PIPELINED TRANSFORMERS: AUTOREGRESSIVE GENERATION WITH INCREASED PARALLELISM

Anonymous authors
Paper under double-blind review

ABSTRACT

Transformers are slow at generating text. Each token must be *fully* generated *before* work starts on the next token. Diffusion models break this serial decoding bottleneck, but they are non-autoregressive latent-variable models. We propose PIPET , a *pipelined* architecture that remains autoregressive and simple to train. Tokens are decoded in parallel while attending to one another bidirectionally, as in diffusion, but token $i+1$ is slightly delayed with respect to token i, so it can depend autoregressively on token i and all previous tokens. This architecture decodes faster than a standard Transformer, reduces KV cache size, and is formally more expressive. Although pretraining and prompt prefilling are slower than in a standard Transformer, this tradeoff may be worth it when generation is a bottleneck, as when generating code or long reasoning traces during RL training or after deployment. We evaluate models with 4M–200M parameters trained on the SlimPajama-6B subset using Chinchilla-style token budgets. At 100M parameters, a 3-layer PIPET matches the test loss of a 4-layer standard Transformer of the same size while decoding $1.8\times$ faster with a $3.1\times$ smaller KV cache. It nearly matches a 16-layer standard Transformer (the optimal number of layers that we evaluated at this parameter budget), with only 0.035 nats/token higher test loss while decoding $4.4\times$ faster with a $6.3\times$ smaller KV cache.

1 INTRODUCTION

Long generations are increasingly central to language-model use. Reasoning models chatter to themselves, coding agents revise long programs, and reinforcement learning relies on repeated rollouts. Decoding speed therefore affects both training speed and runtime response speed.

Standard Transformers achieve little within-sequence parallelism during decoding. During training and prompt encoding, they can handle all positions in parallel (Vaswani et al., 2017). Unfortunately, autoregressive decoding handles only one position at a time. Each position must traverse all L layers before computation begins for the next.

In contrast, diffusion language models decode all output tokens in parallel: at each step, all tokens refine themselves through bidirectional attention to other tokens (Li et al., 2022; Lou et al., 2024; Nie et al., 2025). However, often so many steps are needed that the models may be slower overall (Feng et al., 2025). Diffusion models are also more difficult to train, requiring a noising process and a denoising schedule. Finally, diffusion models are latent-variable models that do not compute deterministic encodings (needed for text embedding), next-token probabilities (needed for distillation training), or whole-sequence probabilities (needed for evaluating given strings and for many post-training methods such as PPO and DPO).

This raises a question: *can an* autoregressive *model parallelize computation across positions?*

We propose PIPET, a pipelined autoregressive architecture. On a modern pipelined CPU, different hardware stages execute different phases of an instruction, such as reading operands, computing a result, and writing it back. The next instruction can therefore begin before the current one finishes. Applying this idea to autoregressive decoding, however, presents a challenge: unlike the next instruction, the next token is not known in advance.

PIPET addresses this by moving token input and next-token prediction to intermediate layers. A stream begins before its token is available, receives it later, predicts the next token, and then contin-

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

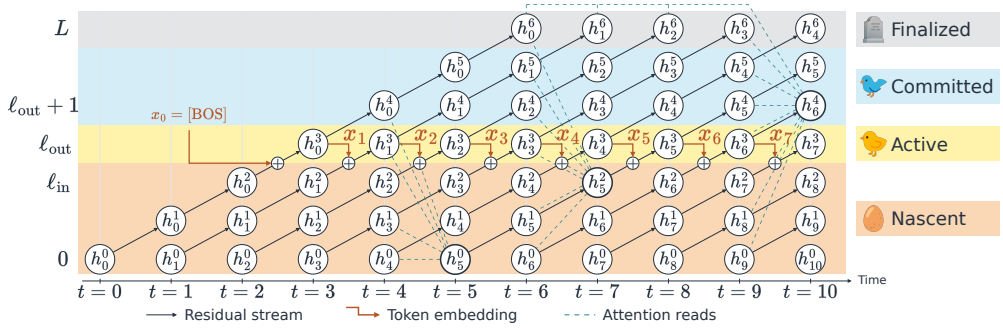


Figure 1: At timestep t (column), PIPET updates hidden vectors at different layers (superscripts) and positions (subscripts) in parallel. Diagonal black arrows trace each residual stream. A stream at layer ℓ_{out} predicts the next token, whose embedding enters the corresponding stream at layer ℓ_{in} (orange arrow). Each vector attends to the highest available state of each extant stream. The attention reads for h_5^0, h_5^2, and h_6^4 are shown with dashed blue lines. For example, h_6^4 attends to finalized tokens 0–3 (at layer $L = 6$), committed tokens 4–5 (at layers 5 and 4), active token 6 (at layer 3), and nascent *future* tokens 7–9 (at layers $\leq \ell_{\text{in}}$ that are still ignorant of the token identity). Colors mark the four stream stages. The example uses $(L, \ell_{\text{in}}, \ell_{\text{out}}) = (6, 2, 3)$; Figure 4 gives the analogous graph for standard transformer, which is L times sparser at each computation timestep.

ues refining its own representation. Token $i+1$ is still predicted only after token i becomes available, preserving the autoregressive factorization and ordinary teacher-forced next-token training.

This change allows PIPET to advance multiple streams concurrently, each at a different position and depth. Figure 1 illustrates the scheme. Notice that PIPET adopts **mixed-layer attention** among the tokens' **residual streams**. Each attention operator attends to the highest available state of every existing stream—earlier positions contribute higher-layer states, whereas later positions contribute lower-layer states that have not yet received their token embeddings. Multiple queries can therefore reuse one loaded KV history, and finalized streams retain only one terminal KV pair.

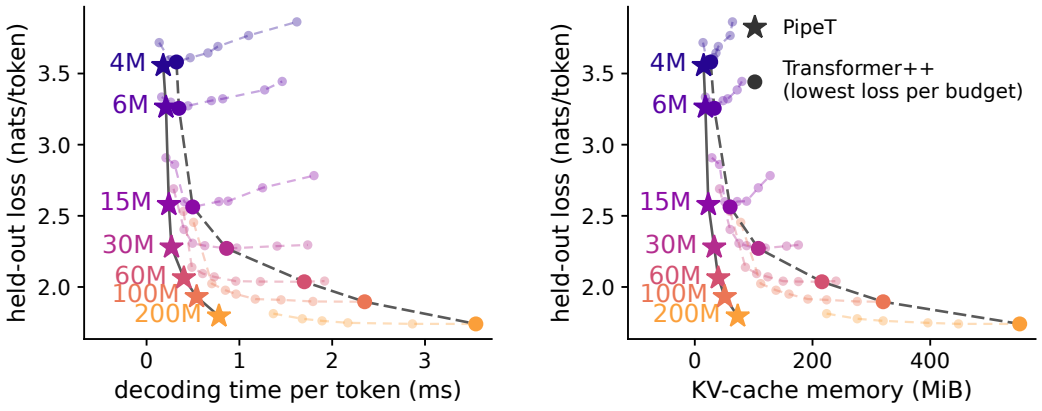
Across 4M–200M parameters, PIPET offers competitive test loss with substantially faster decoding and smaller KV caches, at the cost of slower training. §2.4 explains these architectural consequences and their relationship to a standard Transformer in more detail. At 100M, it matches a 4-layer Transformer's loss while decoding $1.8\times$ faster with a $3.1\times$ smaller cache; relative to the lowest-loss evaluated Transformer, it trades 0.035 nats/token for a $4.4\times$ speedup and a $6.3\times$ smaller cache.

Our contributions are:

- We introduce a pipelined architecture that overlaps computation across positions while preserving autoregressive next-token modeling.
- We formalize the pipelined computation and analyze its sequential depth, total work, and memory requirements, together with the representational capabilities enabled by recurrence under explicit theoretical assumptions.
- We evaluate quality–runtime and quality–memory tradeoffs and sweep different depth, width, and input–output layer placement, yielding a heuristic allocation rule across parameter budgets.

2 METHOD

In a standard L-layer autoregressive Transformer, computation for position i advances a residual stream from the embedding of x_i to a prediction of x_{i+1}. We define the **input layer**, ℓ_{in}, as the layer that receives the token embedding, and the **output layer**, ℓ_{out}, as the layer that predicts the next token. A standard Transformer fixes $\ell_{\text{in}} = 0$ and $\ell_{\text{out}} = L$. To allow computation across streams to overlap, PIPET permits $0 \leq \ell_{\text{in}} < \ell_{\text{out}} \leq L$, so that a stream can begin computation before receiving its token and continue after predicting the next one. A stream's lifecycle has four stages:



(a) Runtime per token

(b) KV-cache memory

Figure 2: **Quality–efficiency tradeoffs at matched parameter and training-token budgets.** Color denotes parameter budget; stars are recipe-selected PIPET configurations. Faded circles show Transformer depth–width sweeps, with the lowest-loss configuration at each budget enlarged and connected. **(a)** Runtime per decoded token. **(b)** Retained KV-cache memory for 512 decoding steps. Lower and further left are better; §E.2 gives measurement details.

1. 🥚 **Nascent.** Stream i is **initialized** at layer 0 to a learned vector shared across positions.
2. 🐤 **Active.** Once x_i becomes available, its embedding is **injected** into stream i by adding it to the layer-ℓ_{in} representation. The stream then advances to ℓ_{out} to predict x_{i+1}.
3. 🐦 **Committed.** After **predicting** x_{i+1} at ℓ_{out}, stream i continues through the remaining layers, enriching the representation available to subsequent predictions.
4. 🪦 **Finalized.** At the **terminal layer**, L, the stream stops evolving and supplies a persistent key–value pair for future attention.

The nascent computation is absent when $\ell_{\text{in}} = 0$, and commitment coincides with finalization when $\ell_{\text{out}} = L$. In a standard Transformer, a stream is active from layer 0 and becomes committed and finalized at layer L. The background colors in Figure 1 illustrate the correspondence between layers and stages for a fully pipelined PIPET configuration with $L = 6$, $\ell_{\text{in}} = 2$, and $\ell_{\text{out}} = 3$.

2.1 Notation and terminology

For a sequence $x_{1:I}$ and a fixed beginning-of-sequence symbol $x_0 = [\text{BOS}]$, we model

$$p_\theta(x_{1:I} \mid x_0) = \prod_{i=1}^{I} p_\theta(x_i \mid x_{<i}). \tag{1}$$

We index token and stream positions by i, computation timesteps by t, and layers by $\ell \in \{0, \dots, L\}$. The state of stream i at layer ℓ is h_i^ℓ. We write Emb for the token embedding map, Unemb for the **language-modeling head** mapping hidden states to vocabulary logits, and h_0 for the learned initialization, so that a new stream starts at $h_i^0 = h_0$. Token injection adds $\text{Emb}(x_i)$ to $h_i^{\ell_{\text{in}}}$ in place.

The **layer operator** LayerUpdate_ℓ advances a stream from layer $\ell - 1$ to layer ℓ. For PIPET, it applies pre-LayerNorm multi-head softmax attention and a residual update, followed by a pre-LayerNorm GEGLU feed-forward network and a second residual update. Its named arguments specify the current state used to form the query and the collection of source streams that supplies the key–value context. The layer operator applies the appropriate key–value map to each source according to its current layer. Let KV_ℓ denote this map for a source state at layer ℓ, including a terminal map KV_L. §E.1 describes the corresponding normalization and feed-forward choices for the standard Transformer baseline. A standard causal Transformer computes

$$h_i^\ell = \text{LayerUpdate}_\ell\big(\text{query} = h_i^{\ell-1},\ \text{KVContext} = \{h_j^{\ell-1}\}_{j=0}^{i}\big) \tag{2}$$

so that attention reads the current and preceding positions at the same layer. PIPET changes which positions and layers provide these keys and values.

A **schedule** specifies when streams get created, advance through layers, receive tokens, and make predictions. It must respect layer order and token availability: an update reads only previously computed states, and a token is injected only after its predictive distribution has been computed, with [BOS] supplied as the fixed initial token.

2.2 PIPET

We describe the fully pipelined schedule used in our main experiments: $\ell_{\text{out}} = \ell_{\text{in}} + 1$, which allows every unfinished stream to advance one layer per timestep. After startup, each timestep produces one token. §A generalizes the schedule to allow larger $\ell_{\text{out}} - \ell_{\text{in}}$, so that the computation that advances from x_i to x_{i+1} is deeper but takes more than one timestep.

2.2.1 MIXED-LAYER ATTENTION

Mixed-layer attention reads the highest currently available representation of every existing stream. All updates in a timestep read the same snapshot, formed after the injection of the newly decoded toke. Let $\mathcal{V}$ be the set of existing streams and ℓ_i the highest available layer of stream i at the snapshot. An update computes

$$\begin{aligned} \ell_i &= \max\{\ell : h_i^\ell \text{ is available}\}, \\ h_i^\ell &= \text{LayerUpdate}_\ell\Big(\text{query} = h_i^{\ell-1},\ \text{KVContext} = \{h_j^{\ell_j}\}_{j\in\mathcal{V}}\Big). \end{aligned} \tag{3}$$

Each stream contributes one key–value pair. For a finalized stream, $\ell_i = L$ and its terminal pair is computed once and retained in a shared KV cache. Queries and keys use stream positions for rotary position encoding.

When stream i advances from layer ℓ to layer $\ell + 1$, earlier streams provide states at layers above ℓ, up to the terminal layer, and later streams provide states below ℓ. The current stream contributes its state at ℓ. Because the snapshot is fixed, updates to different streams can run in parallel against the same KV context, while each uses the query projection and other parameters of its own layer operator. Attention to later positions does not expose future tokens: when predicting x_{i+1}, all states in the snapshot depend only on $x_{\le i}$. Later streams exist, but have not yet received their tokens.

2.2.2 DECODING

Algorithm 1 gives the complete decoding procedure, including startup. A single loop advances every stream below layer L. During the initial iterations, no stream has yet reached the output layer; BOS is injected as soon as its stream reaches the input layer. Thereafter, reaching the output layer triggers prediction of the next token, which is immediately injected into its stream before the next attention update:

$$\begin{aligned} p_\theta(\cdot \mid x_{\le i}) &= \text{softmax}\Big(\text{Unemb}(h_t^{\ell_{\text{out}}})\Big), \\ h_{i+1}^{\ell_{\text{in}}} &\leftarrow h_{i+1}^{\ell_{\text{in}}} + \text{Emb}(x_{i+1}). \end{aligned} \tag{4}$$

Streams that have reached layer L remain in the collection as attention sources but receive no further updates. When $\ell_{\text{in}} = 0$, startup is empty and the initial stream immediately receives BOS. §A.5 gives the unrolled computation, while §A discusses alternative stream creation and stalling schedules.

