LOOPED TRANSFORMERS AS OPTIMIZERS

Anonymous authors
Paper under double-blind review

ABSTRACT

Looped Transformers provide a parameter-efficient approach to depth scaling by repeatedly applying shared Transformer blocks. Recent reasoning models have likewise highlighted the value of scaling test-time computation through longer computation trajectories. However, the principles for designing effective loop transitions remain poorly understood. We view the looped hidden state as a fast weight[1] that is updated throughout the depth. We formulate loop transitions as local gradient-based updates, with recurrent blocks predicting implicit targets at each depth. Our framework derives loop transitions in closed form from a projection, a local objective and an optimizer update rule. Mapping representative loop transitions into this framework reveals mismatches between their transitions and projections. We first align the input maps of existing transitions. We then derive OperLoop, which combines explicit weight decay, adaptive step size and a delta objective. The aligned variants reduce training loss and improve average commonsense accuracy. OperLoop improves average generative performance over the compared looped and non-looped baselines under matched training FLOPs. These results support the framework's usefulness for loop design. We extend the analysis to additional loop models and outline a roadmap for future loop transition design.

1 INTRODUCTION

The scaling of modern large language models (LLMs) follows an empirical principle (Sutton, 2019): jointly increasing model size, training data and computation generally improves performance (Kaplan et al., 2020). In conventional Transformers, increasing depth by stacking independently parameterized blocks also increases the number of unique parameters (Vaswani et al., 2017). Greater depth can improve performance on challenging reasoning tasks (Rodkin et al., 2026), but additional parameters can yield diminishing returns under data constraints (Muennighoff et al., 2023). Looped Transformers address this tension by decoupling depth from parameter scaling. They repeatedly reuse shared blocks without proportionally increasing the unique parameters (Dehghani et al., 2019; Lan et al., 2020). The repeated computation increases the effective depth, providing additional latent refinement that can enhance the model's reasoning ability (Saunshi et al., 2025).

Looped Transformer designs vary along two complementary dimensions: *where to loop* and *how to loop*. For *where to loop*, recurrence varies in both depth and feedback location. It may span the entire model (Zhu et al., 2025), middle blocks (Geiping et al., 2025), or individual layers (Gao et al., 2026), and feedback is drawn from different token or layer locations (Mozer et al., 2026; Wang et al., 2026). For *how to loop*, methods differ in their recurrent-state update mechanisms, using state decay and source injection for stability (Prairie et al., 2026), or an expanded state with parallel streams (Xie et al., 2025) to improve the parameter compute trade-off (Zeitoun et al., 2026).

Despite these diverse designs, existing *where* and *how* designs are often coupled to specific architectures. Regardless of loop placement, each iteration must transform the state produced at one step into the state used at the next. We refer to this transformation as the *recurrent transition*. The transition determines how the state is read, retained and updated, governing representational refinement (Hegazy et al., 2026) and dynamical stability (Prairie et al., 2026). However, existing transition mechanisms remain closely coupled to their underlying architectures, making it difficult to compare them or derive general principles for improving recurrent state evolution.

[1] Our use of the term *fast weight* is inspired by Schlag et al. (2021)'s work, which interprets the recurrent state as dynamically updated weights that evolve across the sequence. The title of this work pays homage to Li et al. (2018); Schlag et al. (2021); Dao & Gu (2024).

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.

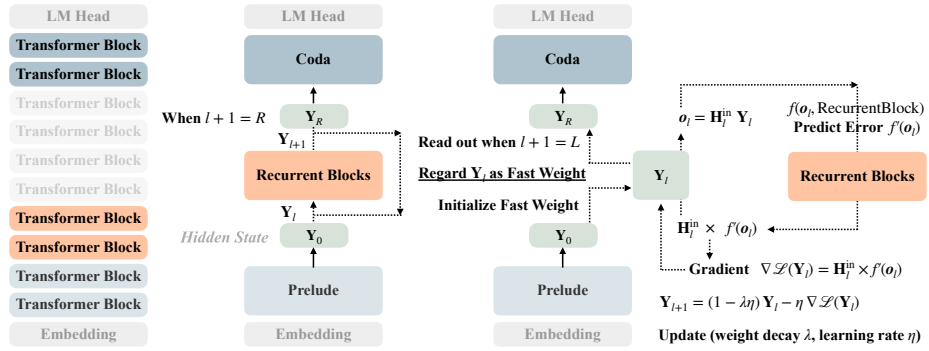


Figure 1: Optimization view of Looped Transformers. (a) A standard Transformer uses independently parameterized blocks. (b) A middle-looped Transformer repeatedly applies shared blocks; the Prelude and Coda may be identity maps in a fully looped model. (c) The recurrent state is treated as a fast weight. An input map $\mathbf{H}_l^{\text{in}}$ reads out the state, and the shared blocks predict an implicit target. The local objective and optimizer update rule determine the state update. Setting $\mathbf{H}_{\text{in}} = \mathbf{I}$, $\lambda = \eta = 1$ and $f'(\mathbf{o}_l) = -\,\text{RecurrentBlocks}(\boldsymbol{o}_l)$ recovers the vanilla loop in (b).

In this work, we propose a unified optimization framework for analyzing and designing loop transitions (Figure 1). The framework treats the recurrent state as a *fast weight* updated across the depth and connects three components: projection, objective, and optimizer update rule. The projection reads out the fast weight, while shared recurrent blocks implicitly predict the target. The objective relates the projection to a target. A loop transition is therefore a fast weight update governed by the optimizer rule and the objective's gradient. Mapping several loop models into this framework reveals mismatches between their transitions and projections. Aligning these maps improves performance, supporting the framework's design constraints. We then derive OperLoop using this optimization framework. Under matched training FLOPs, OperLoop improves average generative performance over the evaluated looped and non-looped baselines. Together, these results suggest that the framework provides useful guidance for improving existing transitions and designing new ones. Finally, we analyze additional loop models and outline a roadmap for future transition design. Our contributions are:

- **Unified Framework.** We develop a unified optimization framework that treats recurrent states as fast weights updated across the depth. It derives the loop transitions in closed form from a projection, a local objective, and an optimizer update rule.
- **Input-Map Alignment.** We use the framework to identify input-map mismatches and construct aligned variants of existing looped models. Their improved performance provides empirical support for the framework's design constraints.
- **Optimizer-Guided Loop Design.** We derive OperLoop from the framework using a delta objective and adaptive step-size control. Matched compute budget evaluations and component ablations further support the framework's ability to guide effective loop-transition design.
- **Extended Analysis and Roadmap.** We apply the framework to additional loop models, identifying shared objective forms and recurring input-map mismatches. This analysis informs a roadmap within the framework to guide future Looped Transformers architecture design.

2 Preliminaries

Notation. Let $\mathbf{X} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N)$ be a sequence of N tokens and $\mathbf{H} \in \mathbb{R}^{N\times d}$ its hidden states, where d is the model width. The symbols k and l denote the depth and loop indices, respectively. We use L, K, and R for the depth of a conventional Transformer, the number of distinct blocks in the shared recurrent blocks, and the number of recurrent steps, respectively. Unless stated otherwise, bold uppercase and lowercase letters denote matrices and vectors, respectively. We use function composition notation, writing $\mathcal{F}_3(\mathcal{F}_2(\mathcal{F}_1(x)))$ compactly as $\mathcal{F}_3 \circ \mathcal{F}_2 \circ \mathcal{F}_1(x)$.

Standard Transformers. A standard Transformer increases its effective depth by stacking independently parameterized blocks (Vaswani et al., 2017). Given a hidden state $\mathbf{H}^{(k)}$, a pre-norm Transformer block applies an attention sublayer followed by an FFN sublayer:

$$\mathbf{H}^{(k+1)} = \widetilde{\mathbf{H}}^{(k)} + \mathrm{FFN}_{\theta_k^{\mathrm{FFN}}}\left(\mathrm{LN}(\widetilde{\mathbf{H}}^{(k)})\right), \qquad \widetilde{\mathbf{H}}^{(k)} = \mathbf{H}^{(k)} + \mathrm{Attn}_{\theta_k^{\mathrm{Attn}}}\left(\mathrm{LN}(\mathbf{H}^{(k)})\right). \tag{1}$$

For depth index $k = 0, \ldots, L-1$, let $\theta_k := (\theta_k^{\mathrm{Attn}}, \theta_k^{\mathrm{FFN}})$ and $\mathbf{H}^{(k+1)} := \mathcal{F}_{\theta_k}(\mathbf{H}^{(k)})$ for simplicity. Given $\mathbf{H}^{(0)} = \mathrm{Embedding}(\mathbf{X})$, stacking these blocks yields the final hidden state in Eq. 2:

$$\mathbf{H}^{(L)} = \mathcal{F}_{\theta_{L-1}} \circ \mathcal{F}_{\theta_{L-2}} \circ \cdots \circ \mathcal{F}_{\theta_1} \circ \mathcal{F}_{\theta_0}(\mathbf{H}^{(0)}). \tag{2}$$

