. Table 1 summarizes the settings used for the reported experiments.

Table 1: Optimizer settings. LR denotes peak or base learning rate. WSD uses linear warmup, a constant phase through 80% of training, and linear decay to zero.

Hyperparameter	NanoGPT	Dense LLaMA	LLaMA MoE
Parameter assignment	Muon: hidden matrices. AdamW: embeddings, output head, and Decode-Branch coupling vectors.	Muon: matrix parameters except embeddings, LM head, and norms. AdamW: all remaining parameters.	Same as dense; expert matrices use Muon.
AdamW LR	Embeddings: 0.3; head: $1/320$; coupling vectors: 0.01.	Peak 10^{-3}	Peak 10^{-3}
Muon base LR	0.02	Peak 10^{-3}	Peak 10^{-3}
Schedule	Constant for first 30%, then linear decay to zero.	Linear warmup, then cosine decay to 5×10^{-5}.	WSD; linear decay to zero over final 20%.
Warmup steps	None	400 / 800 / 1,000 / 1,000 at 0.12B / 0.25B / 0.5B / 1B	400 / 800 / 1,000 at 0.25B / 0.5B / 1B
AdamW coefficients	Embeddings/head: $\beta = (0.8, 0.95)$; coupling: $(0.9, 0.95)$; $\epsilon = 10^{-10}$; WD 0.	$\beta = (0.9, 0.95)$; WD 0.1	$\beta = (0.9, 0.95)$; WD 0.1
Muon coefficients	Momentum 0.95; WD 0.01	Momentum 0.95; WD 0.1	Momentum 0.95; WD 0.1

The NanoGPT experiments use the Muon implementation and repository-tuned learning rate from the referenced modded-NanoGPT configuration. For the dense and sparse LLaMA-style experiments, we use Kimi-style Muon (Liu et al., 2025), which rescales the scheduled base learning rate η for a matrix $W \in \mathbb{R}^{A \times B}$ as

$$\eta_W = 0.2\,\eta\sqrt{\max(A, B)}. \tag{14}$$

We tune the peak base learning rate over $\{10^{-4}, 4 \times 10^{-4}, 7 \times 10^{-4}, 10^{-3}, 2 \times 10^{-3}, 3 \times 10^{-3}\}$ and select 10^{-3} by validation loss. AdamW and Muon follow the same schedule shown in Table 1.

B Model configurations

Table 2 gives the LLaMA-style architecture presets. Model-size labels denote the parameter count of the corresponding single-path primary/base model, including token embeddings, the LM head, and, for sparse models, all routed and shared experts. A Decode-Branch model inherits the size label of its primary path; the additional branch embedding and coupling parameters are not included in that label.

Table 2: Dense and sparse LLaMA-style architecture presets. Each MoE layer contains 32 routed experts and one shared expert; routed experts have one quarter of the shared-expert intermediate width. All attention configurations use a head dimension of 64.

Family	Size	Layers	d_{model}	Query heads	KV heads	Shared/dense width	Routed experts $\times$ width
Dense	0.12B	6	768	12	12	2,048	–
Dense	0.25B	12	1,024	16	16	2,730	–
Dense	0.5B	18	1,280	20	20	3,413	–
Dense	1B	22	1,792	28	28	4,778	–
MoE	0.25B	6	640	10	10	1,706	32×426
MoE	0.5B	9	768	12	12	2,048	32×512
MoE	1B	12	1,024	16	16	2,730	32×682

C Additional training curves

C.1 NanoGPT component trajectories

Figure 8 complements the endpoint comparison in Figure 3 with the validation-loss trajectories at $D = 3$. It compares the Transformer, PHD-2, the minimal Decode-Branch construction without primary-to-branch coupling, whose next-token prediction uses only the branch distribution, and the complete Decode-Branch model.