2.3 TRAINING

We train with the ordinary next-token negative log-likelihood,

$$\mathcal{L}(\theta) = -\mathbb{E}_{x_{1:I}\sim\mathcal{D}}\left[\sum_{i=1}^{I} \log p_\theta(x_i \mid x_{<i})\right], \tag{5}$$

where $\mathcal{D}$ is the training distribution over length-I sequences. The token log-probabilities here are computed by line 11 of Algorithm 1; during training, teacher forcing supplies the observed tokens $x_{1:I}$ in place of samples. §E.1 provides more details about hyperparameters.

Algorithm 1 PIPET decoding with $\ell_{\text{out}} = \ell_{\text{in}} + 1$.

Require: $0 \le \ell_{\text{in}} < L$; model parameters; BOS and EOS tokens
Ensure: A decoded sequence ending in EOS and all token distributions

// The residual stream s at token position $s.i \ge 0$ stores hidden state $s.h \in \mathbb{R}^d$ for its most recently computed layer $s.\ell \in \{0, \dots, L\}$.

$Streams \leftarrow \{(0, 0, h_0)\}$
$x_0 \leftarrow \text{BOS}$; $i \leftarrow 0$ ▷ i tracks the current output token position
while true **do**
  // Inject BOS as soon as its stream reaches the input layer.
  if $i = 0$ and a stream u is at layer ℓ_{in} **then**
    $u.h \leftarrow u.h + \text{Emb}(x_0)$ ▷ Inject BOS
  // Advance unfinished streams together. Layer L remains available doesn't update.
  for all $s \in Streams$ with $s.\ell < L$, in parallel **do**
    $s.h \leftarrow \text{LayerUpdate}_{s.\ell+1}(\text{query} = s.h,\ \text{KVContext} = Streams)$
    $s.\ell \leftarrow s.\ell + 1$
  // Create the next stream.
  Append $(s.i, s.\ell, s.h) = (\text{next position}, 0, h_0)$ to $Streams$
  // Predict and inject when a stream reaches the output layer.
  if a new stream c has just reached layer ℓ_{out} **then**
    $p_t \leftarrow \text{softmax}(\text{Unemb}(c.h))$ ▷ Read from the output layer
    $x_{i+1} \sim p_i$ ▷ Use the observed token under teacher forcing
    if $x_{i+1} = \text{EOS}$ **then**
      return $x_{1:i+1}, p_{0:i}$ ▷ Stop after decoding EOS
    $u \leftarrow$ the stream in $Streams$ at layer ℓ_{in}
    $u.h \leftarrow u.h + \text{Emb}(x_{i+1})$ ▷ Inject before the next attention update
    $i \leftarrow i + 1$

2.4 Discussion: Differences from a standard Transformer decoder

PIPET has higher throughput due to greater parallelism. It updates L residual streams in parallel at each timestep, always generating a token (from the stream that reaches layer ℓ_{out} at this timestep). This contrasts with the standard Transformer, which updates only the single newest residual stream and thus only generates a token once per L timesteps (when that stream reaches layer L). The increased parallelism gives an L-fold increase in arithmetic intensity—the number of floating-point operations performed per byte transferred from memory.

PIPET has a more compact KV cache. Each stream makes only its most recent KV pair available for attention by other streams. The pairs from lower layers need not be stored, in contrast to a standard Transformer at decoding time. For L-layer architectures of a given width d, PIPET therefore has a smaller KV cache, by a factor of L. This relieves pressure on GPU memory (Kwon et al., 2023), allowing the working memory of a given GPU[1] to handle prompt+response sequences that are L times longer, or to serve L times as many requests in parallel.

PIPET plans for tokens before receiving them at layer ℓ_{in}. In a standard Transformer, a stream cannot begin computation until its token is available. In PIPET, when $\ell_{\text{in}} > 0$, the nascent layers at the start of the stream can begin planning before receiving that token. As the stream advances toward ℓ_{in}, it attends to representations that incorporate increasingly recent tokens, allowing it to refine this plan. We hypothesize that the computation schedule encourages this behavior. In our main configuration, the model has only one layer update to incorporate x_i before predicting x_{i+1}. This creates an incentive to prepare a representation useful for predicting x_{i+1} before x_i is available, so that the prediction can be formed quickly once this last conditioning token is sampled and injected.

PIPET refines a token's representation after layer ℓ_{out}, using later tokens. The architectures also differ at the end of a stream. A standard causal Transformer encodes x_i using only $x_{\le i}$. In PIPET, stream i continues evolving after predicting x_{i+1} at layer ℓ_{out}. Its committed layers can

[1] That is, the part of the memory not reserved for storing model parameters.

therefore attend to representations containing later tokens as those tokens become available, in the spirit of bidirectional encoding. For example, in the English sentences "I see that point" or "I consent to dine," the third word's part of speech can be disambiguated once the fourth word arrives (cf. "I see that you win," "I consent to dinner"). The resulting refined representation of the third word will be computed in time for the sixth word to attend to it. This does not violate autoregressive conditioning: the tokens $x_{>i}$ do not affect layer ℓ_{out} and hence do not affect the distribution over x_{i+1}.

PIPET are more powerful thanks to within-layer recurrent connections. Standard Transformers are *fixed-depth* arithmetic circuits, which limits the properties that they can extract from the context (Merrill & Sabharwal, 2023; Chiang, 2025). This is also true of state-space models (Merrill et al., 2024). In contrast, PIPET's computation graph deepens with the sequence length, just as an RNN's does. Hidden vectors at a given layer can pass information forward to later hidden vectors at the *same* layer (e.g., at layer 3 of Figure 1, h_5^3 attends to h_4^3). This allows the layer to track the state of the sequence as new tokens are appended. Of course, other recent models have also combined within-layer recurrences with attention (e.g., De et al., 2024), as did the earliest attention work in NLP (Bahdanau et al., 2016).

PIPET sacrifices parallelism during training. During training and prompt prefilling (encoding), all tokens are provided at once. Here, standard Transformers can process all I *positions* within each layer in parallel. The within-layer recurrent connections in PIPET prevent this; it can only process the L *layers* in parallel, where usually $L \ll I$. The increased parallelism during decoding therefore comes at the cost of reduced parallelism during training and prefilling. Possibly hybrid approaches are discussed briefly in §5. Without such approaches, PIPET is most attractive when decoding dominates the runtime, as when producing long outputs (including reasoning traces, rapid action sequences in an environment, or dialogue turns that need not be re-encoded on future turns because they were cached), or when repeatedly sampling rollouts during RL training.

3 EXPERIMENTS

We evaluate prediction quality, runtime per token, and KV-cache memory under matched parameter and training-token budgets. At each parameter budget, we compare a single PIPET configuration, selected using a scale-dependent architecture recipe derived from validation-loss sweeps (§3.4), against a family of standard Transformers spanning different depth–width configurations. We find that across scales, PIPET reorganizes computation to provide a more favorable tradeoff between predictive quality (log-loss) and inference efficiency.

We additionally evaluate PIPET on a synthetic problem to test whether PIPET's within-layer connections improve its ability to track the state of a certain deterministic finite-state automaton beyond the sequence lengths observed during training (§3.3).

3.1 EXPERIMENTAL SETUP

Models and budgets. Parameter budgets are $N \in \{4, 6, 15, 30, 60, 100, 200\}$ million, with ≈ 20 training tokens per parameter following Hoffmann et al. (2022). §E.1 gives complete architecture and training details.

Data and training. We use a 6B-subset of the SlimPajama Shen et al. (2024) dataset with fixed training, validation, and test splits, a 1,000-token BPE vocabulary, and length-512 sequences. All configurations use AdamW, an effective batch size of 64, and seed 41. We report test loss in nats per scored token; the validation split is used for monitoring, architecture search, and every configuration choice reported below.

Timing protocol. We time the production of 512 output distributions per sequence under teacher forcing on a single NVIDIA H100 NVL GPU allocation on a shared node, using fp32 with TF32 disabled and batch size 8. This executes the same model computation as decoding but omits the sampling step; the timed region includes PIPET's pipeline startup. Both model families use compiled, CUDA-graph-captured single-step decoders. After two warm-up passes, we report the median of five runs in fresh processes. §E.2 provides further details on timing and KV-cache accounting.

3.2 PIPET ACHIEVES BETTER QUALITY–EFFICIENCY TRADEOFFS

At each parameter budget, we compare the selected PIPET configuration against a depth–width sweep of standard Transformers trained under the same parameter and token budgets. The PIPET configurations are selected by the architecture recipe summarized in §3.4. Figure 2 summarizes the resulting quality–runtime–memory tradeoffs.

Across all parameter budgets, the efficiency gains of PIPET come with only modest changes in predictive quality. At 4M parameters, PIPET achieves lower test loss than all evaluated Transformers, while at 6M it achieves similar loss. From 15M through 200M, PIPET incurs 0.009–0.055 nats/token higher loss than the lowest-loss Transformer at each budget while substantially reducing runtime per token and KV-cache memory.

These differences become most pronounced at larger scales. At 100M parameters, relative to the lowest-loss 16-layer Transformer, PIPET is $4.36\times$ faster per token and uses a $6.3\times$ smaller KV cache, at a cost of 0.035 nats/token higher test loss. At 200M parameters, relative to the lowest-loss 24-layer Transformer, PIPET is $4.55\times$ faster and uses a $7.58\times$ smaller KV cache, at a cost of 0.055 nats/token higher loss.

We also compare the two model families under a common-loss constraint. At each parameter budget, we select the fastest Transformer whose test loss is no higher than that of PIPET. Relative to these configurations, PIPET is 1.7–$3.2\times$ faster across 6M–200M parameters. At 100M and 200M, for example, PIPET is respectively $2.17\times$ and $2.15\times$ faster than the corresponding 6-layer Transformers while achieving no higher test loss.

§E.2 and §E.4 report the individual configurations and complete measurement protocol.

3.3 PIPET CAN LEARN TO DO STATE TRACKING

We empirically evaluate whether the recurrent computation induced by pipelining improves PIPET's state-tracking capability relative to a standard Transformer. We consider $\mathrm{W}(S_k)$, the word problem for the symmetric group S_k, which for $k = 5$ is believed to be beyond the formal power of a standard Transformer with limited numerical precision (Merrill & Sabharwal, 2023), but is obviously within the power of an RNN since it simply tracks the state in a certain DFA (Strobl et al., 2024). The model receives a sequence of permutation functions on $\{1, 2, \ldots, k\}$ and is trained to predict their running composition at every position.[2] Figure 3 compares PIPET and Transformer++ for $k = 3$ and $k = 5$, including sequence lengths beyond those seen during training. PIPET achieves higher permutation-state accuracy and substantially stronger length generalization, with the clearest separation on $\mathcal{S}_5$. Both architectures perform better on tracking whether the running composition is an even or odd permutation, which is essentially equivalent to the $k = 2$ problem. See §C for further details.

3.4 ARCHITECTURE SELECTION ACROSS SCALE

To select one PIPET configuration at each parameter budget for the natural language experiments, we use a staged validation-loss search over depth, width, and input–output layer placement. The sweeps consistently favor shallow, wide configurations, with preferred depth increasing only gradually with model scale. We provide a provisional scaling recipe and use it to select the specific architecture for the main experiment §3.2. The recipe is reported in §E.3.

3.5 ADDITIONAL EXPERIMENTS

We perform several additional experiments to characterize PIPET beyond the configurations used in the main comparison.

Trading parallelism for sequential refinement. §A generalizes the PIPET architecture to allow $\tau = \ell_{\mathrm{out}} - \ell_{\mathrm{in}}$ to be greater than 1. This provides an additional axis for trading parallelism against sequential refinement. At 30M parameters and fixed depth $L = 6$, partially stalled configurations

[2] We evaluate the log-loss of these predictions. The formal problem is actually a decision problem—to determine whether the current composition is the identity permutation.

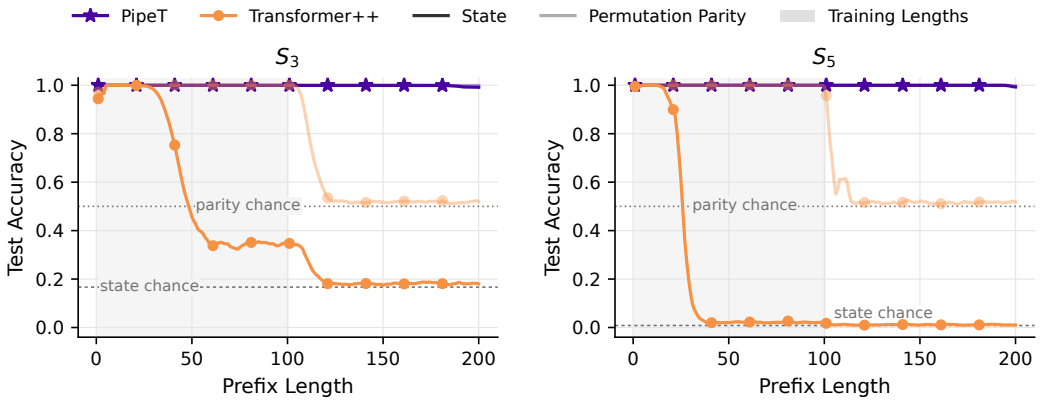


Figure 3: **State tracking and length generalization.** Test accuracy on permutation-group word problems over S_3 and S_5. The shaded region denotes sequence lengths seen during training. Solid curves show full permutation-state prediction; faded curves show the corresponding parity task. Dotted lines denote chance accuracy, $1/n!$ for state prediction and $1/2$ for parity. We report the best result over five seeds after a learning-rate sweep.

with $\tau \in \{3, 4, 5\}$ outperform the matched 6-layer Transformer—the lowest-loss Transformer evaluated at this budget—in test loss, runtime per token, and KV-cache memory. Relative to the fully pipelined $\tau = 1$ configuration used in the main comparison, increasing τ improves predictive quality while giving up some runtime and memory savings. Further increases in τ continue to reduce loss, but with diminishing improvements and increasing sequential cost (§A.4 and §E.4).