Looped Transformers. Looped Transformers increase effective depth by reusing the same blocks without introducing additional parameters (Dehghani et al., 2019). Let $\mathrm{Blocks}_\theta := \mathcal{F}_{\theta_{i+K-1}} \circ \cdots \circ \mathcal{F}_{\theta_{i+1}} \circ \mathcal{F}_{\theta_{i+0}}$, $\mathrm{Prelude}_\varphi := \mathcal{F}_{\theta_{i-1}} \circ \cdots \circ \mathcal{F}_{\theta_0}$ and $\mathrm{Coda}_\phi := \mathcal{F}_{\theta_{L-1}} \circ \cdots \circ \mathcal{F}_{\theta_{L-j}}$, where i and j denote the numbers of Prelude and Coda blocks, respectively. The middle-looped architecture (Geiping et al., 2025) yields the final hidden state in Eq. 3:

$$\mathbf{H}^{(L)} = \mathrm{Coda}_\phi \circ \mathrm{Blocks}_\theta \circ \cdots \circ \mathrm{Blocks}_\theta \circ \mathrm{Prelude}_\varphi(\mathbf{H}^{(0)}). \tag{3}$$

The shared Blocks_θ is iteratively applied R times to update the latent hidden state. The Prelude embeds the input representation into the latent space, while the Coda decodes the final latent state into the output (Geiping et al., 2025). For a fully-looped model (Zhu et al., 2025), both the Prelude and Coda degenerate to the identity. In this work, we focus on the middle-loop architecture, which retains separate prelude and coda modules. Additional related work is discussed in Appendix A.

3 A Unified Framework for Looped Transformers

This section develops a unified framework for Looped Transformers by focusing on the recurrent state transitions induced by shared recurrent blocks. We first characterize the transition structure of representative architectures, then formulate loop transitions from an optimization perspective, and finally map these transitions into the resulting framework, as summarized in Table 1.

3.1 A Transition View of Looped Transformers

Vanilla State Transition. The vanilla loop updates the recurrent state through a simple transition. Let $\boldsymbol{y}_l \in \mathbb{R}^d$ denote the recurrent state and $\boldsymbol{b} \in \mathbb{R}^d$ an optional bias. Its transition is given by Eq. 4. Given the prelude output $\boldsymbol{e} = \mathrm{Prelude}_\phi(\mathbf{H}^{(0)}) \in \mathbb{R}^d$, the initial state is either $\boldsymbol{y}_0 = \boldsymbol{e}$ or $\boldsymbol{y}_0 \sim \mathcal{N}(\mu, \sigma)$. Under this abstraction, Ouro (Zhu et al., 2025) omits bias ($\boldsymbol{b} = \mathbf{0}$), whereas Huginn (Geiping et al., 2025) uses injection ($\boldsymbol{b} = \boldsymbol{e}$). The final state is read out as $\mathbf{H}^{(L)} = \mathrm{Coda}_\phi(\boldsymbol{y}_R)$.

$$\boldsymbol{y}_{l+1} = \mathrm{Blocks}_\theta(\boldsymbol{y}_l + \boldsymbol{b}). \tag{4}$$

Structured State Transition. Parcae (Prairie et al., 2026) augments the vanilla transition with a structured state space model inspired by Gu et al. (2021). Its transition[2] is given by Eq. 5. For learnable parameters Δ, $\mathbf{A} \in \mathbb{R}^d$ and $\mathbf{B} \in \mathbb{R}^{d \times d}$, the diagonal operator $\bar{\mathbf{A}} = \mathrm{Diag}(\exp(-\Delta\mathbf{A}))$ decays the state $\boldsymbol{y}_l \in \mathbb{R}^d$ and promotes $\rho(\bar{\mathbf{A}}) < 1$ for stable dynamics. Following Huang (2026), the input gain is $\bar{\mathbf{B}} = \mathrm{Diag}(\frac{1-\exp(-\Delta\mathbf{A})}{\mathbf{A}})\mathbf{B}$. The final state is read out as $\mathbf{H}^{(L)} = \mathrm{Coda}_\phi(\bar{\mathbf{C}}\,\boldsymbol{y}_R)$.

$$\boldsymbol{y}_{l+1} = \mathrm{Blocks}_\theta(\bar{\mathbf{A}}\boldsymbol{y}_l + \bar{\mathbf{B}}\boldsymbol{e}). \tag{5}$$

Expanded State Transition. HyperLoop (Zeitoun et al., 2026) expands the recurrent state to r streams as $\mathbf{Y}_{l+1} \in \mathbb{R}^{r \times d}$. Its transition is given by Eq. 6. The state-dependent transformations $\mathbf{H}_l^{\mathrm{pre}}, \mathbf{H}_l^{\mathrm{post}} \in \mathbb{R}^{1 \times r}$ map between the streams and the d-dimensional block input, while the matrix $\mathbf{H}_l^{\mathrm{res}} \in \mathbb{R}^{r \times r}$ mixes the carried state. The final state is read out as $\mathbf{H}^{(L)} = \mathrm{Coda}_\phi(\mathrm{Mean}_r(\mathbf{Y}_R))$. Here $\boldsymbol{b}_l \in \mathbb{R}^d$ is an optional independent bias, which we omit for notational simplicity.

$$\mathbf{Y}_{l+1} = \mathbf{H}_l^{\mathrm{res}}\mathbf{Y}_l + (\mathbf{H}_l^{\mathrm{post}})^\top \left(\mathrm{Blocks}_\theta\left(\mathbf{H}_l^{\mathrm{pre}}\mathbf{Y}_l\right) + \mathbf{b}_l\right). \tag{6}$$

Despite their architectural differences, these models share a common pattern: they read the recurrent state, transform it using the shared block Blocks_θ and write the result back as the next state.

[2]The original paper writes the recurrent update as $\boldsymbol{y}_{l+1} = \bar{\mathbf{A}}\boldsymbol{y}_l + \bar{\mathbf{B}}\boldsymbol{e} + \bar{\mathcal{R}}(\boldsymbol{y}_l, \boldsymbol{e})$, which represents the computation inside the shared blocks. We abstract the shared blocks as Blocks_θ. Here $\Delta, \mathbf{A} > 0$, $\bar{\mathbf{C}} \in \mathbb{R}^{d \times d}$.

Table 1: Comparison of loop transitions under the optimizer update rule in Eq. 7. The lower rows identify the decay coefficient (λ), step size (η), input map ($\boldsymbol{x}$), output error (f'), and recurrent state ($\mathbf{W}$). Red marks mismatches between the update's implied input map and the projection's map. Aligning the transition with the input map used in the projection improves performance (§ 5.1).

Models	**Ouro** (Zhu et al., 2025)	**Huginn** (Geiping et al., 2025)	**Parcae** (Prairie et al., 2026)	**HyperLoop** (Zeitoun et al., 2026)
Original loop transitions				
Loop	$\boldsymbol{y}_{l+1} = \text{Blocks}_\theta(\boldsymbol{o}_l)$	$\boldsymbol{y}_{l+1} = \text{Blocks}_\theta(\boldsymbol{o}_l)$	$\boldsymbol{y}_{l+1} = \text{Blocks}_\theta(\boldsymbol{o}_l)$	$\mathbf{Y}_{l+1} = \mathbf{H}_l^{\text{res}}\mathbf{Y}_l +$ $(\mathbf{H}_l^{\text{post}})^\top \text{Blocks}_\theta(\boldsymbol{o}_l)$
Projection	$\boldsymbol{o}_l = \boldsymbol{y}_l$	$\boldsymbol{o}_l = \boldsymbol{y}_l + \mathbf{e}$	$\boldsymbol{o}_l = \bar{\mathbf{A}}\boldsymbol{y}_l + \bar{\mathbf{B}}\mathbf{e}$	$\boldsymbol{o}_l = \mathbf{H}_l^{\text{pre}}\mathbf{Y}_l$
Optimization mapping to $\mathbf{W}_{i+1} = (1-\lambda\eta)\,\mathbf{W}_i - \eta\,(\boldsymbol{x}^\top f')$				
λ	1	1	1	$\mathbf{I} - \mathbf{H}_l^{\text{res}}$
η	1	1	1	1
$\boldsymbol{x}$	$\mathbf{I}$	$\mathbf{I}$	$\mathbf{I}$(Mismatch with $\bar{\mathbf{A}}$)	$\mathbf{H}_l^{\text{post}}$(Mismatch with $\mathbf{H}_l^{\text{pre}}$)
$f'(\boldsymbol{o}_l)$	$-\text{Blocks}_\theta(\boldsymbol{o}_l)$	$-\text{Blocks}_\theta(\boldsymbol{o}_l)$	$-\text{Blocks}_\theta(\boldsymbol{o}_l)$	$-\text{Blocks}_\theta(\boldsymbol{o}_l)$
$\mathbf{W}$	$\boldsymbol{y}_l \in \mathbb{R}^d$	$\boldsymbol{y}_l \in \mathbb{R}^d$	$\boldsymbol{y}_l \in \mathbb{R}^d$	$\mathbf{Y}_l \in \mathbb{R}^{r\times d}$

3.2 An Optimization View of Looped Transformers

To build a connection between the looped transition and optimization, we interpret the recurrent state as a fast weight updated across the depth. Then we further express loop transitions in an optimization form. This formulation reveals the implicit target predicted by the recurrent blocks.