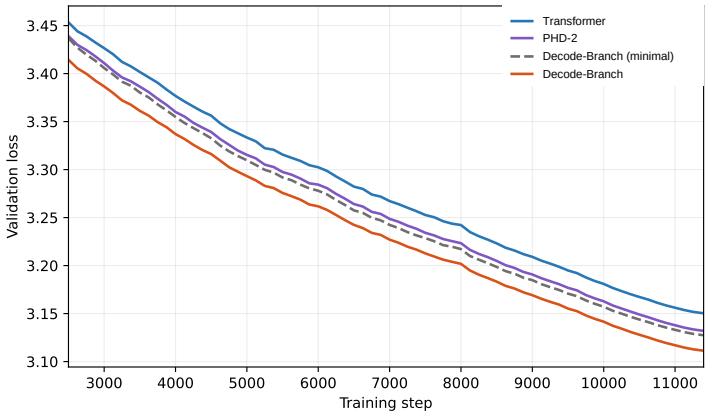


Figure 8: NanoGPT validation loss versus training step at $D = 3$. Curves begin at step 2,500 to emphasize the separation among the Transformer, PHD-2, minimal Decode-Branch without coupling and with a branch-only readout, and complete Decode-Branch constructions.

C.2 Fixed-prefill MoE trajectories

The main text summarizes the fixed-prefill sweep using validation loss as a function of the branch expert count. Figure 9 provides the corresponding training trajectories at both tested model scales. Its legend reports the number of experts activated by the decode branch.

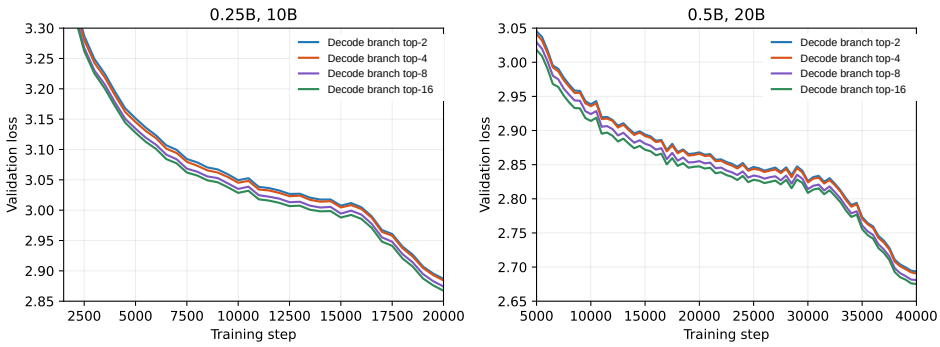


Figure 9: Validation loss during the completed fixed-prefill MoE sweeps. All runs use primary top-4 routing and independently routed decode-branch top-$2/4/8/16$.

C.3 Compute-matched MoE allocation

The fixed-decode experiment in Section 5 matches the total number of routed-expert applications, but its default implementation evaluates attention and the shared expert separately for both paths. This is motivated by decode execution: attention over the KV cache and the shared expert are expected to be limited primarily by memory traffic, and grouped execution can reuse their weights and cached state. We nevertheless construct a stricter control that removes their decode-branch arithmetic.

Let N_1^ℓ and N_2^ℓ be the normalized primary and branch states at layer ℓ. In the default Decode-Branch MoE, the primary attention output is computed from primary queries, keys, and values, while the decode branch forms its own queries and reads the primary KV:

$$A_1^\ell = \mathrm{Attn}(Q_1^\ell, K_1^\ell, V_1^\ell)W_O^\ell, \qquad A_2^\ell = \mathrm{Attn}(Q_2^\ell, K_1^\ell, V_1^\ell)W_O^\ell. \tag{15}$$

For the compute-matched variant, we replace the second attention evaluation by a learned vector-gated copy, $A_2^\ell = g_A^\ell \odot A_1^\ell$, where $g_A^\ell \in \mathbb{R}^d$ is initialized to one. Thus both paths receive an attention update, but attention is evaluated only once.