Scaling along the architecture recipe. Loss along the selected architecture recipe follows a smooth scaling trend. A power-law fit to the recipe-selected models from 4M through 100M parameters predicts the withheld 200M result to within 0.0003 nats/token (§F.1). We also sweep alternative training-token budgets across model scales. Over the evaluated range, the compute-optimal token-parameter ratio for PIPET exceeds 20 and increases with compute budget (§F.2).

4 Related Work

PIPET connects several lines of work. Recurrent and feedback Transformers reuse representations across depth or positions (Fan et al., 2020; Ju et al., 2021; Dehghani et al., 2018). Pipelined, speculative, and adaptive-depth decoding methods overlap or shorten the work between successive tokens (Leviathan et al., 2022; Hooper et al., 2023; Yang et al., 2024a; Schuster et al., 2022). Diffusion language models refine multiple positions in parallel (Sahoo et al., 2024; Nie et al., 2025). Various attention, recurrent, and hybrid architectures have been designed to reduce decoding cost (Shazeer, 2019; Brandon et al., 2024; Gu & Dao, 2024; Ren et al., 2025). More details on each of these in §D.

5 Discussion, Limitations, and Future Work

Beyond the architectural consequences discussed in §2.4, our architecture sweep finds competitive performance with shallow, wide configurations, motivating further study of the scaling behavior of this architecture and its preferred allocation of parameters between depth and width.

A central tradeoff, described in §2.4, is reduced parallelism during pretraining and prompt prefilling. Informally, we observed training slowdowns of 3–10$\times$ relative to the standard Transformer,[3] whose training better exploits the fact that the string $x_{1:I}$ is known in advance.

[3]We do not report a wall-clock training comparison because our current implementation dispatches many fine-grained operations and is not optimized for this workload. A controlled comparison with fused kernels on the same GPU hardware is needed to cleanly separate out the cost of the architecture's sequential dependencies.

Thus, improving training efficiency is an important direction for future work. Distillation from a standard Transformer could provide supervision over the full next-token distribution, potentially improving learning efficiency even though the student's sequential dependencies remain. A complementary direction is a hybrid encoder–decoder architecture, pairing a pretrained standard Transformer encoder with a PIPET decoder. Such a design could combine parallel prompt encoding with pipelined decoding, while requiring the decoder to learn to use the encoder's representations.

Our evaluation is limited to models with 4M–200M parameters trained on SlimPajama. Although the decoding and cache advantages over the lowest-loss evaluated Transformer continue to grow through 200M, the loss gap also grows from 0.035 nats/token at 100M to 0.055 at 200M. At 200M we evaluate only the recipe-selected PIPET configuration, whereas the Transformer receives a six-point depth sweep. We therefore cannot tell whether a different depth, width, or input–output placement would recover this quality gap, or whether the observed efficiency trend continues at larger scales. Likewise, test loss does not establish performance on reasoning, code tasks, or post-training. Extending the evaluation to these settings would also help determine whether the observed preferences for shallow stacks and intermediate output layers persist.

Because our evaluation uses teacher forcing, comparable test loss does not rule out assigning excessive probability to poor choices that cause free-running decoding to drift and fail to recover. Future work should test non-teacher-forced decoding directly and determine whether imitation learning or reinforcement learning can correct such errors in this architecture.

Beyond predictive performance, the pre-input planning and post-output refining hypothesis decribed in §2.4 are concrete targets for further investigation. Mechanistic interpretability studies could examine what these representations encode and how they influence subsequent predictions. Varying the input layer, output layer, and decoding interval would provide controlled interventions for studying how computation before the input layer and after the output layer contributes to decoding.

One peculiarity of PIPET is that layer 0 is constant. One could instead make it random, so that h_i^0 provides some early information about the token x_i that will be sampled at layer ℓ_{out}. (The sampling distribution at layer ℓ_{out} will then itself be a random object, and does not need to carry all the entropy in x_i.) This means that lower layers of the residual stream contain more semantic information about x_i (and perhaps subsequent tokens), which can be usefully attended to by other streams. This variant would not be an ordinary autoregressive model, but a latent-variable model—a variational autoencoder that can be trained via a learned proposal distribution that imputes the random layer 0 given the string (Kingma & Welling, 2019).

Adaptive decoding is another promising direction: the model could choose ℓ_{in}, ℓ_{out}, or the number of recurrent depth updates separately for each stream, allocating more computation to difficult predictions and less to easier ones.

Finally, our pipelining approach could also be applied to stacked RNNs and LSTMs (Graves, 2014) (including models such as linear Transformers and Gated DeltaNet), which can already decode I tokens with only $O(LI)$ runtime and $O(L)$ memory. The standard topology would be preserved, with h_i^ℓ depending on h_{i-1}^ℓ and $h_i^{\ell-1}$. Sharma & Casas (2020) previously showed that encoding could be pipelined. To pipeline decoding as well, one could simply apply PIPET's trick of bringing ℓ_{in} and ℓ_{out} closer together than the usual 0 and L, reducing runtime to $O((\ell_{\text{out}} - \ell_{\text{in}})I)$.

6 Conclusion

We introduced PIPET, a pipelined architecture that overlaps computation across token positions while preserving autoregressive decoding and ordinary next-token training. Mixed-layer attention allows representations at different layers to interact, with computation extending both before the input layer and after the output layer. Across the evaluated parameter budgets, PIPET achieves competitive prediction quality with substantially faster decoding and smaller KV caches, at the cost of reduced parallelism during training and prefilling. This tradeoff makes it particularly relevant to workloads dominated by decoding and motivates exploring combinations with decoding-time scaling methods. Our results encourage a broader reconsideration of architectures and algorithms that jointly optimize prediction quality, decoding speed, and memory consumption.

AI USE STATEMENT

In this work, we used generative AI tools, including ChatGPT and Claude, for writing assistance such as grammar correction, clarity improvements, and presentation suggestions, as well as for implementation and debugging of experimental infrastructure. We did not use generative AI tools to determine the scientific ideas, experimental design, analyses, or claims presented in this work. We reviewed and verified all AI-assisted work and take responsibility for the final content of this paper, including all text, claims, code, and other artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

We provide the model definition, decoding procedure, and training objective in §2, together with explicit algorithms for the decoding schedules in the appendix. §B gives the computation and storage accounting, and §E reports the model parameterization, initialization, data construction, optimization settings, training-token budgets, and parameter-matching procedure. §E.2 specifies the hardware, software, compilation, timing, and KV-cache measurement protocols used for the efficiency results, while §E.3 reports the architecture-search procedure and individual configurations. The state-tracking construction and synthetic-task setup are provided in §C. All language-modeling losses are evaluated on a common held-out test split, with validation used for architecture selection, and all runtime and KV-cache comparisons use the same matched-output protocol.

REFERENCES

Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints, 2023. URL https://arxiv.org/abs/2305.13245.

Marianne Arriola, Aaron Gokaslan, Justin T. Chiu, Zhihan Yang, Zhixuan Qi, Jiaqi Han, Subham Sekhar Sahoo, and Volodymyr Kuleshov. Block diffusion: Interpolating between autoregressive and diffusion language models, 2025. URL https://arxiv.org/abs/2503.09573.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate, 2016. URL https://arxiv.org/abs/1409.0473.

William Brandon, Mayank Mishra, Aniruddha Nrusimha, Rameswar Panda, and Jonathan Ragan-Kelley. Reducing Transformer Key-Value Cache Size with Cross-Layer Attention, 2024. URL https://arxiv.org/abs/2405.12981.

David Chiang. Transformers in uniform tc^0, 2025. URL https://arxiv.org/abs/2409.13629.

SangLyul Cho, Langqing Cui, Sehoon Kim, Dongsu Han, and Insu Han. LoopSpec: Pipelined Self-Speculative Decoding for Looped Transformers, 2026. URL https://arxiv.org/abs/2609.17184.

Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness, 2022. URL https://arxiv.org/abs/2205.14135.

Soham De, Samuel L. Smith, Anushan Fernando, Aleksandar Botev, George Cristian-Muraru, Albert Gu, Ruba Haroun, Leonard Berrada, Yutian Chen, Srivatsan Srinivasan, Guillaume Desjardins, Arnaud Doucet, David Budden, Yee Whye Teh, Razvan Pascanu, Nando De Freitas, and Caglar Gulcehre. Griffin: Mixing gated linear recurrences with local attention for efficient language models, 2024. URL https://arxiv.org/abs/2402.19427.

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

Angela Fan, Thibaut Lavril, Edouard Grave, Armand Joulin, and Sainbayar Sukhbaatar. Addressing Some Limitations of Transformers with Feedback Memory, 2020. URL https://arxiv.org/abs/2002.09402.

Guhao Feng, Yihan Geng, Jian Guan, Wei Wu, Liwei Wang, and Di He. Theoretical benefit and limitation of diffusion language model, 2025. URL https://arxiv.org/abs/2502.09622.

Alex Graves. Generating sequences with recurrent neural networks, 2014. URL https://arxiv.org/abs/1308.0850.

Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces, 2024. URL https://arxiv.org/abs/2312.00752.

Namgyu Ho, Sangmin Bae, Taehyeon Kim, Hyunjik Jo, Yireun Kim, Tal Schuster, Adam Fisch, James Thorne, and Se-Young Yun. Block Transformer: Global-to-Local Language Modeling for Fast Inference, 2024. URL https://arxiv.org/abs/2406.02657.

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, Jack W. Rae, Oriol Vinyals, and Laurent Sifre. Training Compute-Optimal Large Language Models, 2022. URL https://arxiv.org/abs/2203.15556.

Coleman Hooper, Sehoon Kim, Hiva Mohammadzadeh, Hasan Genc, Kurt Keutzer, Amir Gholami, and Sophia Shao. SPEED: Speculative Pipelined Execution for Efficient Decoding. In *NeurIPS Workshop on Efficient Natural Language and Speech Processing*, 2023. URL https://arxiv.org/abs/2310.12072.

Da Ju, Stephen Roller, Sainbayar Sukhbaatar, and Jason Weston. Staircase Attention for Recurrent Processing of Sequences, 2021. URL https://arxiv.org/abs/2106.04279.

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.

Diederik P. Kingma and Max Welling. An introduction to variational autoencoders. *Foundations and Trends in Machine Learning*, 12(4):307–392, 2019. URL http://dx.doi.org/10.1561/2200000056.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP)*, 2023.

Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast Inference from Transformers via Speculative Decoding, 2022. URL https://arxiv.org/abs/2211.17192.

Belinda Z. Li, Zifan Carl Guo, and Jacob Andreas. (how) do language models track state?, 2025. URL https://arxiv.org/abs/2503.02854.

Xiang Lisa Li, John Thickstun, Ishaan Gulrajani, Percy Liang, and Tatsunori B. Hashimoto. Diffusion-LM Improves Controllable Text Generation, 2022. URL https://arxiv.org/abs/2205.14217. Advances in Neural Information Processing Systems 35 (NeurIPS 2022).

Aaron Lou, Chenlin Meng, and Stefano Ermon. Discrete Diffusion Modeling by Estimating the Ratios of the Data Distribution, 2024. URL https://arxiv.org/abs/2310.16834. International Conference on Machine Learning (ICML 2024).

Will Merrill and Ashish Sabharwal. A little depth goes a long way: The expressive power of log-depth transformers. In D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen (eds.), *Advances in Neural Information Processing Systems*, volume 38, Main Conference, pp. 95315–95339. Curran Associates, Inc., 2025. doi: 10.52202/085713-3186. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/88dd7aa6979e352fda7c4952ca8eac59-Paper-Conference.pdf.

William Merrill and Ashish Sabharwal. The Parallelism Tradeoff: Limitations of Log-Precision Transformers. *Transactions of the Association for Computational Linguistics*, 11:531–545, 2023. doi: 10.1162/tacl_a_00562. URL https://aclanthology.org/2023.tacl-1.31/.

William Merrill and Ashish Sabharwal. The Expressive Power of Transformers with Chain of Thought. In *International Conference on Learning Representations*, 2024. URL https://arxiv.org/abs/2310.07923.

William Merrill, Jackson Petty, and Ashish Sabharwal. The Illusion of State in State-Space Models. In *Proceedings of the 41st International Conference on Machine Learning (ICML)*, pp. 35492–35506, 2024. URL https://proceedings.mlr.press/v235/merrill24a.html.

Shen Nie, Fengqi Zhu, Zebin You, Xiaolu Zhang, Jingyang Ou, Jun Hu, Jun Zhou, Yankai Lin, Ji-Rong Wen, and Chongxuan Li. Large Language Diffusion Models, 2025. URL https://arxiv.org/abs/2502.09992. Advances in Neural Information Processing Systems 38 (NeurIPS 2025).

NVIDIA, Aaron Blakeman, Aarti Basant, Abhinav Khattar, et al. Nemotron-H: A Family of Accurate and Efficient Hybrid Mamba-Transformer Models, 2025. URL https://arxiv.org/abs/2504.03624.

Bo Peng, Eric Alcaide, Quentin Anthony, Alon Albalak, Samuel Arcadinho, Stella Biderman, Huanqi Cao, Xin Cheng, Michael Chung, Matteo Grella, Kranthi Kiran GV, Xuzheng He, Haowen Hou, Jiaju Lin, Przemyslaw Kazienko, Jan Kocon, Jiaming Kong, Bartlomiej Koptyra, Hayden Lau, Krishna Sri Ipsit Mantri, Ferdinand Mom, Atsushi Saito, Guangyu Song, Xiangru Tang, Bolun Wang, Johan S. Wind, Stanislaw Wozniak, Ruichong Zhang, Zhenyuan Zhang, Qihang Zhao, Peng Zhou, Qinghua Zhou, Jian Zhu, and Rui-Jie Zhu. Rwkv: Reinventing rnns for the transformer era, 2023. URL https://arxiv.org/abs/2305.13048.

Liliang Ren, Yang Liu, Yadong Lu, Yelong Shen, Chen Liang, and Weizhu Chen. Samba: Simple hybrid state space models for efficient unlimited context language modeling, 2025. URL https://arxiv.org/abs/2406.07522.

Subham Sekhar Sahoo, Marianne Arriola, Yair Schiff, Aaron Gokaslan, Edgar Marroquin, Justin T. Chiu, Alexander Rush, and Volodymyr Kuleshov. Simple and Effective Masked Diffusion Language Models. In *Advances in Neural Information Processing Systems*, 2024. URL https://arxiv.org/abs/2406.07524.

Tal Schuster, Adam Fisch, Jai Gupta, Mostafa Dehghani, Dara Bahri, Vinh Q. Tran, Yi Tay, and Donald Metzler. Confident Adaptive Language Modeling, 2022. URL https://arxiv.org/abs/2207.07061.