We first consider a toy optimization example. Let $\boldsymbol{x} \in \mathbb{R}^{1\times d_1}$, $\boldsymbol{b} \in \mathbb{R}^{1\times d_2}$ and $\mathbf{W} \in \mathbb{R}^{d_1\times d_2}$ be an input, bias and weights, respectively. The output is $\boldsymbol{o} = \boldsymbol{x}\mathbf{W} + \boldsymbol{b} \in \mathbb{R}^{1\times d_2}$. For the local objective $\mathcal{L} := f(\boldsymbol{o})$, the output error is $\frac{\partial \mathcal{L}}{\partial \boldsymbol{o}} := f'(\boldsymbol{o}) \in \mathbb{R}^{1\times d_2}$. Then the gradient with respect to the weight is $\frac{\partial \mathcal{L}}{\partial \mathbf{W}} = \boldsymbol{x}^\top f'(\boldsymbol{o}) \in \mathbb{R}^{d_1\times d_2}$. For step size η and weight decay λ, the optimization update is:

$$\mathbf{W}_{i+1} = \mathbf{W}_i - \eta\,(\nabla_{\mathbf{W}}\mathcal{L} + \lambda\,\mathbf{W}_i) = (1-\eta\lambda)\,\mathbf{W}_i - \eta\,(\boldsymbol{x}^\top f'(\boldsymbol{o})). \tag{7}$$

To relate loop transitions to optimization, we interpret the recurrent state $\mathbf{Y}_l \in \mathbb{R}^{d_1\times d_2}$ as a fast weight playing the role of $\mathbf{W}$ in Eq. 7. The dimensions d_1, d_2 denote its row and column dimensions, which are specified for each architecture. Proposition 1 gives the resulting optimizer-form transition.

Proposition 1 (Optimization interpretation of loop transitions)**.** *Let $\mathbf{Y}_l \in \mathbb{R}^{d_1\times d_2}$ denote the fast weight updated across recurrent depth l. Let $\mathbf{H}_l^{\text{in}} \in \mathbb{R}^{m\times d_1}$ be an adaptive input map and $\boldsymbol{b}_l \in \mathbb{R}^{m\times d_2}$ a bias. For a projection $\boldsymbol{o}_l = \mathbf{H}_l^{\text{in}}\mathbf{Y}_l + \boldsymbol{b}_l$ with the local objective $\mathcal{L} := f(\boldsymbol{o}_l)$, the loop transition can be written in the gradient-based optimization form shown in Eq. 8:*

$$\mathbf{Y}_{l+1} = (\mathbf{I}_{d_1} - \eta_l\mathbf{\Lambda}_l)\,\mathbf{Y}_l - \eta_l\left((\mathbf{H}_l^{\text{in}})^\top f'(\boldsymbol{o}_l)\right), \tag{8}$$

where η_l is an adaptive step size and $\mathbf{\Lambda}_l \in \mathbb{R}^{d_1\times d_1}$ is a decay operator representing weight decay.

Proof Sketch. From the optimizer perspective, the general update in Eq. 8 contains an implicit forward computation and a local objective evaluation. In particular, the fast weight projection is computed as $\boldsymbol{o}_l = \mathbf{H}_l^{\text{in}}\mathbf{Y}_l + \boldsymbol{b}_l \in \mathbb{R}^{m\times d_2}$, where $\mathbf{H}_l^{\text{in}} \in \mathbb{R}^{m\times d_1}$ is the input map and $\boldsymbol{b}_l \in \mathbb{R}^{m\times d_2}$ is an optional bias. To update the fast weight $\mathbf{Y}_l$, we consider gradient-based optimization with decay:

$$\mathbf{Y}_{l+1} = \mathbf{Y}_l - \eta_l\,(\nabla_{\mathbf{Y}_l}\mathcal{L} + \mathbf{\Lambda}_l\mathbf{Y}_l)\,. \tag{9}$$

The partial derivative of $\mathcal{L}$ is $\frac{\partial \mathcal{L}}{\partial \boldsymbol{o}_l} = f'(\boldsymbol{o}_l) \in \mathbb{R}^{m\times d_2}$. We then obtain the gradient:

$$\nabla_{\mathbf{Y}_l}\mathcal{L} = \frac{\partial \mathcal{L}}{\partial \mathbf{Y}_l} = (\mathbf{H}_l^{\text{in}})^\top \frac{\partial \mathcal{L}}{\partial \boldsymbol{o}_l} = (\mathbf{H}_l^{\text{in}})^\top f'(\boldsymbol{o}_l). \tag{10}$$

The gradient has shape $d_1 \times d_2$, matching that of the fast weight. The matrix product in Eq. 10 reduces to an outer product when $m = 1$. Substituting Eq. 10 into Eq. 9 yields:

$$\mathbf{Y}_{l+1} = \mathbf{Y}_l - \eta_l\left((\mathbf{H}_l^{\text{in}})^\top f'(\boldsymbol{o}_l) + \mathbf{\Lambda}_l\mathbf{Y}_l\right) = (\mathbf{I}_{d_1} - \eta_l\mathbf{\Lambda}_l)\,\mathbf{Y}_l - \eta_l\left((\mathbf{H}_l^{\text{in}})^\top f'(\boldsymbol{o}_l)\right). \tag{11}$$

Finally, Eq. 11 recovers the formulation in Eq. 8, as stated in Proposition 1. □

Let $\boldsymbol{t}_l = \text{Blocks}_\theta(\boldsymbol{o}_l)$ be the implicit target predicted by the shared recurrent blocks. For the local algebraic derivation, we hold $\boldsymbol{t}_l$ fixed when differentiating the implicit negative inner product objective $\mathcal{L}_l = -\langle \boldsymbol{o}_l, \boldsymbol{t}_l \rangle$, giving $f'(\boldsymbol{o}_l) = \nabla_{\boldsymbol{o}_l} \mathcal{L}_l = -\boldsymbol{t}_l = -\text{Blocks}_\theta(\boldsymbol{o}_l)$. The fixed-target convention is used only to derive the closed-form transition algebraically. It does not imply that gradients are stopped during training. Under this hypothesis and Proposition 1, we can map the transition in § 3.3.

3.3 Mapping Loop Transitions

In this subsection, we map the loop transitions to the optimizer update rule introduced in § 3.2. We identify the recurrent state, projection, input map, decay operator and step size for Vanilla Loop, Parcae and HyperLoop. This analysis reveals input-map mismatches in Parcae and HyperLoop. The resulting mappings are summarized in Table 1.

Vanilla Loop. Let $\mathbf{Y}_l = \boldsymbol{y}_l \in \mathbb{R}^{d\times 1}$ and $\boldsymbol{b} \in \mathbb{R}^{d\times 1}$. The projection $\boldsymbol{o}_l = \boldsymbol{y}_l + \boldsymbol{b}$ uses $\mathbf{H}_l^{\text{in}} = \mathbf{I}_d$ and $\boldsymbol{b}_l = \boldsymbol{b}$. Setting $\eta_l = 1$ and $\mathbf{\Lambda}_l = \mathbf{I}_d$ in Eq. 8 gives $\boldsymbol{y}_{l+1} = -f'(\boldsymbol{o}_l) = \text{Blocks}_\theta(\boldsymbol{o}_l) = \text{Blocks}_\theta(\boldsymbol{y}_l + \boldsymbol{b})$, which recovers Eq. 4. Vanilla Loop is a special case of the optimizer update rule.

Parcae. For the structured state with $\boldsymbol{e} \in \mathbb{R}^{d\times 1}$, the projection $\boldsymbol{o}_l = \bar{\mathbf{A}}\boldsymbol{y}_l + \bar{\mathbf{B}}\boldsymbol{e}$ uses $\mathbf{H}_l^{\text{in}} = \bar{\mathbf{A}} \in \mathbb{R}^{d\times d}$ and $\boldsymbol{b}_l = \bar{\mathbf{B}}\boldsymbol{e} \in \mathbb{R}^{d\times 1}$. Setting $\eta_l = 1$ and $\mathbf{\Lambda}_l = \mathbf{I}_d$ in Eq. 8 yields $\boldsymbol{y}_{l+1} = -\bar{\mathbf{A}}^\top f'(\boldsymbol{o}_l) = \bar{\mathbf{A}}^\top \text{Blocks}_\theta(\boldsymbol{o}_l)$. Compared with Eq. 5, this update exposes the missing input-map factor.

HyperLoop. For the expanded state $\mathbf{Y}_l \in \mathbb{R}^{r\times d}$, the projection $\boldsymbol{o}_l = \mathbf{H}_l^{\text{pre}}\mathbf{Y}_l \in \mathbb{R}^{1\times d}$ uses $\mathbf{H}_l^{\text{in}} = \mathbf{H}_l^{\text{pre}} \in \mathbb{R}^{1\times r}$ without bias. Setting $\eta_l = 1$ and $\mathbf{\Lambda}_l = \mathbf{I}_r - \mathbf{H}_l^{\text{res}}$ in Eq. 8 gives $\mathbf{Y}_{l+1} = \mathbf{H}_l^{\text{res}}\mathbf{Y}_l + \mathbf{H}_l^{\text{pre}\top} \text{Blocks}_\theta(\mathbf{H}_l^{\text{pre}}\mathbf{Y}_l)$. Compared with Eq. 6, this update requires input-map alignment.