We apply the same construction to the shared expert. Let $G^\ell(\cdot)$ denote the shared-expert network and let $R_i^\ell(N_i^\ell; k_i)$ denote the sum of the k_i independently routed expert outputs for flow i. The primary output is $M_1^\ell = R_1^\ell(N_1^\ell; k_1) + G^\ell(N_1^\ell)$, while the branch output is

$$M_2^\ell = R_2^\ell(N_2^\ell; k_2) + g_G^\ell \odot G^\ell(N_1^\ell), \tag{16}$$

where $g_G^\ell \in \mathbb{R}^d$ is another learned vector initialized to one. Equation 16 evaluates the shared expert only on the primary state; the auxiliary retains its own routed computation and receives a gated primary shared-expert output.

We use the 0.25B architecture and impose

$$k_1 + k_2 = K = 4. \tag{17}$$

Consequently, for every layer and decode token, the standard top-K MoE and this Decode-Branch variant both perform exactly one attention evaluation, one shared-expert evaluation, and K routed-expert applications:

$$C_{\mathrm{decode}}^{\mathrm{DB}} = C_{\mathrm{attn}} + C_{\mathrm{shared}} + (k_1 + k_2)C_{\mathrm{expert}} = C_{\mathrm{decode}}^{\mathrm{MoE}}. \tag{18}$$

The remaining branch-specific operations are lightweight elementwise gates and residual-state updates. During prefill, the decode branch is omitted, giving

$$\frac{C_{\mathrm{prefill,routed}}^{\mathrm{DB}}}{C_{\mathrm{prefill,routed}}^{\mathrm{MoE}}} = \frac{k_1}{K}. \tag{19}$$

The horizontal axis in Figure 10 is therefore exactly the retained fraction of baseline routed-expert prefill computation.

Figure 10 shows a clear quality–prefill trade-off under this stricter accounting: validation loss improves as a larger fraction of the expert budget is assigned to the primary path. Notably, the half-prefill allocation matches the standard top-4 MoE baseline while using 50% fewer routed-expert applications during prompt processing and no additional dominant backbone computation during decode.

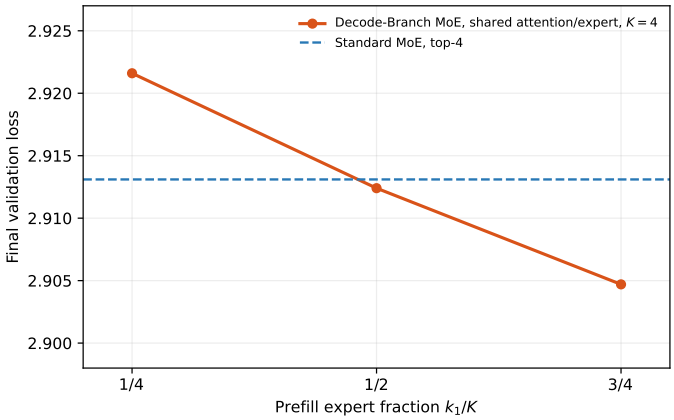


Figure 10: Strictly compute-matched allocation at 0.25B with $K = k_1 + k_2 = 4$. Decode-Branch MoE evaluates attention and the shared expert only on the primary path and sends their outputs to the decode branch through learned gates. All points match the standard top-4 MoE in attention, shared-expert, and routed-expert evaluations per layer; k_1/K is the fraction of baseline routed-expert computation retained during prefill.

Table 3: Final validation loss for one and two decode branches at matched NanoGPT training-token budgets.