Robin Kumar Sharma and Marc Casas. Wavefront parallelization of recurrent neural networks on multi-core architectures. In *Proceedings of the 34th ACM International Conference on Supercomputing*, ICS '20, 2020. ISBN 9781450379830. doi: 10.1145/3392717.3392762. URL https://doi.org/10.1145/3392717.3392762.

Noam Shazeer. Fast Transformer Decoding: One Write-Head is All You Need, 2019. URL https://arxiv.org/abs/1911.02150.

Zhiqiang Shen, Tianhua Tao, Liqun Ma, Willie Neiswanger, Zhengzhong Liu, Hongyi Wang, Bowen Tan, Joel Hestness, Natalia Vassilieva, Daria Soboleva, and Eric Xing. Slimpajama-dc: Understanding data combinations for llm training, 2024. URL https://arxiv.org/abs/2309.10818.

Paul Soulos, Aleksandar Terzić, Michael Hersche, and Abbas Rahimi. Recurrent Transformers Trade-off Parallelism for Length Generalization on Regular Languages. In *The First Workshop on System-2 Reasoning at Scale, NeurIPS*, 2024. URL https://openreview.net/forum?id=6PjZA4Jvge.

Lena Strobl, William Merrill, Gail Weiss, David Chiang, and Dana Angluin. What formal languages can transformers express? a survey. *Transactions of the Association for Computational Linguistics*, 12:543–561, 05 2024. ISSN 2307-387X. doi: 10.1162/tacl_a_00663. URL https://doi.org/10.1162/tacl_a_00663.

Jianlin Su, Yu Lu, Shengfeng Pan, Ahmed Murtadha, Bo Wen, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding. *Neurocomputing*, 568:127063, 2024. doi: 10.1016/j.neucom.2023.127063.

Yutao Sun, Li Dong, Yi Zhu, Shaohan Huang, Wenhui Wang, Shuming Ma, Quanlu Zhang, Jianyong Wang, and Furu Wei. You Only Cache Once: Decoder-Decoder Architectures for Language Models, 2024. URL https://arxiv.org/abs/2405.05254.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention Is All You Need, 2017. URL https://arxiv.org/abs/1706.03762.

Seongjun Yang, Gibbeum Lee, Jaewoong Cho, Dimitris Papailiopoulos, and Kangwook Lee. Predictive Pipelined Decoding: A Compute-Latency Trade-off for Exact LLM Decoding. *Transactions on Machine Learning Research*, 2024a. URL https://arxiv.org/abs/2307.05908.

Songlin Yang, Jan Kautz, and Ali Hatamizadeh. Gated Delta Networks: Improving Mamba2 with Delta Rule, 2024b. URL https://arxiv.org/abs/2412.06464.

Songlin Yang, Bailin Wang, Yu Zhang, Yikang Shen, and Yoon Kim. Parallelizing Linear Transformers with the Delta Rule over Sequence Length. In *Advances in Neural Information Processing Systems*, 2024c. URL https://arxiv.org/abs/2406.06484.

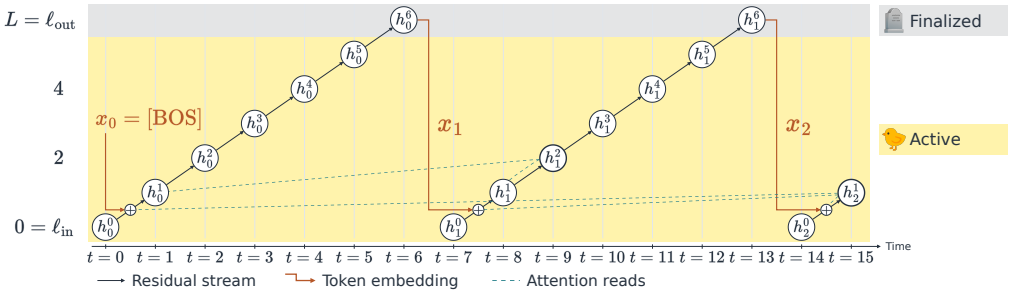


Figure 4: **Standard Transformer decoding in the time–layer view of Figure 1.** A residual stream receives its token at layer $0 = \ell_{\text{in}}$, traverses all $L = 6$ layers sequentially, and predicts the next token at layer $L = \ell_{\text{out}}$ before the successor stream begins. The horizontal axis therefore spans six layer-update timesteps per decoded token. Dashed teal lines show example causal attention reads from a query to earlier positions at the same layer. Unlike PIPET, the standard Transformer performs no computation before receiving a stream's token or after that stream predicts its successor.

A Decoding schedules and the stalling tradeoff

The **decoding interval** τ is a hyperparameter specifying the number of sequential computation rounds between successive outputs in steady state. The main text describes PIPET under $\tau = 1$, which produces one token per parallel layer-update round. Here we consider larger decoding intervals, which allow additional sequential computation between successive outputs and richer interactions among stream representations.

Realizing these opportunities requires deciding where to stall. In a CPU pipeline, an instruction may stall while waiting for a result produced by an earlier instruction. Here, deliberately stalling some streams allows others to advance further before the next mixed-layer transition, making their higher representations available to attention.

For comparison with the main architecture diagram, Figure 4 shows standard Transformer decoding using the same explicit time–layer view as Figure 1. Figure 5 then uses a complementary compact view, encoding computation time by node color, to compare standard decoding with the two stalled pipeline schedules.

We consider two ways to place stalls in the pipelined schedule. **Local stalling** delays the introduction of new streams while existing streams continue to advance. **Global stalling** pauses the other streams while one stream performs additional layer updates. These choices lead to different patterns of sparsity across depth and time. Figure 5 contrasts both schedules with standard Transformer decoding. We first develop the two schedules, then evaluate the quality–efficiency tradeoff introduced by global stalling.

A.1 Scheduling constraints

Under the definitions in §2.1, the sequential path from the input layer to the output layer requires

$$\ell_{\text{out}} - \ell_{\text{in}} \leq \tau. \tag{6}$$

A standard L-layer Transformer has $\tau = L$. For a given $\tau > 1$, this constraint leaves freedom to choose when different streams advance. Throughout, mixed-layer attention reads the highest available state of each existing stream.

We use **just-in-time initiation**: a nascent stream is introduced at the latest mixed-layer transition that still lets it reach the input layer when its token becomes available. Delaying initiation in this way lets nascent computation use the most recent available context. Creating all streams in advance would instead let later streams complete this computation using older context and then wait at the input layer.

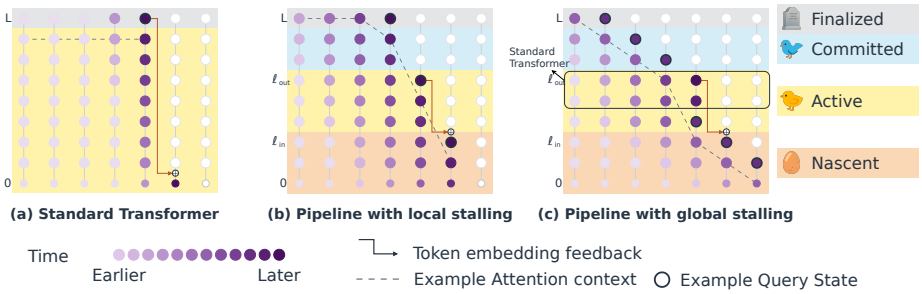


Figure 5: **Standard Transformer decoding and two stalling schedules for PIPET.** Columns represent stream positions, with depth increasing upward. Node colors indicate computation time; background colors mark stream stages. Dashed lines connect states whose projected keys and values form an attention context; solid arrows indicate token-embedding feedback. **(a)** A standard Transformer completes the full stack before starting the next position. **(b)** Local stalling delays stream initiation while existing streams advance, producing a frontier sparse across depth. **(c)** Global stalling pauses other streams while the decoding stream advances from $\ell_{\text{in}} + 1$ to ℓ_{out} through standard attention layers. This preserves consecutive layers in the mixed-layer regions while spacing parallel mixed-layer transitions farther apart in time.

A.2 Local stalling

A direct schedule sets $\ell_{\text{out}} - \ell_{\text{in}} = \tau$ and advances every unfinished stream by one layer per round.[4] Successive streams then start τ rounds apart. For $\tau > 1$, we call this **local stalling**: initiation of the next stream is delayed while existing streams continue to advance (Figure 5(b)). At $\tau = 1$, a new stream enters every round and no stalling is needed.

A.3 Global stalling and the sparsity tradeoff

With local stalling, new streams enter every τ rounds while existing streams advance one layer per round. For integer $\tau > 1$, consecutive streams are therefore separated by τ layers in steady state. For example, when $\tau = 3$, the highest available states of three consecutive streams may be

$$(i, 7),\ (i+1, 4),\ (i+2, 1). \tag{7}$$

Only approximately L/τ distinct layers participate in each mixed-layer transition. Increasing τ thus makes attention **sparse across depth**: each transition draws on fewer of the model's intermediate layers.

We introduce **global stalling** to reduce this sparsity. During a global stall, one stream performs consecutive standard-attention layer updates while all other streams remain fixed. We place these updates between the input and output layers: the first update after the input layer uses mixed-layer attention, and the remaining $\tau - 1$ updates use standard attention with causal masking.

This allocation allows a new stream to enter after *every* global mixed-layer transition. Consecutive streams then occupy consecutive layers within the mixed-layer regions. At this scheduling resolution, the input–output layer segment acts as one stage, with its standard layers traversed during the stall.

Global stalling therefore makes the representations available to mixed-layer attention denser across depth, while spacing mixed-layer transitions farther apart in time. It **trades sparsity across depth for sparsity in time**: the intervening rounds are spent on local refinement within the selected stream. At $\tau = 1$, no stalling is needed.

[4] At the endpoint $\ell_{\text{out}} - \ell_{\text{in}} = \tau = L$, the only possible placement is $\ell_{\text{in}} = 0$ and $\ell_{\text{out}} = L$. This recovers the standard Transformer's serial depth schedule, but attention reads terminal representations of preceding tokens rather than their layer-specific states.

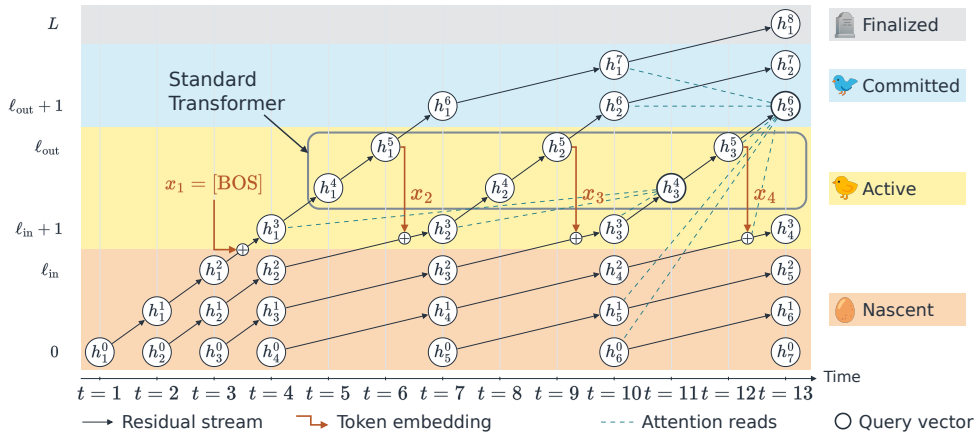


Figure 6: **Detailed timeline for global stalling.** The example uses $(L, \ell_{\text{in}}, \ell_{\text{out}}) = (8, 2, 5)$ and decoding interval $\tau = 3$. Black arrows trace residual streams, and orange arrows pass predicted-token embeddings to successor streams. After a parallel mixed-layer transition, the decoding stream alone traverses two standard causal-attention layers while the remaining streams stall. The schedule therefore produces one output every three computation rounds. Background colors mark the nascent, active, committed, and finalized stages.

Figure 6 shows the resulting computation timeline for $\tau = 3$. Unlike the fully pipelined schedule in Figure 1, the decoding stream advances through two standard-attention layers while all other streams pause, so consecutive outputs are separated by three computation rounds.

Layer allocation and decoding cycle. The global-stalling allocation sets $\ell_{\text{out}} - \ell_{\text{in}} = \tau$. Layers $1, \ldots, \ell_{\text{in}} + 1$ and $\ell_{\text{out}} + 1, \ldots, L$ use mixed-layer attention; the intervening $K = \tau - 1$ layers form an **isolated pause block** of standard Transformer layers. Each standard-attention layer uses its own layer-specific history as in equation (2). All layers count toward the terminal layer L.

Each cycle consists of one parallel mixed-layer transition followed by K sequential standard-attention updates. At the cycle boundary, all unfinished streams are at mixed-layer inputs, so all advance in the parallel transition. Only the decoding stream then traverses the pause block. Readout at the output layer and entry of the successor stream follow equation (4).

Algorithm 2 extends Algorithm 1 to this schedule using the same stream notation and event-driven control flow. The black lines follow the main algorithm. The blue block is the only addition: after the decoding stream enters the isolated pause block, all other streams stall while it advances through standard-attention layers until reaching the output layer. When the pause block is empty, the algorithm reduces to Algorithm 1.

Note each standard-attention update reads the causal history previously computed at its source layer thus an implementation also needs to save those layer's history to the KV cache. Mixed-layer updates continue to read the highest available state of every stream, as in §2.2.1. Initiating one nascent stream after each mixed-layer transition ensures that a tokenless successor reaches the input layer exactly when the preceding stream produces its token; the token is then injected before the next mixed-layer transition.

A.4 Experiments with global stalling

The decoding interval τ controls a tradeoff between parallel computation and sequential refinement. At fixed total depth, global stalling replaces layers using mixed-layer attention with standard-attention layers, moving from the fully pipelined architecture toward a predominantly standard Transformer stack. The endpoint retains one layer using mixed-layer attention and therefore differs from a standard Transformer. Motivated by the competitive performance of shallow PIPET

Algorithm 2 Decoding with optional global stalling, extending Algorithm 1.

Require: $0 \le \ell_{\text{in}} < \ell_{\text{out}} \le L$; model parameters; BOS and EOS tokens
Ensure: A decoded sequence ending in EOS and all token distributions

// Streams have the representation and fields defined in Algorithm 1.
// Blue lines are the global-stalling extension.