After mapping, Parcae and HyperLoop exhibit input-map mismatches under the optimization view. The gradient form in Eq. 10 uses the matrix product of the transposed input map and the output error: Parcae omits $\bar{\mathbf{A}}^\top$, while HyperLoop uses separate $\mathbf{H}_l^{\text{pre}}$ and $\mathbf{H}_l^{\text{post}}$. After input-map alignment, both Parcae and HyperLoop show improved performance as shown in § 5.1, providing further support for the proposed framework. These findings motivate the optimizer-guided loop design in § 4.

4 Optimizer-Guided Loop Design

To examine whether the framework can guide loop-transition design, we derive OperLoop using HyperLoop's expanded state and a delta objective. Let $\mathbf{Y}_l \in \mathbb{R}^{r\times d}$ be the expanded fast weight and $\mathbf{H}_l^{\text{in}} \in \mathbb{R}^{1\times r}$ the input map. The projection is $\boldsymbol{o}_l = \mathbf{H}_l^{\text{in}}\mathbf{Y}_l \in \mathbb{R}^{1\times d}$. We switch to the delta objective $\mathcal{L} = \frac{1}{2}\|\boldsymbol{o}_l - \boldsymbol{t}_l\|_2^2$, whose gradient with respect to the fast weight is $\nabla_{\mathbf{Y}_l}\mathcal{L}_l = (\mathbf{H}_l^{\text{in}})^\top(\boldsymbol{o}_l - \boldsymbol{t}_l) = (\mathbf{H}_l^{\text{in}})^\top(\boldsymbol{o}_l - \text{Blocks}_\theta(\boldsymbol{o}_l))$[3]. Substituting this gradient into Eq. 8 yields:

$$\mathbf{Y}_{l+1} = (\mathbf{I} - \eta_l \mathbf{\Lambda}_l)\,\mathbf{Y}_l + \eta_l\,(\mathbf{H}_l^{\text{in}})^\top\,(\text{Blocks}_\theta(\boldsymbol{o}_l) - \boldsymbol{o}_l)\,. \tag{12}$$

We parameterize the state-dependent input map $\mathbf{H}_l^{\text{in}}$, decay operator $\mathbf{\Lambda}_l$ and step size η_l as follows:

$$\eta_l = \sigma\left(a^{\text{lr}} \cdot \left(\mathbf{W}^{\text{lr}}\mathbf{Z}_l\right) + b^{\text{lr}}\right) \cdot \eta_{l-1} \qquad \in \mathbb{R}, \tag{13}$$

$$\mathbf{\Lambda}_l = \text{Diag}\left(\sigma\left(\boldsymbol{a}^{\text{wd}} \cdot \left(\mathbf{W}^{\text{wd}}\mathbf{Z}_l\right) + \boldsymbol{b}^{\text{wd}}\right)\right) \qquad \in \mathbb{R}^{r\times r}, \tag{14}$$

$$\mathbf{H}_l^{\text{in}} = \sigma\left(\boldsymbol{a}^{\text{in}} \cdot \left(\mathbf{W}^{\text{in}}\mathbf{Z}_l\right) + \boldsymbol{b}^{\text{in}}\right) \qquad \in \mathbb{R}^{1\times r}, \tag{15}$$

where the causal step size decreases monotonically with $\eta_l = \sigma \cdot \eta_{l-1}$ and is initialized with $\eta_{-1} = 1$. This design is inspired by learning-rate scheduling. Here, σ denotes the sigmoid function. We set $\mathbf{Z}_l = \text{RMSNorm}(\text{flatten}(\mathbf{Y}_l)) \in \mathbb{R}^{rd}$. The projection matrices satisfy $\mathbf{W}^{\text{wd}}, \mathbf{W}^{\text{in}} \in \mathbb{R}^{r\times rd}$, while $\mathbf{W}^{\text{lr}} \in \mathbb{R}^{1\times rd}$. The biases satisfy $\boldsymbol{b}^{\text{wd}}, \boldsymbol{b}^{\text{in}} \in \mathbb{R}^r$ and $\boldsymbol{b}^{\text{lr}} \in \mathbb{R}$. After R loops, we obtain $\mathbf{Y}^{(R)}$.

This architecture can also be expressed in a transition form similar to that of HyperLoop, but with a different parameterization. Substituting $\boldsymbol{o}_l = \mathbf{H}_l^{\text{in}}\mathbf{Y}_l$ into Eq. 12, we get the new transition:

$$\mathbf{Y}_{l+1} = \left(\mathbf{I} - \eta_l(\mathbf{\Lambda}_l + (\mathbf{H}_l^{\text{in}})^\top\mathbf{H}_l^{\text{in}})\right)\mathbf{Y}_l + \eta_l(\mathbf{H}_l^{\text{in}})^\top \text{Blocks}_\theta(\mathbf{H}_l^{\text{in}}\mathbf{Y}_l). \tag{16}$$

Specifically, setting $\mathbf{H}^{\text{pre}} = \mathbf{H}^{\text{in}}$, $\mathbf{H}^{\text{post}} = \eta_l\mathbf{H}^{\text{in}}$, and $\mathbf{H}^{\text{res}} = \mathbf{I} - \eta_l\left(\mathbf{\Lambda}_l + (\mathbf{H}_l^{\text{in}})^\top\mathbf{H}_l^{\text{in}}\right)$ recovers the proposed transition. This correspondence allows us to implement this loop transition within the HyperLoop implementation, as shown in Algorithm 1 in Appendix B.

[3]This delta objective is also inspired by Delta Linear Attention (Yang et al., 2024; 2025; Kimi Team et al., 2025; Huang et al., 2026b). The main distinction is that the looped block implicitly predicts the target representation, whereas Delta-based Linear Attention explicitly generates the value vector through linear projections.

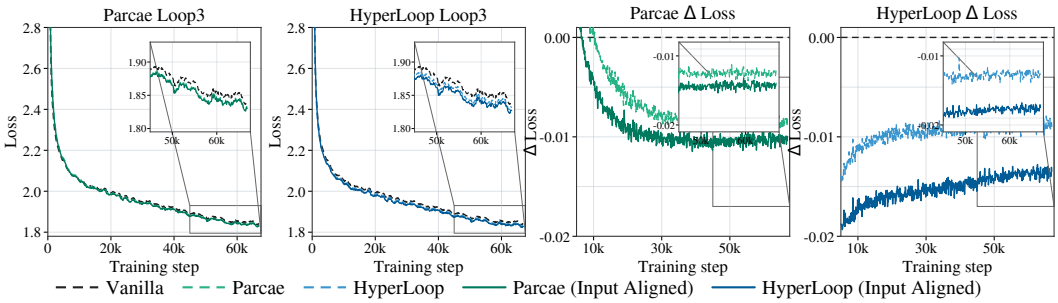


Figure 2: Training-loss comparison for optimizer-aligned loop transitions. The left two panels show training-loss trajectories for Parcae Loop3 and HyperLoop Loop3, together with their input-map-aligned variants. The right two panels show the corresponding loss differences relative to Vanilla Loop3, defined as $\Delta\mathcal{L} = \mathcal{L}_{\text{variant}} - \mathcal{L}_{\text{Vanilla}}$. Negative values indicate lower loss than Vanilla Loop3.

Table 2: Effect of input-map alignment on Parcae and HyperLoop in language modeling and commonsense evaluation. The looped variants differ only in their transitions. Parameters and training FLOPs are relative to the 18-layer baseline. The subscript n denotes normalized accuracy. ΔAvg. is the improvement over the corresponding unaligned model, in percentage points.

Model	*Param.*	*FLOPs*	**PPL ↓**		**Commonsense Acc. (%) ↑**							
			Lamb.	Wiki.	Lamb.	ARCc$_n$	ARCe$_n$	HellaS.$_n$	WinoG.	PIQA$_n$	Avg.	ΔAvg.
Baseline [18L]	1×	1×	6.56	13.08	60.16	40.96	71.13	64.78	59.51	76.61	62.19	–
Parcae [4L-4L×3-2L]	0.56×	1×	7.71	13.98	57.44	**38.57**	**65.91**	62.00	56.12	74.81	59.14	–
Parcae (Aligned)	0.56×	1×	**7.41**	**13.93**	**58.59**	37.71	65.53	**62.09**	**56.91**	**74.92**	**59.29**	**+0.15**
HyperLoop [4L-4L×3-2L]	0.56×	1×	7.68	13.96	57.46	**38.40**	63.51	61.41	57.77	**75.03**	58.93	–
HyperLoop (Aligned)	0.56×	1×	**7.35**	**13.93**	**58.63**	35.15	**66.67**	**62.83**	**58.56**	74.97	**59.47**	**+0.54**

5 Experiments

We use an MoE backbone with sliding-window attention (SWA) and full attention in a 3:1 ratio (Huang et al., 2026a). Our non-looped baselines are an 18-layer, 5B-parameter model with 600M activated parameters and a depth-scaled 32-layer, 10B-parameter model with 900M activated parameters. Following HyperLoop (Zeitoun et al., 2026), each looped variant uses the middle-loop pattern and matches the unrolled training FLOPs of its non-looped baseline, with approximately 50% as many parameters. All HyperLoop and OperLoop variants use four parallel streams ($r = 4$) throughout the experiments. After unrolling, the looped and non-looped baseline models have the same effective depth, KV-cache layout, and attention pattern. Comparisons among looped variants under these matched settings isolate the effect of the loop transition.