Model	1×	2×	3×	4×	5×
One decode branch	3.24035	3.15306	3.11145	3.08609	3.06789
Two decode branches	**3.22114**	**3.13304**	**3.09244**	**3.06690**	**3.04940**

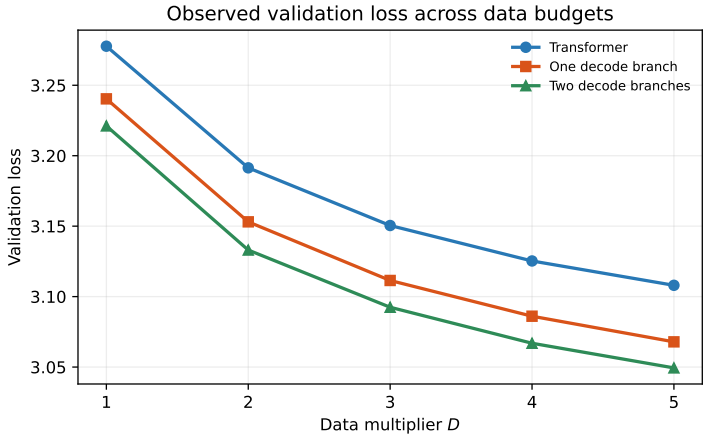


Figure 11: Observed validation loss for the one- and two-branch constructions across matched training-token budgets.

D Other extensions of phase-decoupled computation

The main text focuses on expert-budget allocation because it directly exposes the prefill–decode trade-off in contemporary MoE models. We additionally explore two extensions for adding continuation-side capacity: another decode branch and dense weights specific to the decode branch. Both preserve the primary prompt path and persistent KV cache while expanding the phase-decoupled design space.

D.1 More decode branches

For $K > 2$, each decode branch $j \in \{2, \dots, K\}$ has an independent embedding and same-position residual state but reads the same primary KV cache. We average the branch distributions,

$$\bar{q}_t(v) = \frac{1}{K-1} \sum_{j=2}^{K} q_{j,t}(v), \tag{20}$$

and substitute $\bar{q}$ for q in Equation 8. This keeps the prompt-wide primary computation unchanged while increasing continuation arithmetic and temporary state approximately with K. With independent embeddings, however, each added branch also introduces another vocabulary-scale table and additional training computation.

We instantiate two decode branches, for three residual paths in total, with coupling parameters shared across the branches. Table 3 and Figure 11 show lower validation loss than the standard one-branch Decode-Branch Transformer at all five NanoGPT budgets, demonstrating that the shared-weight construction extends naturally to multiple decode branches.

D.2 Auxiliary-specific dense weights

The decode branch need not share all primary-path parameters. We give it separate query and attention-output projections and a separate MLP while retaining primary keys and values. These matrices are omitted at earlier prompt positions and evaluated at the continuation boundary and subsequent decode positions. This preserves the primary prefill graph and persistent cache, but increases deployed parameters and the unique continuation-step weight footprint.

Figure 12 shows that this variant lowers validation loss relative to shared-weight Decode-Branch throughout the measured NanoGPT range, demonstrating that branch-specific capacity provides another effective axis for continuation-side scaling.

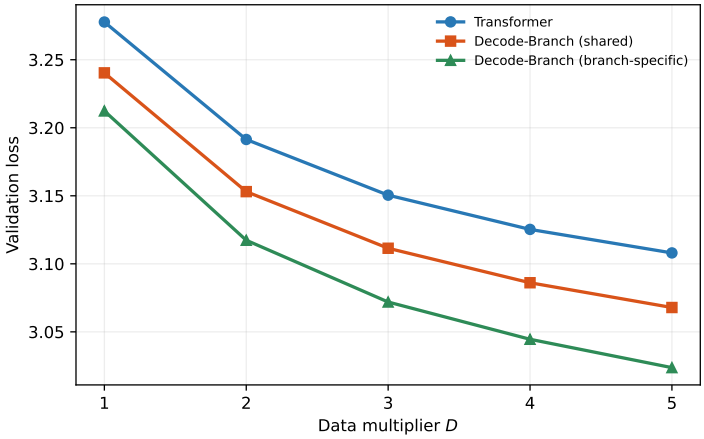


Figure 12: Validation loss with shared versus branch-specific dense weights across matched NanoGPT training-token budgets.