$Streams \leftarrow \{(0, 0, h_0)\}$
$x_0 \leftarrow \text{BOS}$; $i \leftarrow 0$ ▷ i tracks the current output token position
while true **do**
  // Inject BOS as soon as its stream reaches the input layer.
  if $i = 0$ and a stream u is at layer ℓ_{in} **then**
    $u.h \leftarrow u.h + \text{Emb}(x_0)$ ▷ Inject BOS
  // Advance streams whose next update uses mixed-layer attention.
  for all unfinished $s \in Streams$ whose next layer uses mixed-layer attention, in parallel **do**
    $s.h \leftarrow \text{LayerUpdate}_{s.\ell+1}(\text{query} = s.h,\ \text{KVContext} = Streams)$
    $s.\ell \leftarrow s.\ell + 1$
  // Stall other streams while the decoding stream traverses the pause block.
  if a stream c has just entered the isolated pause block **then**
    while c has not reached layer ℓ_{out} **do**
      $c.h \leftarrow \text{LayerUpdate}_{c.\ell+1}(\text{query} = c.h,\ \text{KVContext} = \text{causal history at layer } c.\ell)$
      $c.\ell \leftarrow c.\ell + 1$
  // Create the next stream.
  Append $(s.i, s.\ell, s.h) = (\text{next position}, 0, h_0)$ to $Streams$
  // Predict and inject when a stream reaches the output layer.
  if a new stream c has just reached layer ℓ_{out} **then**
    $p_i \leftarrow \text{softmax}(\text{Unemb}(c.h))$ ▷ Read from the output layer
    $x_{i+1} \sim p_i$ ▷ Use the observed token under teacher forcing
    if $x_{i+1} = \text{EOS}$ **then**
      return $x_{1:i+1}, p_{0:i}$ ▷ Stop after decoding EOS
    $u \leftarrow$ the stream in $Streams$ at layer ℓ_{in}
    $u.h \leftarrow u.h + \text{Emb}(x_{i+1})$ ▷ Inject before the next mixed-layer update
    $i \leftarrow i + 1$

configurations in §3.4, we investigate whether a small mixed-layer component can also benefit a stack containing mostly standard layers.

Experimental setup. We fix the total depth at $L = 6$, hidden width at 640, number of attention heads at 10, and feed-forward dimension at 1532. Layers using mixed-layer attention and standard attention use the same normalization and feed-forward parameterization, so changing their allocation preserves the parameter count of 30,005,120. We vary τ from 1 to 6, using $K = \tau - 1$ standard-attention layers and $L - K$ layers with mixed-layer attention. For $\tau \le 5$, the input layer remains at $\ell_{\text{in}} = 1$ and the output layer moves to $\ell_{\text{out}} = 1 + \tau$. At $\tau = 6$, we adopt the only feasible placement, $(\ell_{\text{in}}, \ell_{\text{out}}) = (0, 6)$.

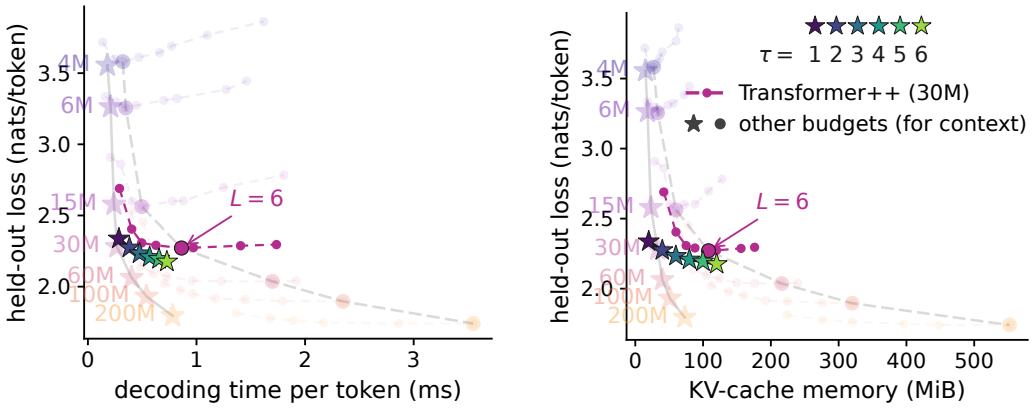
All configurations follow the training protocol in §3.1, with 600M training tokens. The $\tau = 1$ reference is the six-layer configuration from the depth–width sweep in Figure 9a; configuration details are listed in Table 7. We also report the six-layer Transformer++ baseline at the same parameter and training-token budgets.

Prediction quality and computational cost. Table 1 summarizes prediction quality and decoding cost for each allocation, and Figure 7 places the sweep on the quality–efficiency axes of Figure 2. A decoding cycle contains one parallel mixed-layer transition and $K = \tau - 1$ standard updates, giving τ sequential steps. Each token contributes one shared terminal key–value pair and K layer-specific pairs, also totaling τ (§B).

Test loss decreases monotonically from 2.3371 at $\tau = 1$ to 2.1754 at $\tau = 6$. The marginal improvement generally diminishes: increasing τ from 1 to 2 reduces loss by 0.063 nats/token, whereas each of the last two increments improves loss by approximately 0.014–0.018 nats/token. Meanwhile,

Table 1: **Global stalling at 30M parameters and fixed depth $L = 6$.** Test loss is measured after 600M training tokens. Architecture diagrams order layers from left to right: blue denotes mixed-layer attention, yellow denotes standard attention, and arrows indicate token-embedding feedback. Sequential attention steps count parallel updates as one step. Bold losses improve on Transformer++. Runtime per token and KV cache are from the matched-output benchmark of §E.2.

Configuration	τ	Architecture	Sequential attention steps	Loss	Runtime per token (ms/tok)	KV cache (MiB)
PIPET	1		1	2.3371	0.285	19.8
	2		2	2.2740	0.382	39.9
	3		3	**2.2331**	0.476	59.9
	4		4	**2.2074**	0.569	80.0
	5		5	**2.1937**	0.655	100.0
	6		6	**2.1754**	0.727	120.0
Transformer++	6		6	2.2711	0.862	108.0



(a) Runtime per token

(b) KV-cache memory

Figure 7: **The decoding interval as a tradeoff axis, on the quality–efficiency axes of Figure 2.** Stars joined by the solid line are the $\tau = 1, \ldots, 6$ configurations of Table 1, all at 30M parameters and total depth 6. The 30M Transformer++ depth sweep is drawn in full colour for comparison, with its matched-depth $L = 6$ member outlined; the remaining budgets are faded and carry no comparison here. Increasing τ moves down and to the right: lower loss at higher runtime per token and larger cache. **(a)** Every τ decodes faster than the matched-depth Transformer++. **(b)** $\tau \leq 5$ also uses a smaller cache; $\tau = 6$ retains 120 MiB against 108 MiB. Measurement details are in §E.2.

sequential attention steps and retained KV pairs grow linearly with τ. Configurations with $\tau \geq 3$ achieve lower loss than the Transformer++ baseline. At $\tau = 6$, a single layer using mixed-layer attention beneath five standard-attention layers achieves 0.096 nats/token lower loss.

Interpretation and future directions. These results suggest that mixed-layer attention can be useful both throughout the model and as a small component within a predominantly standard stack. At $\tau = L$, the remaining mixed-layer-attention update reads terminal representations of preceding streams, introducing a feedback connection from their completed computation to the bottom of the stack. This resembles the use of higher-level historical representations in Feedback Transformer (Fan et al., 2020).

This configuration motivates two extensions. First, replacing the standard-attention block with a recurrent block could allow its number of refinement steps, and hence the decoding interval, to vary across predictions. Second, a pretrained Transformer could initialize the standard-attention

block and be adapted jointly with a small number of layers using mixed-layer attention, potentially reducing training cost and supporting distillation from the original model.

For efficient decoding, larger decoding intervals carry an increasing cost in sequential computation and retained history, while yielding generally smaller incremental reductions in loss. In the measured tradeoff (Figure 7), $\tau \in \{3, 4, 5\}$ improves on the six-layer Transformer in loss, runtime per token, and KV-cache memory. At $\tau = 6$, the model remains faster and more accurate but uses 120 MiB rather than 108 MiB of cache. Together with the shallow configurations identified in §3.4, these results motivate treating the decoding interval as an additional architectural scaling axis: for a fixed parameter budget and training-data size, it is worth varying τ alongside the model's layer allocation (Figure 2).

A.5 Stream layers and pipeline occupancy

We describe the steady progression of the streams in Algorithm 2 in more detail. Here t indexes decoding cycles. Each cycle contains one parallel mixed-layer transition followed by the standard-attention updates inside the isolated pause block. The event-driven startup fills the lower pipeline before the first prediction.

At the start of cycle $t \geq 0$, let $\mathcal{V}_t = \{0, \ldots, t + \ell_{\text{in}}\}$ denote the set of existing streams. The highest available layer of stream $j \in \mathcal{V}_t$ is

$$\ell_j(t) = \begin{cases} \min\{L, \ell_{\text{out}} + t - j - 1\}, & 0 \leq j < t, \\ \ell_{\text{in}} + t - j, & t \leq j \leq t + \ell_{\text{in}}. \end{cases} \tag{8}$$

Earlier streams have reached their output layer and continue advancing until finalization. Stream t is at the input layer, while later streams are still nascent.

All mixed-layer updates read the same pre-update snapshot. As described in Algorithm 2, each eligible stream $i \in \mathcal{V}_t$ performs

$$h_i^{\ell_i(t)+1} = \text{LayerUpdate}_{\ell_i(t)+1}\Big(\text{query} = h_i^{\ell_i(t)},\ \text{KVContext} = \{h_j^{\ell_j(t)}\}_{j \in \mathcal{V}_t}\Big), \tag{9}$$

where the right-hand side uses the frozen snapshot rather than partially updated states. The layer operator applies the key–value map corresponding to each source stream's highest available layer. Finalized streams remain in $\mathcal{V}_t$ with $\ell_j(t) = L$ and supply their cached terminal pairs.

All $\ell_{\text{in}} + 1$ layers in the lower mixed-layer region participate from cycle 0. The upper mixed-layer region fills one depth per cycle, so cycle t performs $\ell_{\text{in}} + 1 + \min\{t, L - \ell_{\text{out}}\}$ mixed-layer updates in parallel. Once the upper region is full, each cycle performs $L - K$ such updates in parallel and K standard layer applications sequentially. Thus steady-state decoding performs L layer applications per decoded token across $\tau = K + 1$ computation rounds. At $\tau = 1$, all L applications occur in one parallel round.

B Computation and storage costs

We analyze the schedule in Algorithm 2, with total depth L and $K = \tau - 1$ standard-attention layers between the input and output layers. The main experiments use $\tau = 1$, so $K = 0$ and all layers use mixed-layer attention.

Arithmetic intensity. Consider one attention head reading a history of n positions, with head dimension d_h and b bytes per cached key or value element. Computing attention scores and the weighted value sum requires approximately $4nd_h$ floating-point operations while reading $2nd_hb$ bytes of keys and values. When these reads dominate memory traffic, the arithmetic intensity of standard Transformer decoding is therefore

$$\text{AI}_{\text{T}} \approx \frac{4nd_h}{2nd_hb} = \frac{2}{b} \quad \text{FLOPs/byte}. \tag{10}$$

This ratio is independent of context length and remains low when batching separate sequence-specific histories, so decoding attention is commonly limited by memory traffic rather than available compute.

In the fully pipelined schedule, all L layers issue queries against the same shared history. Loading that history once and reusing it across the L queries performs approximately $4Lnd_h$ operations for the same $2nd_hb$ bytes, giving

$$\mathrm{AI}_{\mathrm{P}} \approx \frac{2L}{b} \quad \text{FLOPs/byte.} \tag{11}$$

This is an ideal L-fold increase when the attention implementation realizes the reuse. Under global stalling, the $L - K$ concurrent mixed-layer queries receive this shared-history benefit, while the K standard-attention updates read their layer-specific histories sequentially.

Sequential computation and total work. A standard Transformer of depth L_{T} performs L_{T} sequential layer applications per output. In steady state, PIPET performs $L - K$ mixed-layer updates in parallel, followed by K sequential standard-attention updates on the decoding stream. Thus each output requires L layer applications across $\tau = K + 1$ sequential computation rounds (§A.5).

Let d denote hidden width, n_f the number of finalized streams, a the number of unfinished streams, and n the number of positions in each standard-layer history. With dense attention, a fixed feed-forward expansion ratio, and source projections shared across mixed-layer queries, the arithmetic work per output per sequence is

$$\mathcal{W}_{\mathrm{P}} = \Theta\big(Ld^2 + \big[(L-K)(n_f + a) + Kn\big]d\big)\,. \tag{12}$$

The first term accounts for projections and feed-forward computation. The second accounts for the $L - K$ mixed-layer queries over the shared context and the K standard queries over their layer-specific histories. Under this schedule, $a = O(L)$. The vocabulary projection contributes an additional cost.

Pipelining therefore reduces sequential dependency depth without eliminating the work associated with the full stack. Its effect on wall-clock runtime depends on how effectively parallel execution uses the hardware, as well as on model width and context length.

Retained histories and frontier state. Let B be batch size, q the number of bytes per stored element, and $d_{\mathrm{KV}} = h_{\mathrm{KV}}d_h$ the width of a key or value vector. A standard Transformer retains

$$M^{\mathrm{T}}_{\mathrm{KV}}(n) = 2BqnL_{\mathrm{T}}d^{\mathrm{T}}_{\mathrm{KV}}. \tag{13}$$

The factor of two accounts for keys and values. For PIPET, finalized streams contribute one shared terminal history, while each of the K standard layers retains its own history:

$$\begin{aligned} M^{\mathrm{P}}_{\mathrm{state}}(n) &= 2Bq\left(n_f d^{\mathrm{mixed}}_{\mathrm{KV}} + nKd^{\mathrm{standard}}_{\mathrm{KV}}\right) + M_{\mathrm{frontier}}, \\ M_{\mathrm{frontier}} &= O\big(BqL(d + d^{\mathrm{mixed}}_{\mathrm{KV}})\big)\,. \end{aligned} \tag{14}$$

Here we assume a common storage precision for the frontier and histories. The frontier contains unfinished streams' hidden states and, if retained, their current key–value projections. These projections may instead be recomputed from the hidden states, as in our evaluated implementation. The frontier bound excludes other temporary workspace.