All models are trained with a sequence length of 4096. The 18-layer 5B and 32-layer 10B baselines are trained on approximately 300B and 500B tokens over 67K and 120K updates, respectively. Both scales use cosine learning-rate decay, a weight decay of 0.1, and global gradient clipping at 1.0. Model parameters are optimized with Muon using momentum 0.95 and six Polar Express NS iterations (Liu et al., 2025). The looped variants follow the same training recipe as their corresponding baselines. We evaluate perplexity and downstream accuracy using the open-source `lm-eval-harness` toolkit (Gao et al., 2024), largely following its default settings. Further model and training details are provided in Appendix C, and evaluation details are given in Appendix D.

5.1 Controlled Study of Optimizer-Aligned Loop Transitions

We first test whether the framework can guide improvements to existing loop transitions. Specifically, input-map alignment enforces the transpose relationship between the read and write maps required by the corresponding gradient update. For Parcae, we replace Eq. 5 with the aligned update $\boldsymbol{y}_{l+1} = \bar{\mathbf{A}}^{\top}\,\text{Blocks}_{\theta}(\bar{\mathbf{A}}\boldsymbol{y}_l + \bar{\mathbf{B}}\boldsymbol{e})$. For HyperLoop in Eq. 6, we set $\mathbf{H}_l^{\text{post}} = \mathbf{H}_l^{\text{pre}}$, giving $\mathbf{Y}_{l+1} = \mathbf{H}_l^{\text{res}}\mathbf{Y}_l + (\mathbf{H}_l^{\text{pre}})^{\top}(\text{Blocks}_{\theta}(\mathbf{H}_l^{\text{pre}}\mathbf{Y}_l) + \mathbf{b}_l)$. This constraint also reduces the number of parameters in the HyperLoop transition by removing the separate $\mathbf{H}^{\text{post}}$ map.

Table 3: Main results on commonsense, math, code, and reasoning benchmarks. Parameters and training FLOPs are relative to the corresponding non-looped baseline. **Bold** and underlined values indicate the best and second-best results among looped models at each scale, respectively. The subscript n denotes normalized accuracy, and $*$ denotes CoT evaluation. H.Eval and H.Eval$^+$ denote HumanEval and HumanEval+, respectively, both report pass@1 with greedy decoding.

		300B Training Tokens, 4k Sequence Length				*500B Training Tokens, 4k Sequence Length*			
Category	**Dataset**	**Base**$_{[18L]}$	**Loop**$_{[4L\text{-}4L\times3\text{-}2L]}$			**Base**$_{[32L]}$	**Loop**$_{[4L\text{-}8L\times3\text{-}4L]}$		
(*Metrics*)		**5B MoE**	**Parcae**	**HyperLoop**	**OperLoop**	**10B MoE**	**Parcae**	**HyperLoop**	**OperLoop**
	Parameters	1×	0.56×	0.56×	0.56×	1×	0.48×	0.48×	0.48×
	FLOPs	1×	1×	1×	1×	1×	1×	1×	1×
(*PPL* ↓)	**Wikitext**	13.08	13.98	13.96	**13.90**	11.06	11.76	11.87	**11.74**
	Lambada	6.56	7.71	7.68	**7.21**	4.82	5.20	5.32	**5.14**
Com.Sen.	**Lambada**	60.16	57.44	57.46	**59.29**	67.48	65.96	64.91	**66.04**
	ARCc$_n$	40.96	**38.57**	38.40	37.63	44.62	**45.73**	45.56	44.34
	ARCe$_n$	71.13	65.91	63.51	**65.99**	72.26	71.17	73.23	**73.65**
	HellaSwag$_n$	64.78	62.00	61.41	**62.20**	71.78	70.05	69.33	**70.07**
	WinoGrande	59.51	56.12	57.77	**57.85**	63.61	62.98	**64.17**	63.43
	PIQA$_n$	76.61	74.81	75.03	**75.63**	78.62	77.41	77.42	**77.88**
(*LL* ↑)	*LL Avg.*	*62.19*	*59.14*	*58.93*	***59.77***	*66.40*	*65.55*	*65.77*	***65.90***
Math	**GSM8K$_{5shot}$**	15.77	14.25	12.89	**15.85**	33.06	32.07	**33.36**	33.21
	GSM8K$_{8shot}$*	17.44	17.06	14.03	**17.13**	37.15	39.35	37.68	**39.65**
Code	**H.Eval$_{0shot}$**	21.95	21.34	18.90	**23.78**	28.66	28.05	**30.49**	29.88
	H.Eval$^+$$_{0shot}$	18.90	17.68	16.46	**20.12**	25.00	24.39	25.61	**26.22**
Reason.	**MMLU$_{5shot}$**	37.52	42.46	42.71	**44.41**	52.81	**53.92**	52.62	52.76
	BBH$_{3shot}$*	31.75	31.70	31.05	**32.18**	41.86	39.90	39.64	**43.04**
(*EM* ↑)	*EM Avg.*	*23.89*	*24.08*	*22.67*	***25.58***	*36.42*	*36.28*	*36.57*	***37.46***
Overall	*EM & LL Avg.*	*43.04*	*41.61*	*40.80*	***42.68***	*51.41*	*50.92*	*51.17*	***51.68***

Alignment improves training loss, perplexity, and average commonsense accuracy for both Parcae and HyperLoop (Figure 2; Table 2). For example, HyperLoop improves average accuracy from 58.93 to 59.47 while using fewer transition parameters. **These results support input-map alignment as a useful design constraint and provide empirical support for the optimization view**.

5.2 Main Downstream Results

We next test whether the framework can guide new transition designs by evaluating OperLoop. At both model scales, OperLoop improves average generative performance over the evaluated baselines.

Relative to the looped baselines in Table 3, OperLoop achieves the highest overall and EM averages among the evaluated looped models at both scales. OperLoop also achieves the lowest PPL and highest LL average among the looped variants. Under matched training FLOPs, these **results support the benefits of its transition design for both language modeling and generation**.

Relative to the non-looped baselines in Table 3, OperLoop uses approximately half the parameters while improving the EM average by 1.69 and 1.04 percentage points at the two scales. For PPL and LL accuracy, all looped models perform slightly worse at both scales. These differing rankings indicate that PPL and LL alone would understate looped models' generative performance in this comparison. Overall, **even though its PPL and LL results are slightly worse than those of the non-looped baseline, a looped model can achieve better results in the EM evaluation**.

The gains also vary across tasks and scales. Performance on both code benchmarks improves over that of the non-looped baselines at both scales. At the larger scale, OperLoop surpasses the baseline on GSM8K with CoT and BBH, reaching 43.04 on BBH compared with 41.86. The higher EM average thus reflects different task-level gains across scales. **Together, the alignment and OperLoop results support the framework for modifying existing transitions and designing new ones**.

Table 4: Downstream evaluation at Loop3 and Loop6. Models are trained separately at each loop count. Com. Avg. denotes average commonsense accuracy. EM Avg. is the arithmetic mean of the six reported math, code, and reasoning scores, and Δ is its difference from the baseline.

			PPL ↓		LL Acc ↑	EM Acc ↑							
Model	Params.	FLOPs	Wiki.	Lamb.	Com. Avg.	GSM8K	GSM8K*	H.Eval	H.Eval+	MMLU	BBH*	EM Avg.	Δ
Baseline (5B, 18L)	1×	1×	**13.08**	6.56	**62.19**	15.77	17.44	21.95	18.90	37.52	31.75	23.89	-
HyperLoop (Loop 3)	0.56×	1×	13.96	7.68	58.93	12.89	14.03	18.90	16.46	42.71	31.05	22.67	-1.22
OperLoop (Loop 3)	0.56×	1×	13.90	7.21	59.77	15.85	17.13	23.78	20.12	44.41	32.18	25.58	+1.69
HyperLoop (Loop 6)	0.56×	1.67×	13.35	6.51	61.39	22.44	27.14	24.39	20.73	48.25	35.54	29.75	+5.86
OperLoop (Loop 6)	0.56×	1.67×	13.16	**6.39**	61.72	**24.87**	**31.08**	**27.44**	**21.95**	**50.15**	**35.89**	**31.90**	**+8.01**

5.3 Ablation Study

We first evaluate scaling from three to six recurrent steps, then ablate the delta objective and step-size schedule at three loops. We report Wikitext and Lambada perplexity (PPL), average commonsense accuracy based on log-likelihood (LL), and the reported six-task exact match (EM) average.