At fixed depth, $n - n_f = O(L)$, so the ratio of Transformer to PIPET history-growth slopes is

$$\frac{L_{\mathrm{T}}d^{\mathrm{T}}_{\mathrm{KV}}}{d^{\mathrm{mixed}}_{\mathrm{KV}} + Kd^{\mathrm{standard}}_{\mathrm{KV}}}. \tag{15}$$

At equal KV widths, this ratio is L_{T}/τ; at equal depth as well, it becomes L/τ. For $\tau = 1$, each finalized stream adds $2Bqd^{\mathrm{mixed}}_{\mathrm{KV}}$ bytes. With ordinary multi-head attention and fp32 storage, this is $8Bd$ bytes.

Suppressing width, batch size, and precision, retained histories and frontier state scale as $O(\tau n + L)$, reducing to $O(n + L)$ at $\tau = 1$. This state accounting is broader than the KV-buffer metric in Figure 2b, which excludes hidden states and temporary projections. Finite-length cache sizes also depend on pipeline startup and allocated capacity. We therefore report measured KV-buffer bytes using the protocol in §E.2, rather than treating asymptotic growth ratios as measured memory reductions.

C EXPRESSIVITY: RECURRENT STATE TRACKING

The pipelined schedule induces a recurrent computation across streams: the update that produces the current stream's output state can attend directly to the preceding stream's output state. This creates a sequential state-propagation path whose length grows with the input sequence despite the model having fixed parameterized depth.

n this section, we use this recurrent path to establish an expressivity separation from fixed-depth sequence models characterized by TC^0. In particular, we show how PIPET can implement finite-state recurrence and hence compute permutation composition in $\mathcal{S}_5$, an NC^1-complete state-tracking problem. Under the standard conjecture $\mathsf{TC}^0 \neq \mathsf{NC}^1$, this yields an expressivity separation from the corresponding fixed-depth Transformer formalizations characterized by TC^0.

We illustrate the construction in the simplest setting: a one-layer PIPET with

$$L = 1, \qquad \ell_{\text{in}} = 0, \qquad \ell_{\text{out}} = 1. \tag{16}$$

This is the unique input–output placement for a one-layer model. Let

$$s_i := h_i^1 \tag{17}$$

denote the layer-1 output representation of stream i. Stream i receives the current input x_i at layer 0 and produces s_i at layer 1. When s_i is computed, tilted attention can read the preceding layer-1 representation s_{i-1}. Hence the update can depend jointly on the current input x_i and the state encoded in s_{i-1}.

C.1 FINITE-STATE RECURRENCE IN ONE LAYER

Let

$$\mathcal{A} = (\mathcal{Q}, \Sigma, \delta, q_0) \tag{18}$$

be a deterministic finite-state recurrent system, where $\mathcal{Q}$ is a finite state set, Σ is an input alphabet, and

$$\delta : \mathcal{Q} \times \Sigma \to \mathcal{Q} \tag{19}$$

is the transition map. On inputs $x_1, \ldots, x_T \in \Sigma$, the state trajectory is

$$q_i = \delta(q_{i-1}, x_i). \tag{20}$$

We reserve a designated subspace of s_i to encode the recurrent state, writing $e(q_i)$ for the encoding of q_i. It suffices for the one-layer update to implement the following three operations.

Previous-state retrieval. A designated attention head retrieves $e(q_{i-1})$ from the preceding output representation s_{i-1}.

Current-input access. The current stream contains the representation of x_i, injected at layer 0.

State update. The layer-1 update therefore has access to

$$\big(e(q_{i-1}), e(x_i)\big). \tag{21}$$

Because $\mathcal{Q}$ and Σ are finite, the transition

$$(q, x) \mapsto \delta(q, x) \tag{22}$$

is a finite map. With sufficient width, the layer update can realize this map on the chosen encodings and write

$$e(\delta(q_{i-1}, x_i)) \tag{23}$$

into the designated coordinates of s_i. Repeating the same update across streams realizes equation (20) for arbitrary sequence length.

The recurrent state is therefore propagated through successive output representations, while the current input is supplied through the ordinary embedding injection into the active stream.

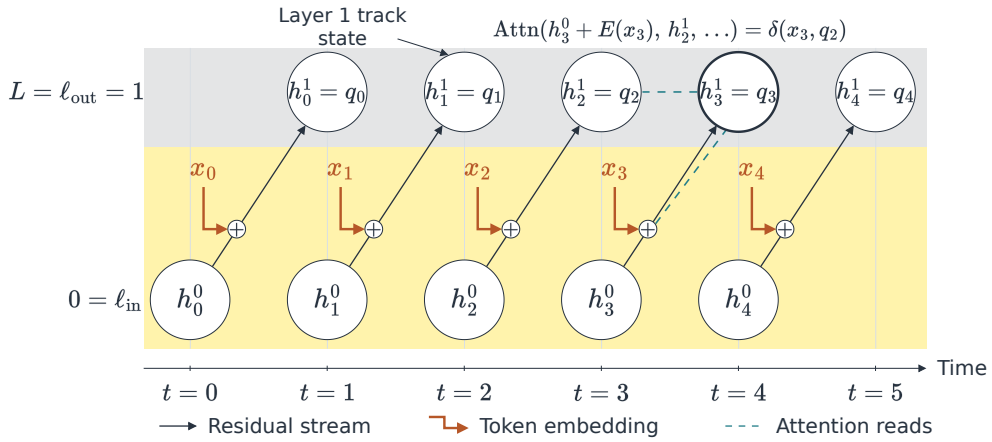


Figure 8: **One-layer recurrent state tracking in PIPET.** We show the $L = 1$ configuration with $\ell_{\mathrm{in}} = 0$ and $\ell_{\mathrm{out}} = 1$. Each stream receives the current input x_i at layer 0, and its layer-1 output representation encodes the recurrent state q_i. The highlighted update illustrates how tilted attention reads the preceding state q_{i-1} while x_i is injected into the current stream, allowing the update to implement $q_i = \delta(q_{i-1}, x_i)$. For permutation composition, this becomes $q_i = \pi_i \circ q_{i-1}$. For clarity, only the attention read relevant to the highlighted recurrent update is shown; the remaining attention context is omitted.

C.2 Permutation composition in S_5

Permutation composition provides a concrete instance of this recurrent update. Let

$$\pi_1, \ldots, \pi_T \in \mathcal{S}_5 \tag{24}$$

be a sequence of T permutations of five elements, and define

$$q_0 = \mathrm{id}, \qquad q_i = \pi_i \circ q_{i-1}. \tag{25}$$

Thus the current input is $x_i = \pi_i$ and the transition map is

$$\delta(q, \pi) = \pi \circ q. \tag{26}$$

Since $|\mathsf{S}_5| = 120$, this is a finite-state recurrence and fits the construction above directly.

Permutation composition is a standard state-tracking problem in the expressivity literature. Computing products in a fixed nonsolvable finite group such as $\mathcal{S}_5$ is NC^1-complete (Merrill & Sabharwal, 2025; Li et al., 2025). The construction above therefore gives PIPET a recurrent computation capable of realizing an NC^1-complete state-tracking problem with fixed parameterized depth. Under the standard conjecture $\mathsf{TC}^0 \neq \mathsf{NC}^1$, this separates the construction from the corresponding fixed-depth Transformer formalizations characterized by TC^0.

C.3 Permutation-group word problems

We empirically test the state-tracking construction above using permutation-group word problems. We train PIPET and Transformer++ on the recurrence in equation (25), using $\mathcal{S}_3$ and $\mathcal{S}_5$. Given a permutation sequence $\pi_1, \ldots, \pi_T$, each model predicts the corresponding running states $q_1, \ldots, q_T$, with cross-entropy loss applied at every position (Li et al., 2025).

We additionally evaluate the parity of the running permutation product (the oddity of the number of inversions it contains). Parity retains only whether the composed permutation is even or odd and therefore reduces the recurrent state to two possibilities, providing an easier state-tracking task.

Figure 3 shows test accuracy as sequence length extends beyond the training regime. PIPET maintains near-perfect permutation-state accuracy both within and beyond the training lengths, while

Transformer++ degrades substantially with length. The separation is largest on $\mathcal{S}_5$, where the running state ranges over $5! = 120$ possible permutations. Both architectures perform better on parity, consistent with the substantially smaller state space. As a sanity check, we run GRU on this task and also observe near-perfect test accuracy similar to PIPET.

D Additional related work

Recurrent alternatives to attention. Several recent architectures have attempted to mitigate the limitations of transformers by using recursive components (Peng et al., 2023). Mamba maintains a selective recurrent state, while DeltaNet and Gated DeltaNet use matrix-valued states with delta-rule updates (Gu & Dao, 2024; Yang et al., 2024c;b). Hybrids such as Samba and Nemotron-H combine recurrent layers with attention (Ren et al., 2025; NVIDIA et al., 2025). PIPET retains softmax attention over individual stream representations and introduces recurrence through their evolution across depth and autoregressive time. Its shared history still grows with sequence length.

Pipelined and speculative decoding. Speculative decoding verifies proposed tokens in parallel and can preserve the target model's sampling distribution (Leviathan et al., 2022). SPEED and Predictive Pipelined Decoding more directly overlap work on successive tokens by initiating computation from tentative predictions (Hooper et al., 2023; Yang et al., 2024a); LoopSpec applies pipelined self-speculation to looped Transformers (Cho et al., 2026). PIPET instead trains an autoregressive architecture whose nascent streams can advance before receiving their tokens, without speculative token guesses or a verification stage. Adaptive-depth methods such as CALM select earlier exits (Schuster et al., 2022), whereas our output layer placement permits subsequent computation to contribute to future predictions.

Architecture expressivity. Circuit-complexity analyses relate the power of bounded-depth Transformers to their parallelizability under explicit precision assumptions (Merrill & Sabharwal, 2023). Chain-of-thought results show how intermediate decoding can expand computational power (Merrill & Sabharwal, 2024), while studies of recurrent Transformers connect recurrence and chunk size to state tracking (Ju et al., 2021; Soulos et al., 2024). These works motivate examining the computation graph across autoregressive time, in addition to the number of architectural layers. In PIPET, additional computation occurs through evolving stream states; we empirically validate the added expressivity of PIPET in 3.3.

Diffusion language models. Masked diffusion models refine multiple positions through denoising, providing another route to parallel decoding (Sahoo et al., 2024; Nie et al., 2025). Block Diffusion combines autoregressive conditioning between blocks with diffusion within each block (Arriola et al., 2025). Our parallelism operates over stream representations at different layers: tokens are still emitted one at a time with explicit autoregressive conditioning, and training uses the next-token likelihood. Thus, bidirectional communication among streams does not require a diffusion process over token sequences.

Architectures and kernels for cheaper decoding. Multi-query and grouped-query attention share key–value heads (Shazeer, 2019; Ainslie et al., 2023); Cross-Layer Attention shares key–value states between layers (Brandon et al., 2024). YOCO constructs a shared history with a self-decoder and reuses it in a cross-decoder (Sun et al., 2024). Block Transformer combines global processing of blocks with local token decoding (Ho et al., 2024). These approaches reduce decoding cost through head sharing, layer sharing, or hierarchical context processing. PIPET couples a shared terminal history with overlapping computation across token positions. FlashAttention improves the memory efficiency of exact attention (Dao et al., 2022); kernel efficiency is a separate consideration from the dependencies imposed by the architecture.

E Additional experimental details

This section provides the complete model, training, measurement, and architecture-selection details for the experiments in §3. Unless stated otherwise, reported losses are evaluated on a fixed held-

Component	PIPET	Standard Transformer++
Normalization	Pre-LN LayerNorm, weight and bias	Pre-norm RMSNorm, weight only
Feed-forward network	GEGLU: $a \cdot \mathrm{GELU}(b)$	SwiGLU
Input/output embeddings	Untied	Untied
Projection biases	None	None
Position encoding	Full-head RoPE, base 10^4	Full-head RoPE, base 10^4
Attention	Multi-head; $h_{\mathrm{KV}} = h_{\mathrm{Q}}$	Multi-head; $h_{\mathrm{KV}} = h_{\mathrm{Q}}$
Vocabulary size	1,000	1,000
Normalization epsilon	10^{-5}	10^{-5}
Dropout	0	0

Table 2: **Model components for PIPET and Transformer++.**

out test set shared across configurations; the validation split is used for monitoring and architecture selection.

E.1 Model and training settings

Table 2 summarizes the architectural components used by PIPET and the standard Transformer baseline. The baseline follows the standard Transformer++ implementation, while PIPET uses its own normalization and feed-forward parameterization. Except for the global-stalling study in §A.4, all PIPET experiments use $\tau = 1$, i.e., $\ell_{\mathrm{in}} = \ell_{\mathrm{out}} - 1$.

Token embeddings and the learned initial stream state in PIPET are initialized from $\mathcal{N}(0, 1)$. Attention and feed-forward weight banks use Xavier-uniform initialization with gain 1, applied separately to each layer. Transformer attention, embedding, and feed-forward input weights use a normal distribution with standard deviation $d^{-1/2}$, truncated at three standard deviations; feed-forward output weights use $d_{\mathrm{FF}}^{-1/2}$. Both language-modeling heads use $\mathcal{U}(-d^{-1/2}, d^{-1/2})$ initialization. Normalization scales are initialized to 1 and LayerNorm biases to 0. Neither family uses depth-scaled initialization.

For hidden width d, the head count is the largest integer $h \le d/64$ such that $d \bmod h = 0$ and d/h is even. Head dimensions therefore vary with model width; among the reported configurations, they range from 64 to 146. Optimization settings are listed in Table 3.

Stream states and retained history. We use Rotary Position Embeddings (RoPE) for positional encoding Su et al. (2024), note we use i instead of t for the positional embedding. Mixed-layer attention reads all existing streams. Each unfinished stream forms its key and value using the source-indexed map KV_ℓ at its current layer ℓ, including the corresponding pre-normalization. Finalized streams use KV_L and retain one shared key–value pair each. There are no standard-layer KV caches when $\tau = 1$.

Parameter matching. We count all allocated model parameters, including untied input and output embeddings. Hidden width and feed-forward dimension are adjusted to match each target parameter budget. Across the reported 4M–200M configurations, realized parameter counts differ from their targets by less than 0.1%.

All runs use an effective batch size of 64 sequences. Most runs process the full batch in a single microbatch. The 100M Transformer with $L = 1$ uses microbatches of 8 with 8 gradient-accumulation steps, while the 200M PIPET configuration uses microbatches of 32 with 2 accumulation steps to fit within GPU memory.

Each PIPET training sequence begins with fresh streams and empty attention histories. Gradients propagate through all stream updates and retained states within the sequence. Observed-token embeddings remain differentiable with respect to their parameters; teacher forcing supplies observed token IDs rather than differentiable samples from earlier logits.

Setting	Specification
Dataset	SlimPajama 6B-token subset: 48 training shards, one validation shard used for monitoring and model selection, and one test shard held out entirely; reported losses are on the test shard
Tokenizer	Byte-level BPE fitted once on the training split and shared across all runs; vocabulary 1,000; BOS/EOS IDs 0/1
Sequence construction	Documents in the pre-shuffled subset are read in corpus order within each shard, marked with BOS/EOS, concatenated, and divided into 512-token sequences without padding
Validation budget	1,048,576 tokens (2,048 sequences)
Validation frequency	Every 1,000 optimizer steps and at the final step
Reported checkpoint	Final training step
Optimizer	AdamW, $\beta = (0.9, 0.95)$, $\epsilon = 10^{-8}$
Weight decay	0.01 on all parameters, including embeddings, normalization, and the learned stream initializer
Gradient clipping	Global L_2 norm clipped at 1.0; updates skipped for non-finite norms or norms above 1,000
Effective batch size	64 sequences
Peak learning rate	10^{-4}
Learning-rate schedule	500-step linear warm-up, constant middle phase, cosine decay to zero over the final 20% of steps
Random seed	41
Precision	fp32
Activation checkpointing	Off

Table 3: **Data and optimization settings for the main suite.** Effective batch size includes micro-batching and gradient accumulation.

Target parameters (M)	Optimizer steps	Training tokens
4	2,441	80M
6	3,662	120M
15	9,155	300M
30	18,311	600M
60	36,621	1.2B
100	61,035	2.0B
200	122,070	4.0B

Table 4: **Training-token budgets.** Token counts equal optimizer steps $\times 64 \times 512$, yielding approximately 20 training tokens per parameter.

E.2 Decoding and KV-cache measurements

Both quality–runtime and quality–memory comparisons use the measurement settings in Table 5.

This benchmark measures decoder computation under the same input protocol and for the same number of emitted output distributions in both families. Fixed pseudo-random inputs make the workload independent of sampled outputs. Timing includes the language-modeling head and PIPET's pipeline startup, but excludes sampling.

Runtime per token. Let D_f denote the median pass wall time for family f and let $G = 512$ be the number of emitted output distributions per sequence. We report

$$\delta_f = \frac{D_f}{G}. \tag{27}$$

The timed pass begins with the first decoder step and ends when the G-th distribution is emitted. A Transformer executes G decoder steps. Under the $\tau = 1$ schedule, PIPET executes $G + \ell_{\text{out}} - 1$ steps because pipeline startup is included in the timed region. The first untimed pass verifies the emitted-output count and startup length.

Setting	Specification
Hardware	NVIDIA H100 NVL; single-GPU allocation on a shared node
Software	PyTorch 2.10.0+cu128, CUDA 12.8
Precision	fp32
Compilation	Compiled single-step decoders with CUDA graph capture for both families
Process isolation	One configuration per fresh process
Batch size	8 sequences
Prompt	None; each pass starts from an empty cache
Decoding budget	512 outputs per sequence for both families
Warm-up	Two untimed passes
Repetitions	Five timed passes; median pass wall time
Timer and synchronization	Pass wall time with GPU synchronization at pass boundaries and before the final 64-step segment; no per-step synchronization
Token selection	Fixed pseudo-random inputs (uniform over non-special token IDs, seed 0); logits computed, sampling excluded, no EOS stopping
Kernel fusion	Both families fuse the $Q/K/V$ projections and the feed-forward gate, so each layer issues the same number of matrix multiplications per decoded token

Table 5: **Shared configuration for decoding and KV-cache comparisons.**

Provisioned KV-cache memory. We measure the bytes allocated to KV-cache buffers for the $G = 512$-output benchmark. Buffers have capacity for exactly the positions written during the pass. Model weights, active hidden states, and temporary workspace are excluded. For each unique cache allocation u, let C_u denote its capacity in elements and q_u its bytes per element. We report

$$M_{\mathrm{KV}} = \sum_u q_u C_u. \tag{28}$$

All memory measurements use MiB, with 1 MiB $= 2^{20}$ bytes.

Every reported cache buffer is fp32 with width $hd_h = d$. A standard Transformer retains a separate key–value pair at each of its L layers, each with capacity for G positions. At $\tau = 1$, PIPET retains one shared pair for finalized streams, with capacity $G + \ell_{\mathrm{out}} - L$. Its pass spans $G + \ell_{\mathrm{out}} - 1$ decoder steps, and each stream finalizes $L - 1$ steps after introduction. Active-stream keys and values have no retained allocation: they are recomputed from stream states at each step, and these temporary projections are excluded from the metric. This provisioned-capacity measurement is distinct from the asymptotic state-growth analysis in §B.

E.3 Architecture search and recipe construction

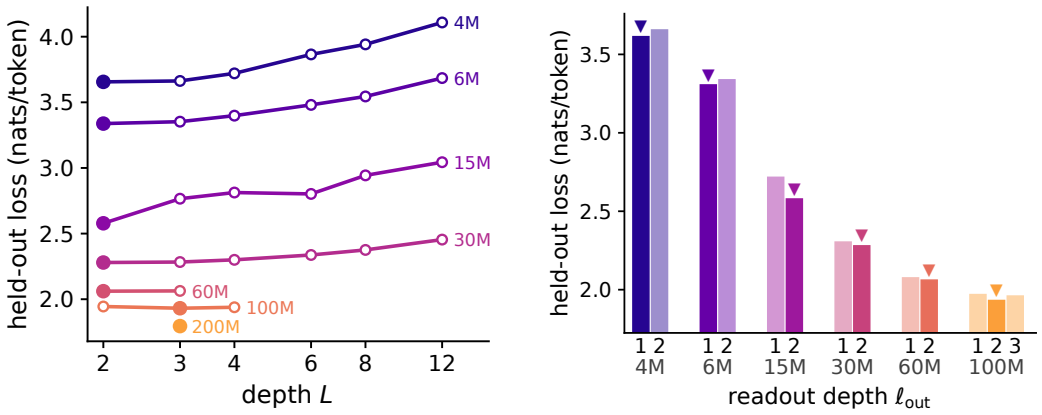
The PIPET configurations used in the main quality–efficiency comparison are selected through a staged architecture search. We first vary depth and width while fixing $(\ell_{\mathrm{in}}, \ell_{\mathrm{out}}) = (1, 2)$, then fix the selected dimensions and vary input–output layer placement. Motivated by the placement results at the smallest scales, we subsequently evaluate single-layer configurations separately.

This staged search reduces cost relative to jointly optimizing depth, width, and layer placement, but may miss better jointly tuned architectures. We use the resulting sweeps to construct a simple architecture-selection heuristic that specifies one PIPET configuration at each parameter budget.

E.3.1 Depth and width

At each parameter budget, we vary depth while adjusting hidden and feed-forward widths to maintain approximately fixed total parameter count. Training-token budgets and optimization settings are held fixed within each comparison.

The first-stage search evaluates $L \in \{2, 3, 4, 6, 8, 12\}$ at 4M–30M parameters, $L \in \{2, 3\}$ at 60M, and $L \in \{2, 3, 4\}$ at 100M, always with $(\ell_{\mathrm{in}}, \ell_{\mathrm{out}}) = (1, 2)$. At 60M and 100M, depths are evaluated in increasing order and the search is stopped after validation loss increases.



(a) Depth and width

(b) Input and output layer placement

Figure 9: **Architecture search at fixed parameter and token budgets.** Color denotes parameter budget; filled markers indicate the minimum validation loss on each curve. **(a)** Depth–width sweep with $(\ell_{\text{in}}, \ell_{\text{out}}) = (1, 2)$. **(b)** Output-layer sweep at the selected dimensions with $\ell_{\text{in}} = \ell_{\text{out}} - 1$. Individual configurations are reported in Table 7.

Across the evaluated range, the search favors relatively shallow, wide configurations. The minimum validation loss occurs at $L = 2$ through 60M parameters and at $L = 3$ for 100M parameters. The optimum is relatively flat at the larger scales: at 60M and 100M, validation loss varies by at most 0.013 nats/token across the evaluated depths.

We summarize this trend with the heuristic

$$L(N) = \max\left\{1, \text{round}\left(a + b \log_2 \frac{N}{N_0}\right)\right\}, \qquad a = 3, \quad b = 0.4, \quad N_0 = 100\text{M}. \tag{29}$$

The heuristic captures the observed preference for shallow models while allowing depth to increase gradually with parameter count. It does not select the empirical minimum at every evaluated budget. At 60M, for example, it selects $L = 3$, whose loss of 2.0634 is 0.0022 nats/token above that of the best evaluated $L = 2$ configuration. Its behavior outside the evaluated scale range remains untested.

E.3.2 Input and output layer placement

After the depth–width search, we fix model dimensions and vary the output layer while placing the input layer immediately below it:

$$\ell_{\text{out}} \in \{1, \ldots, L\}, \qquad \ell_{\text{in}} = \ell_{\text{out}} - 1. \tag{30}$$

The selected depths for this sweep are $L = 2$ at 4M–60M and $L = 3$ at 100M. Model dimensions and training-token budgets remain fixed within each sweep.

For the selected two-layer shapes, output layer 1 gives the lowest validation loss at 4M and 6M, while output layer 2 gives the lowest loss from 15M through 60M. At 100M, output layers 1, 2, and 3 achieve losses 1.9690, 1.9313, and 1.9596, respectively. We summarize this pattern with

$$\ell_{\text{out}} = \min\{2, L\}, \qquad \ell_{\text{in}} = \ell_{\text{out}} - 1. \tag{31}$$

As with the depth heuristic, the evaluated scales do not establish how the preferred input–output placement behaves at substantially larger parameter budgets.

E.3.3 Single-layer configurations

The first-stage depth search fixes $(\ell_{\text{in}}, \ell_{\text{out}}) = (1, 2)$ and therefore cannot include $L = 1$. Because output layer 1 is preferred at the smallest scales, we separately evaluate single-layer configurations with $(\ell_{\text{in}}, \ell_{\text{out}}) = (0, 1)$ from 4M through 60M parameters.

Budget	$L = 1$ loss	$L = 2$ best loss	$L = 2$ best output layer	Recipe depth
4M	**3.5552**	3.6133	1	1
6M	**3.2629**	3.3055	1	1
15M	2.6333	**2.5784**	2	2
30M	2.3112	**2.2797**	2	2
60M	2.1034	**2.0612**	2	3

Table 6: **Single-layer and two-layer PIPET configurations.** Losses are test losses. The two-layer column uses the output layer with the lowest validation loss at each budget, and bold marks the lower of the two. Architecture dimensions are reported in Table 7.

These runs use the same training protocol as the main suite: approximately 20 training tokens per parameter, seed 41, a 500-step learning-rate warm-up followed by a constant phase and cosine decay over the final 20% of training, and microbatch 64 without gradient accumulation. Hidden and feed-forward widths are adjusted to match each parameter target.

At $L = 1$, only one stream is unfinished at a time, so computation does not overlap across streams at different layers. The architecture still differs from a one-layer Transformer: each finalized stream contributes a key–value pair computed from its post-layer state through the terminal projection.

Single-layer models outperform the best evaluated two-layer configurations by 0.058 and 0.043 nats/token at 4M and 6M, respectively. The relationship reverses at larger scales: the best two-layer configurations improve over $L = 1$ by 0.055, 0.031, and 0.042 nats/token at 15M, 30M, and 60M. Thus, among the evaluated budgets, the single-layer advantage is confined to 4M and 6M, consistent with the depth heuristic selecting $L = 1$ only at these scales.

E.3.4 SELECTED CONFIGURATIONS

Combining equations (29) and (31) specifies one architecture per parameter budget, with width adjusted to match the target parameter count. At 4M and 6M, the resulting choice is $L = 1$, and we use the separately evaluated single-layer configurations, which outperform the corresponding two-layer candidates. At 15M and 30M, the recipe selects $L = 2$, while at 60M and 100M it selects $L = 3$.

At 200M, we do not repeat the architecture search. We instead extrapolate the heuristic by one additional budget point, obtaining

$$L = 3, \qquad (\ell_{\text{in}}, \ell_{\text{out}}) = (1, 2). \tag{32}$$

This is the 200M PIPET configuration reported in Figure 2. We include this single extrapolated configuration in the main comparison, but do not interpret the heuristic as establishing an architecture scaling rule beyond the range directly studied here.

Overall, the search favors substantially shallower and wider PIPET configurations than the standard Transformer configurations that achieve the lowest loss at matched parameter budgets, with preferred depth increasing only gradually over the evaluated scales.

E.4 QUALITY–RUNTIME AND QUALITY–MEMORY RESULTS

The quality–runtime and quality–memory panels in Figure 2 use a common checkpoint for each configuration's loss, runtime, and KV-cache measurements. The PIPET points follow the architecture recipe in §3.4, while Transformer points sweep depth and width independently at each parameter budget. The enlarged Transformer marker identifies the Transformer with the lowest *validation* loss within that budget's sweep; the losses plotted and tabulated are test losses. Validation and test select the same configuration at every budget and in both families, so the marker is unchanged if the test loss is used instead.

The starred rows of Table 7 give the selected PIPET configurations through 100M parameters. The single-layer models at 4M and 6M supply the first two PIPET points in Figure 2. The 200M PIPET configuration is obtained by extrapolating the recipe and has $L = 3$, $d = 2336$, $d_{\text{FF}} = 5396$, $h = 16$, placement $(1, 2)$, and 200,005,984 parameters.