Compute Scaling. We train Loop3 and Loop6 separately with three and six looped steps and evaluate each at its trained depth. Loop6 increases both training and inference compute to 1.67× the baseline FLOPs (Figure 3). Additional loops improve all six generation scores for both models, with the largest gains on math tasks (Table 4). For OperLoop, the EM average rises from 25.58 to 31.90, with 8-shot CoT GSM8K improving from 17.13 to 31.08. PPL and LL accuracy also improve, although Loop6 still trails the non-looped baseline on Wikitext PPL and LL accuracy while surpassing it on Lambada PPL. The matched-compute Loop3 comparison further shows why generative evaluation matters: OperLoop exceeds the non-looped baseline in EM average despite worse PPL and LL accuracy. Overall, additional recurrent computation improves generative performance at a fixed parameter budget, highlighting the value of evaluating looped models beyond PPL and LL.

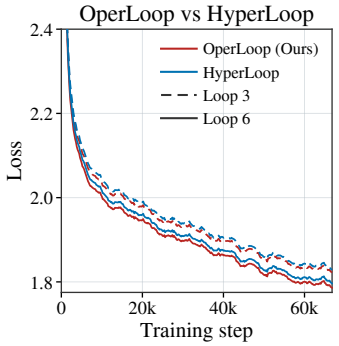


Figure 3: Scaling loop from three (Loop3) to six (Loop6).

Delta Objective Ablation. With the causal step-size schedule fixed, removing the delta objective worsens Wikitext PPL from 13.90 to 13.92 and Lambada PPL from 7.21 to 7.25, while lowering average commonsense accuracy from 59.77 to 59.36 and the EM average on generation tasks from 25.58 to 24.84 (Table 5). Thus, the delta objective improves both language modeling and generative performance. This indicates that loop transition designs derived from an objective under the optimization view can improve performance.

Table 5: Delta objective ablation at three loops with the causal step-size schedule fixed.

Variant	Delta Obj.	Step Size Schedule	Wiki. PPL ↓	Lamb. PPL ↓	LL Avg. ↑	EM Avg. ↑
OperLoop (Ours)	✓	Causal	**13.90**	**7.21**	**59.77**	**25.58**
w/o Delta Obj.	×	Causal	13.92	7.25	59.36	24.84

Step-Size Schedule Ablation. With the delta objective fixed, we compare the causal schedule $\eta_l = \sigma(\cdot)\eta_{l-1}$, the non-causal schedule $\eta_l = \sigma(\cdot)$, and a fixed unit step $\eta = 1$ (Table 6). The causal and non-causal schedules use the state-dependent sigmoid factor in Section 4. The causal schedule improves all reported metrics over unit steps, including EM by 1.18 percentage points. Without a monotonicity constraint, step-size fluctuations may induce oscillatory updates that hinder generation. Relative to the non-causal schedule, it improves the EM average by 1.56 percentage points. Since the step size is inspired by learning-rate scheduling, these performance gains support the optimization view.

Table 6: Step-size schedule ablation at three loops with the delta objective fixed.

Variant	Delta Obj.	Step Size Schedule	Wiki. PPL ↓	Lamb. PPL ↓	LL Avg. ↑	EM Avg. ↑
OperLoop (Ours)	✓	Causal	**13.90**	7.21	59.77	**25.58**
LR w Non-causal	✓	Non-Causal	13.93	**7.15**	**59.80**	24.02
w/o LR	✓	×	13.95	7.48	59.38	24.40

Table 7: Extended comparison of loop transitions under the optimization framework. Recurrent states are treated as fast weights corresponding to $\mathbf{W}$. Vector and expanded states are denoted by $\boldsymbol{y}_l \in \mathbb{R}^d$ and $\mathbf{Y}_l \in \mathbb{R}^{r\times d}$, respectively. Shared blocks predict the implicit target $\boldsymbol{t}_l = \mathrm{Blocks}_\theta(\boldsymbol{o}_l)$. Red highlights input-map mismatches under the chosen optimizer update rule. Here, $\boldsymbol{e}$ is the Prelude output, $\boldsymbol{\epsilon}$ is a noise vector, and $*$ denotes token shifts along the context. For training-free Recirculation, we compare only its transition. The Vanilla category includes Universal Transformers (Dehghani et al., 2019), MoEUT (Csordás et al., 2024), Recursive Transformer (Bae et al., 2025) and Ouro (Zhu et al., 2025).

		$\mathbf{W}_{i+1} = (1-\eta\lambda)\,\mathbf{W}_i - \eta\,(\boldsymbol{x}^\top f'(\boldsymbol{o}_l))$				
	Original Transition	$\boldsymbol{o}_l$	λ	η	$\boldsymbol{x}$	$\mathcal{L}$
Vanilla	$\boldsymbol{y}_{l+1} = \mathrm{Blocks}_\theta(\boldsymbol{o}_l)$	$\boldsymbol{y}_l$	1	1	$\mathbf{I}$	$-\langle \boldsymbol{t}_l, \boldsymbol{o}_l\rangle$
Huginn (Geiping et al., 2025)	$\boldsymbol{y}_{l+1} = \mathrm{Blocks}_\theta(\boldsymbol{o}_l)$	$\boldsymbol{y}_l + \mathbf{e}$	1	1	$\mathbf{I}$	$-\langle \boldsymbol{t}_l, \boldsymbol{o}_l\rangle$
Parcae (Prairie et al., 2026)	$\boldsymbol{y}_{l+1} = \mathrm{Blocks}_\theta(\boldsymbol{o}_l)$	$\bar{\mathbf{A}}\boldsymbol{y}_l + \bar{\mathbf{B}}\mathbf{e}$	1	1	$\mathbf{I}$	$-\langle \boldsymbol{t}_l, \boldsymbol{o}_l\rangle$
Full-Bandwidth* (Wang et al., 2026)	$\boldsymbol{y}_{l+1} = \mathrm{Blocks}_\theta(\boldsymbol{o}_l)$	$\sigma(\mathbf{W}^{\mathrm{g}}\boldsymbol{e}_l) \odot \mathbf{W}^{\mathrm{u}}\boldsymbol{y}_l$	1	1	$\mathbf{I}$	$-\langle \boldsymbol{t}_l, \boldsymbol{o}_l\rangle$
RecurrentGPT (Hegazy et al., 2026)	$\boldsymbol{y}_{l+1} = \boldsymbol{g}_l \odot \boldsymbol{y}_l + (1-\boldsymbol{g}_l) \odot \mathrm{Blocks}_\theta(\boldsymbol{o}_l)$	$\mathbf{W}\,[\boldsymbol{y}_l + \boldsymbol{\epsilon}, \boldsymbol{e}]$	1	$1-\boldsymbol{g}_l$	$\mathbf{I}$	$-\langle \boldsymbol{t}_l, \boldsymbol{o}_l\rangle$
Recirculation* (Mozer et al., 2026)	$\boldsymbol{y}_{l+1} = (1-\alpha)\,\boldsymbol{y}_l + \alpha\|\boldsymbol{y}_l\|_2 \frac{\mathrm{Blocks}_\theta(\boldsymbol{o}_l)}{\|\mathrm{Blocks}_\theta(\boldsymbol{o}_l)\|_2}$	$\boldsymbol{y}_l$	$\frac{1}{\|\boldsymbol{y}_l\|_2}$	$\alpha\|\boldsymbol{y}_l\|_2$	$\mathbf{I}$	$-\langle \frac{\boldsymbol{t}_l}{\|\boldsymbol{t}_l\|_2}, \boldsymbol{o}_l\rangle$
HyperLoop (Zeitoun et al., 2026)	$\mathbf{Y}_{l+1} = \mathbf{H}_l^{\mathrm{res}}\mathbf{Y}_l + (\mathbf{H}_l^{\mathrm{post}})^\top \mathrm{Blocks}_\theta(\boldsymbol{o}_l)$	$\mathbf{H}_l^{\mathrm{pre}}\,\mathbf{Y}_l$	$\mathbf{I} - \mathbf{H}_l^{\mathrm{res}}$	1	$\mathbf{H}_l^{\mathrm{post}}$	$-\langle \boldsymbol{t}_l, \boldsymbol{o}_l\rangle$
OperLoop (Ours)	$\mathbf{Y}_{l+1} = (\mathbf{I} - \eta_l\mathbf{\Lambda}_l)\mathbf{Y}_l + \eta_l(\mathbf{H}_l^{in})^\top(\mathrm{Blocks}_\theta(\boldsymbol{o}_l) - \boldsymbol{o}_l)$	$\mathbf{H}_l^{\mathrm{in}}\mathbf{Y}_l$	$\mathbf{\Lambda}_l$	η_l	$\mathbf{H}_l^{in}$	$\frac{1}{2}\|\boldsymbol{t}_l - \boldsymbol{o}_l\|_2^2$

6 Analysis and Discussion

Table 7 extends the analysis in § 3.3 to additional loop models. We compare their *projections* ($\boldsymbol{o}_l$), local *objectives* ($\mathcal{L}$), and *optimizer update rules* (weight decay λ, step size η, and input map $\boldsymbol{x}$) to identify directions for future transition design.