Budget		L	d	d_{FF}	h	$(\ell_{\mathrm{in}}, \ell_{\mathrm{out}})$	Validation (nats/token)	Test (nats/token)
4M	⋆	1	480	989	6	$(0,1)$	**3.6155**	3.5552
		2	352	913	4	$(0,1)$	3.6655	3.6133
		2	352	913	4	$(1,2)$	3.7107	3.6559
		3	320	632	5	$(1,2)$	3.7125	3.6629
		4	288	533	4	$(1,2)$	3.7630	3.7206
		6	224	543	2	$(1,2)$	3.9066	3.8650
		8	192	503	3	$(1,2)$	3.9817	3.9414
		12	160	411	2	$(1,2)$	4.1533	4.1084
6M	⋆	1	576	1459	9	$(0,1)$	**3.3453**	3.2629
		2	448	1076	7	$(0,1)$	3.3902	3.3055
		2	448	1076	7	$(1,2)$	3.4189	3.3380
		3	384	872	6	$(1,2)$	3.4319	3.3526
		4	352	695	4	$(1,2)$	3.4735	3.3985
		6	288	613	4	$(1,2)$	3.5484	3.4805
		8	256	518	4	$(1,2)$	3.6124	3.5446
		12	192	539	3	$(1,2)$	3.7477	3.6848
15M	⋆	2	736	1712	8	$(1,2)$	**2.7203**	2.5784
		1	960	2299	15	$(0,1)$	2.7525	2.6333
		2	736	1712	8	$(0,1)$	2.8300	2.7165
		3	608	1504	8	$(1,2)$	2.8782	2.7657
		4	544	1268	8	$(1,2)$	2.9261	2.8128
		6	448	1076	7	$(1,2)$	2.9196	2.8022
		8	384	983	6	$(1,2)$	3.0439	2.9434
		12	320	792	5	$(1,2)$	3.1474	3.0438
30M	⋆	2	1056	2464	16	$(1,2)$	**2.4256**	2.2797
		1	1408	3148	22	$(0,1)$	2.4405	2.3112
		2	1056	2464	16	$(0,1)$	2.4390	2.3037
		3	896	2003	14	$(1,2)$	2.4303	2.2833
		4	768	1871	12	$(1,2)$	2.4472	2.3001
		6	640	1532	10	$(1,2)$	2.4876	2.3371
		8	576	1245	9	$(1,2)$	2.5290	2.3756
		12	480	999	6	$(1,2)$	2.6075	2.4543
60M	⋆	3	1280	2851	20	$(1,2)$	2.2036	2.0634
		1	2016	4548	28	$(0,1)$	2.2299	2.1034
		2	1504	3556	16	$(0,1)$	2.2062	2.0753
		2	1504	3556	16	$(1,2)$	**2.2009**	2.0612
100M	⋆	3	1632	3864	24	$(1,2)$	**2.0612**	1.9313
		2	1984	4428	31	$(1,2)$	2.0745	1.9445
		3	1632	3864	24	$(0,1)$	2.0921	1.9690
		3	1632	3864	24	$(2,3)$	2.0900	1.9596
		4	1440	3339	20	$(1,2)$	2.0706	1.9387

Table 7: **All PIPET architecture-search runs at 4M–100M.** Within each budget, the starred row is the configuration selected by the architecture recipe in §3.4; remaining rows belong to the depth or placement searches. Bold marks the lowest validation loss at each budget; configuration selection uses validation loss only, and test loss is reported for reference. At every budget the two agree on which configuration is best. All losses are in nats per scored token, and all runs use the corresponding training-token budget in Table 4.

For Transformer evaluation, candidate depths at 4M, 6M, 15M, 30M, 60M, and 100M are

$$L \in \{1, 2, 3, 4, 6, 8, 12, 16\}.$$

Dimensions are adjusted to match the same total parameter and training-token budgets as PIPET. The 200M comparison uses $L \in \{4, 6, 8, 12, 16, 24\}$; all six Transformer configurations are matched to within 0.01% of the PIPET parameter count.

Table 8: **Quality, runtime per token, and KV-cache measurements.** Configurations correspond to the points in Figure 2. At each budget, PIPET follows the architecture recipe and Transformer depth is swept; bold marks the lowest test loss within each family. The matched-output benchmark uses batch size 8 and 512 outputs per sequence (§E.2).

Budget	Model	L	d	d_{FF}	h	Test loss (nats/token)	δ_f (ms)	KV (MiB)
4M	PIPET	1	480	989	6	**3.5552**	0.179	15.0
	Transformer	1	448	1712	7	3.7172	0.136	14.0
		2	320	1323	5	3.5975	0.249	20.0
		3	288	936	4	**3.5810**	0.322	27.0
		4	256	793	4	3.6110	0.470	32.0
		6	192	790	3	3.6444	0.662	36.0
		8	160	744	2	3.6899	0.769	40.0
		12	160	425	2	3.7670	1.099	60.0
		16	128	438	2	3.8629	1.619	64.0
6M	PIPET	1	576	1459	9	**3.2629**	0.209	18.0
	Transformer	1	576	2037	9	3.3347	0.163	18.0
		2	416	1515	4	3.2984	0.260	26.0
		3	352	1202	4	**3.2555**	0.349	33.0
		4	288	1185	4	3.2731	0.446	36.0
		6	256	849	4	3.3094	0.698	48.0
		8	224	733	2	3.3224	0.821	56.0
		12	192	556	3	3.3849	1.272	72.0
		16	160	526	2	3.4443	1.461	80.0
15M	PIPET	2	736	1712	8	**2.5784**	0.240	23.0
	Transformer	1	928	3483	8	2.9081	0.209	29.0
		2	672	2490	8	2.8608	0.302	42.0
		3	544	2116	8	2.5994	0.406	51.0
		4	480	1797	6	**2.5625**	0.497	60.0
		6	384	1546	6	2.6004	0.779	72.0
		8	352	1222	4	2.6030	0.878	88.0
		12	288	1007	4	2.6970	1.250	108.0
		16	256	837	4	2.7827	1.804	128.0
30M	PIPET	2	1056	2464	16	**2.2797**	0.268	33.0
	Transformer	1	1344	4981	21	2.6897	0.291	42.0
		2	960	3594	15	2.4043	0.403	60.0
		3	800	2877	10	2.3075	0.495	75.0
		4	704	2445	11	2.2903	0.627	88.0
		6	576	2014	9	**2.2711**	0.862	108.0
		8	480	1880	6	2.2737	0.972	120.0
		12	416	1392	4	2.2873	1.406	156.0
		16	352	1264	4	2.2954	1.736	176.0

Continued on next page

Table 8: Quality, runtime per token, and KV-cache measurements (continued).

Budget	Model	L	d	d_{FF}	h	Test loss (nats/token)	δ_f (ms)	KV (MiB)
60M	PIPET	3	1280	2851	20	**2.0634**	0.402	39.9
	Transformer	1	1920	7189	30	2.5313	0.398	60.0
		2	1376	5099	16	2.1388	0.488	86.0
		3	1120	4236	16	2.0944	0.604	105.0
		4	992	3550	8	2.0711	0.729	124.0
		6	800	2988	10	2.0409	0.961	150.0
		8	704	2528	11	2.0381	1.227	176.0
		12	576	2069	9	**2.0367**	1.699	216.0
		16	480	1922	6	2.0415	1.918	240.0
100M	PIPET	3	1632	3864	24	**1.9313**	0.539	50.9
	Transformer	1	2496	9359	39	2.4542	0.506	78.0
		2	1792	6577	28	2.0235	0.703	112.0
		3	1472	5363	23	1.9772	0.849	138.0
		4	1280	4636	20	1.9502	0.964	160.0
		6	1024	3948	16	1.9149	1.171	192.0
		8	896	3372	14	1.9096	1.449	224.0
		12	736	2737	8	1.8982	1.793	276.0
		16	640	2360	10	**1.8961**	2.350	320.0
200M	PIPET	3	2336	5396	16	**1.7952**	0.780	72.9
	Transformer	4	1792	6744	28	1.8127	1.366	224.0
		6	1472	5474	23	1.7767	1.678	276.0
		8	1280	4720	20	1.7620	1.890	320.0
		12	1056	3797	16	1.7480	2.168	396.0
		16	896	3413	14	1.7407	2.864	448.0
		24	736	2764	8	**1.7404**	3.548	552.0

F TOKEN-BUDGET SCALING

The main experiments train recipe-selected architectures with approximately 20 tokens per parameter, following Hoffmann et al. (2022). We investigate two questions: whether scaling at this fixed ratio predicts performance at larger model sizes, and whether a different allocation of parameters and training tokens yields lower loss at the same compute budget. We address these questions with a scaling fit along the allocation recipe and a separate training-token sweep. All losses use the same test split and evaluation protocol as the main experiments.

F.1 SCALING ALONG THE ALLOCATION RECIPE

Along the recipe, the training-token budget satisfies $D \approx 20N$, where N is the number of parameters and D is the number of training tokens. We fit the one-dimensional scaling law

$$\mathcal{L}(N) = E + K\, N^{-\gamma} \tag{33}$$

to the six recipe-selected models at $N \in \{4, 6, 15, 30, 60, 100\}\mathrm{M}$. The fitted law is

$$\mathcal{L}(N) = 1.4457 + 2409.6\, N^{-0.4624}, \tag{34}$$

where N is measured in individual parameters. The root-mean-square fitting error is 0.025 nats/token.

We withhold the 200M model from the fit to evaluate extrapolation. Using the unrounded fit coefficients, the predicted loss is 1.7955, compared with the measured loss of 1.7952, an absolute error of 0.0003 nats/token (Figure 10a). This provides an out-of-fit check of scaling along the recipe, where model size and training-token budget increase together.

F.2 TRAINING-TOKEN BUDGET SWEEP

To examine the allocation of training compute, we conduct 17 additional runs at 4M, 6M, 15M, and 30M parameters. Within each parameter budget, we hold the recipe-selected architecture fixed and vary the number of training steps, and hence the number of training tokens (Table 9). All configurations use $\tau = 1$.

Loss decreases monotonically with the training-token budget at every evaluated model size. At 4M and 6M, increasing the budget from 40 to 80 tokens per parameter reduces loss by 0.220 and 0.178 nats/token, respectively.

Matched-compute comparisons favor allocating more tokens to smaller models in the evaluated cases. At $C \approx 2.70 \times 10^{16}$ FLOPs, the 15M model trained with 20 tokens per parameter achieves a loss of 2.5795, compared with 2.7194 for the 30M model trained with 5 tokens per parameter. At $C \approx 5.40 \times 10^{16}$ FLOPs, the 15M model trained with 40 tokens per parameter achieves 2.4034, compared with 2.4340 for the 30M model trained with 10 tokens per parameter.

To characterize this tradeoff across scales, we fit the two-term law of Hoffmann et al. (2022), $\mathcal{L}(N, D) = E + AN^{-\alpha} + BD^{-\beta}$, to the 17 sweep runs together with the recipe-selected runs from 4M to 200M. Unlike the preceding extrapolation study, this fit includes the 200M result. The larger recipe runs extend the range of model sizes, while the sweep varies the token budget independently of model size. At 15M and 30M, the sweep runs with 20 tokens per parameter agree with their main-suite counterparts to within 0.001 nats/token.

The resulting fit is

$$\mathcal{L}(N, D) = 1.0972 + 805.1\, N^{-0.4443} + 278.6\, D^{-0.2869}, \tag{35}$$

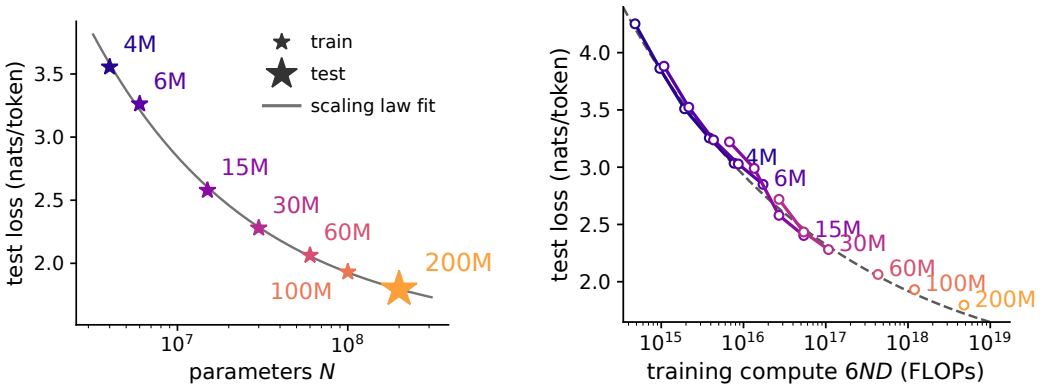
where N and D are measured in individual parameters and tokens. Its root-mean-square fitting error is 0.048 nats/token.

Using the training-compute proxy $C = 6ND$ following Kaplan et al. (2020), we minimize the fitted loss over the allocation of parameters and tokens at each fixed compute budget. This gives

$$N_{\text{opt}} \propto C^{0.39}, \qquad D_{\text{opt}} \propto C^{0.61}. \tag{36}$$

The corresponding minimum predicted loss is shown as the dashed curve in Figure 10b. At the compute budgets of the 4M, 100M, and 200M recipe runs, the fitted optimal allocations use approximately 22, 88, and 119 tokens per parameter, respectively. Each allocation changes both model size and training-token budget while keeping compute fixed.

Thus, the fit favors increasing the number of tokens per parameter with compute, rather than maintaining a fixed ratio of 20. The matched-compute comparisons support this direction within the sweep; the optimal allocations at larger compute budgets remain predictions of the fitted law.



(a) Scaling at 20 tokens per parameter

(b) Loss versus training compute

Figure 10: **Scaling with model size and training-token budget.** Color denotes parameter budget. **(a)** Test loss of recipe-selected models trained with approximately 20 tokens per parameter. The gray curve is fitted to the 4M–100M models; the enlarged star marks the withheld 200M model. **(b)** Test loss against the training-compute proxy $C = 6ND$. Points include all 17 token-sweep runs and the recipe-selected runs from 4M to 200M. Solid lines connect sweep runs at fixed model size. The dashed curve shows the minimum loss predicted by equation (35) when model size and token budget are jointly optimized at fixed compute.

Budget	L	d	d_{FF}	Tokens/param	Steps	Tokens (M)	Test loss (nats/token)
4M	1	480	989	5	610	20	4.2518
				10	1,221	40	3.8622
				20	2,442	80	3.5099
				40	4,883	160	3.2549
				80	9,766	320	3.0345
6M	1	576	1459	5	916	30	3.8819
				10	1,831	60	3.5237
				20	3,662	120	3.2380
				40	7,324	240	3.0279
				80	14,648	480	2.8498
15M	2	736	1712	5	2,289	75	3.2209
				10	4,578	150	2.9893
				20	9,155	300	2.5795
				40	18,310	600	2.4034
30M	2	1056	2464	5	4,578	150	2.7194
				10	9,156	300	2.4340
				20	18,311	600	2.2807

Table 9: **Training-token sweep for PIPET.** Loss is on the test split. Model dimensions are fixed within each parameter budget. Token counts and tokens per parameter are rounded; each step processes 64×512 tokens.