Input injection, gating, and expanded-state readouts define different *projections*. Decay and step size control state retention and update scales in the *optimizer update rule*. Under the chosen local gradient formulation, input-map mismatches occur in Parcae, Full-Bandwidth, RecurrentGPT, and HyperLoop. The framework identifies which transitions need input-map alignment to match the chosen optimizer update rule.

Most compared transitions map to a negative inner product *objective*, with Recirculation using a normalized target. Their write term is driven by the predicted target. OperLoop's delta objective instead uses the difference between the target and projected state, adding an explicit correction for the current projection. The ablation in § 5 supports this objective choice, illustrating how the framework guides the design of a different update signal.

This comparison suggests a roadmap: specify the *projection* and local *objective*, select an *optimizer update rule*, and derive the loop transition. Future work can vary the structure, computation, and initialization of fast weights and explore linear or nonlinear *projections*. *Objectives* shape update signals, and their values may guide adaptive looping for early stopping or extrapolation. Ideas from advanced *optimizers* like Muon (Jordan et al., 2024) or Hyperball (Wen et al., 2026) may inspire new loop architectures to improve loop stability and reasoning performance.

7 Conclusion

In this work, we present an optimization view of Looped Transformers that treats recurrent states as fast weights updated under local prediction objectives. This framework organizes loop transitions into a projection, an objective, and an optimizer rule, with shared blocks predicting implicit targets. It identifies input-map mismatches in existing transitions and guides the design of OperLoop. The alignment studies, downstream evaluations, and component ablations support its design guidance. The extended comparison further identifies directions for improving loop transitions in the future.

AI USE STATEMENT

In this work, we used generative AI tools for language polishing, grammar correction, and figure creation to improve the clarity and readability of the manuscript. We also used these tools to help identify related literature. We did not use generative AI tools to generate original research ideas. We have reviewed all AI-assisted work, including the edited text, figures, and suggested references, to ensure accuracy and consistency with our research. We take responsibility for the final content of this work, including text, claims, or artifacts produced with the aid of generative AI.

REFERENCES

Ekin Akyürek, Dale Schuurmans, Jacob Andreas, Tengyu Ma, and Denny Zhou. What learning algorithm is in-context learning? investigations with linear models. In *International Conference on Learning Representations*, 2023.

Marcin Andrychowicz, Misha Denil, Sergio Gómez, Matthew W. Hoffman, David Pfau, Tom Schaul, Brendan Shillingford, and Nando de Freitas. Learning to learn by gradient descent by gradient descent. In *Advances in Neural Information Processing Systems*, 2016.

Jimmy Lei Ba, Geoffrey E. Hinton, Volodymyr Mnih, Joel Z. Leibo, and Catalin Ionescu. Using fast weights to attend to the recent past. In *Advances in Neural Information Processing Systems*, 2016.

Sangmin Bae, Adam Fisch, Hrayr Harutyunyan, Ziwei Ji, Seungyeon Kim, and Tal Schuster. Relaxed recursive transformers: Effective parameter sharing with layer-wise lora. In *International Conference on Learning Representations*, 2025. URL https://arxiv.org/abs/2410.20672.

Yonatan Bisk, Rowan Zellers, Ronan Le Bras, Jianfeng Gao, and Yejin Choi. PIQA: Reasoning about physical commonsense in natural language. *arXiv preprint arXiv:1911.11641*, 2019. URL https://arxiv.org/abs/1911.11641.

Lizhang Chen, Jonathan Li, Chen Liang, Ni Lao, and Qiang Liu. Training-free looped transformers. *arXiv preprint arXiv:2605.23872*, 2026.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Peter Clark, Isaac Cowhey, Oren Etzioni, Tushar Khot, Ashish Sabharwal, Carissa Schoenick, and Oyvind Tafjord. Think you have solved question answering? try arc, the ai2 reasoning challenge. *arXiv preprint arXiv:1803.05457*, 2018.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

Róbert Csordás, Kazuki Irie, Jürgen Schmidhuber, Christopher Potts, and Christopher D. Manning. Moeut: Mixture-of-experts universal transformers. *arXiv preprint arXiv:2405.16039*, 2024. URL https://arxiv.org/abs/2405.16039.

Tri Dao and Albert Gu. Transformers are ssms: Generalized models and efficient algorithms through structured state space duality. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235, pp. 10041–10071. PMLR, 2024. URL https://proceedings.mlr.press/v235/dao24a.html.

Mostafa Dehghani, Stephan Gouws, Oriol Vinyals, Jakob Uszkoreit, and Lukasz Kaiser. Universal transformers. In *International Conference on Learning Representations*, 2019.

Ying Fan, Yilun Du, Kannan Ramchandran, and Kangwook Lee. Looped transformers for length generalization. In *International Conference on Learning Representations*, 2025.

Jacob Fein-Ashley and Paria Rashidinejad. Solve the loop: Attractor models for language and reasoning. *arXiv preprint arXiv:2605.12466*, 2026.

Leo Gao, Jonathan Tow, Baber Abbasi, Stella Biderman, Sid Black, Anthony DiPofi, Charles Foster, Laurence Golding, Jeffrey Hsu, Alain Le Noac'h, Haonan Li, Kyle McDonell, Niklas Muennighoff, Chris Ociepa, Jason Phang, Laria Reynolds, Hailey Schoelkopf, Aviya Skowron, Lintang Sutawika, Eric Tang, Anish Thite, Ben Wang, Kevin Wang, and Andy Zou. The language model evaluation harness, 07 2024. URL https://zenodo.org/records/12608602.

Zitian Gao, Yilong Chen, Yihao Xiao, Xinyu Yang, Ran Tao, Joey Zhou, and Bryan Dai. Loop the loopies! *arXiv preprint arXiv:2607.16051*, 2026.

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

Alex Graves. Adaptive computation time for recurrent neural networks. *arXiv preprint arXiv:1603.08983*, 2016.

Karol Gregor and Yann LeCun. Learning fast approximations of sparse coding. In *Proceedings of the 27th International Conference on Machine Learning*, 2010. URL https://icml.cc/Conferences/2010/papers/449.pdf.

Albert Gu, Karan Goel, and Christopher Ré. Efficiently modeling long sequences with structured state spaces. *arXiv preprint arXiv:2111.00396*, 2021.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016.

Amr Hegazy, Amr Alanwar, and Mostafa Elhoushi. Recurrentgpt: Expressive depth through recurrent modulation in transformers. *arXiv preprint arXiv:2608.15062v1*, 2026. URL https://arxiv.org/abs/2608.15062v1.

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding. In *International Conference on Learning Representations*, 2021.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8): 1735–1780, 1997.

Ailin Huang, Ang Li, Aobo Kong, et al. Step 3.5 flash: Open frontier-level intelligence with 11b active parameters, 2026a. URL https://arxiv.org/abs/2602.10604.

Benhao Huang. Exact input writes improve stable looped language models. Husky's Log (blog post), April 2026. URL https://huskydoge.github.io/husky-blog/posts/recursive_models/improve-parcae/.

Yulong Huang, Xiang Liu, Hongxiang Huang, Xiaopeng Lin, Zunchang Liu, Xiaowen Chu, Zeke Xie, and Bojun Cheng. Mdn: Parallelizing stepwise momentum for delta linear attention. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2026b. URL https://icml.cc/virtual/2026/poster/63901.

Keller Jordan, Yuchen Jin, Vlado Boza, Jiacheng You, Franz Cesista, Laker Newhouse, and Jeremy Bernstein. Muon: An optimizer for hidden layers in neural networks, 2024. URL https://kellerjordan.github.io/posts/muon/.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

Kimi Team, Yu Zhang, Zongyu Lin, Xingcheng Yao, Jiaxi Hu, Fanqing Meng, Chengyin Liu, Xin Men, Songlin Yang, Zhiyuan Li, Wentao Li, Enzhe Lu, Weizhou Liu, Yanru Chen, Weixin Xu, Longhui Yu, Yejie Wang, Yu Fan, Longguang Zhong, Enming Yuan, Dehao Zhang, Yizhi Zhang, T. Y. Liu, Haiming Wang, Shengjun Fang, Weiran He, Shaowei Liu, Yiwei Li, Jianlin Su, Jiezhong Qiu, Bo Pang, Junjie Yan, Zhejun Jiang, Weixiao Huang, Bohong Yin, Jiacheng You, Chu Wei, Zhengtao Wang, Chao Hong, Yutian Chen, Guanduo Chen, Yucheng Wang, Huabin Zheng, Feng Wang, Yibo Liu, Mengnan Dong, Zheng Zhang, Siyuan Pan, Wenhao Wu, Yuhao Wu, Longyu Guan, Jiawen Tao, Guohong Fu, Xinran Xu, Yuzhi Wang, Guokun Lai, Yuxin Wu, Xinyu Zhou, Zhilin Yang, and Yulun Du. Kimi linear: An expressive, efficient attention architecture, 2025. URL https://arxiv.org/abs/2510.26692.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners. *arXiv preprint arXiv:2205.11916*, 2022.

Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. Albert: A lite bert for self-supervised learning of language representations. In *International Conference on Learning Representations*, 2020.

Huan Li, Yibo Yang, Dongmin Chen, and Zhouchen Lin. Optimization algorithm inspired deep neural network structure design. In Jun Zhu and Ichiro Takeuchi (eds.), *Proceedings of the 10th Asian Conference on Machine Learning*, volume 95 of *Proceedings of Machine Learning Research*, pp. 614–629. PMLR, 2018. URL https://proceedings.mlr.press/v95/li18f.html.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by ChatGPT really correct? rigorous evaluation of large language models for code generation. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=1qvx610Cu7.

Jingyuan Liu, Jianlin Su, Xingcheng Yao, Zhejun Jiang, Guokun Lai, Yulun Du, Yidao Qin, Weixin Xu, Enzhe Lu, Junjie Yan, et al. Muon is scalable for llm training. *arXiv preprint arXiv:2502.16982*, 2025. URL https://arxiv.org/abs/2502.16982.

Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843*, 2016. URL https://arxiv.org/abs/1609.07843.

Michael C Mozer, Shoaib Ahmed Siddiqui, Danny Sawyer, Sunny Sanyal, and Rosanne Liu. Recirculation. *arXiv preprint arXiv:2608.17981*, 2026.

Niklas Muennighoff, Alexander Rush, Boaz Barak, Teven Le Scao, Nouamane Tazi, Aleksandra Piktus, Sampo Pyysalo, Thomas Wolf, and Colin A Raffel. Scaling data-constrained language models. *Advances in Neural Information Processing Systems*, 36:50358–50376, 2023.

Hayden Prairie, Zachary Novack, Taylor Berg-Kirkpatrick, and Daniel Y. Fu. Parcae: Scaling laws for stable looped language models. *arXiv preprint arXiv:2604.12946*, 2026.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. Technical report, OpenAI, 2019. URL https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf.

Ivan Rodkin, Daniil Orel, Konstantin Smirnov, Arman Bolatov, Bilal Elbouardi, Besher Hassan, Yurii Kuratov, Aydar Bulatov, Preslav Nakov, Timothy Baldwin, Artem Shelmanov, and Mikhail Burtsev. Beyond memorization: Extending reasoning depth with recurrence, memory and test-time compute scaling. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 42385–42404, 2026. doi: 10.18653/v1/2026.findings-acl.2103. URL https://aclanthology.org/2026.findings-acl.2103/.

Keisuke Sakaguchi, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. WinoGrande: An adversarial winograd schema challenge at scale. *arXiv preprint arXiv:1907.10641*, 2019. URL https://arxiv.org/abs/1907.10641.

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

Imanol Schlag, Kazuki Irie, and Jürgen Schmidhuber. Linear transformers are secretly fast weight programmers. In *International Conference on Machine Learning*, pp. 9355–9366, 2021. URL https://proceedings.mlr.press/v139/schlag21a.html.

Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling LLM test-time compute optimally can be more effective than scaling model parameters. *arXiv preprint arXiv:2408.03314*, 2024.

Yu Sun, Xinhao Li, Karan Dalal, Jiarui Xu, Arjun Vikram, Genghan Zhang, Yann Dubois, Xinlei Chen, Xiaolong Wang, Sanmi Koyejo, Tatsunori Hashimoto, and Carlos Guestrin. Learning to (learn at test time): RNNs with expressive hidden states. *arXiv preprint arXiv:2407.04620*, 2024. URL https://arxiv.org/abs/2407.04620.

Richard S. Sutton. The bitter lesson. http://www.incompleteideas.net/IncIdeas/BitterLesson.html, 2019.

Mirac Suzgun, Nathan Scales, Nathanael Schärli, Sebastian Gehrmann, Yi Tay, Hyung Won Chung, Aakanksha Chowdhery, Quoc V. Le, Ed H. Chi, Denny Zhou, and Jason Wei. Challenging big-bench tasks and whether chain-of-thought can solve them. *arXiv preprint arXiv:2210.09261*, 2022.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, 2017.

Johannes von Oswald, Eyvind Niklasson, Ettore Randazzo, Joao Sacramento, Alexander Mordvintsev, Andrey Zhmoginov, and Max Vladymyrov. Transformers learn in-context by gradient descent. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pp. 35151–35174. PMLR, 2023. URL https://proceedings.mlr.press/v202/von-oswald23a.html.

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

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*, 2022.

Kaiyue Wen, Xingyu Dang, Kaifeng Lyu, Tengyu Ma, and Percy Liang. Fantastic pretraining optimizers and where to find them ii: Hyperball optimization. *arXiv preprint arXiv:2606.16899*, 2026.

Zhenda Xie, Yixuan Wei, Huanqi Cao, Chenggang Zhao, Chengqi Deng, Jiashi Li, Damai Dai, Huazuo Gao, Jiang Chang, Kuai Yu, et al. mHC: Manifold-constrained hyper-connections. *arXiv preprint arXiv:2512.24880*, 2025.

Songlin Yang, Bailin Wang, Yu Zhang, Yikang Shen, and Yoon Kim. Parallelizing linear transformers with the delta rule over sequence length. *Advances in neural information processing systems*, 37:115491–115522, 2024.

Songlin Yang, Jan Kautz, and Ali Hatamizadeh. Gated delta networks: Improving Mamba2 with delta rule. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=r8H7xhYPwz.

Abbas Zeitoun, Lucas Torroba-Hennigen, and Yoon Kim. HyperLoop transformers. *arXiv preprint arXiv:2604.21254*, 2026.

Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. Hellaswag: Can a machine really finish your sentence? In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, 2019.

Rui-Jie Zhu, Zixuan Wang, Kai Hua, Tianyu Zhang, Ziniu Li, Haoran Que, Boyi Wei, Zixin Wen, Fan Yin, He Xing, Lu Li, Jiajun Shi, Kaijing Ma, Shanda Li, Taylor Kergan, Andrew Smith, Xingwei Qu, Mude Hui, Bohong Wu, Qiyang Min, Hongzhi Huang, Xun Zhou, Wei Ye, Jiaheng Liu, Jian Yang, Yunfeng Shi, Chenghua Lin, Enduo Zhao, Tianle Cai, Ge Zhang, Wenhao Huang, Yoshua Bengio, and Jason Eshraghian. Scaling latent reasoning via looped language models. *arXiv preprint arXiv:2510.25741*, 2025.

A Extended Related Work

Origins and development of Looped Transformers. Looped Transformers build on parameter sharing across depth and repeated application of a shared transition. Adaptive Computation Time makes the number of recurrent updates input-dependent (Graves, 2016). Universal Transformers repeatedly apply a shared Transformer block with adaptive per-position computation (Dehghani et al., 2019). ALBERT improves parameter efficiency through cross-layer sharing (Lan et al., 2020). Later work studies length generalization with adaptive loop counts (Fan et al., 2025). RecurrentGPT adds recurrent modulation to a shared Transformer core (Hegazy et al., 2026). Training-free methods introduce inference-time looping into frozen pretrained models without additional training (Chen et al., 2026). Looping thus increases effective depth without adding new parameters at each step.

Looping, Test-Time Compute, and Reasoning. Looping can scale test-time computation through additional recurrent passes, as demonstrated by Huginn (Geiping et al., 2025). Broader work on test-time scaling studies verifier-guided search and iterative response revision (Snell et al., 2024). Ouro studies latent reasoning through repeated hidden-state computation (Zhu et al., 2025). Attractor Models refine embeddings by solving for a fixed point (Fein-Ashley & Rashidinejad, 2026). Saunshi et al. analyze the reasoning capacity of looped Transformers and establish theoretical connections to chain-of-thought (CoT) (Saunshi et al., 2025). Rodkin et al. study recurrence, memory, and test-time computation in controlled cellular-automata tasks (Rodkin et al., 2026). CoT prompting elicits explicit intermediate reasoning steps (Wei et al., 2022; Kojima et al., 2022), whereas looping refines hidden states. The two approaches therefore provide complementary routes to test-time reasoning.

Recurrent Updates Along Depth and Context. Looped Transformers apply recurrence along depth, updating the state at each visit to a shared block for a fixed input. Residual connections propagate representations across layers in ResNets and Transformers (He et al., 2016; Vaswani et al., 2017). Looping additionally shares block parameters across recurrent visits. Context-axis recurrence instead updates the state as tokens are processed. Examples include recurrent networks (Hochreiter & Schmidhuber, 1997), Fast Weight and Linear-Attention models (Ba et al., 2016; Schlag et al., 2021), and Structured State Space models (Gu et al., 2021; Dao & Gu, 2024). Delta-based linear attention models update recurrent memory along context using delta-rule corrections (Yang et al., 2025; Huang et al., 2026b; Kimi Team et al., 2025). These updates offer useful analogies for loop transitions, but follow a different recurrence axis.

Optimization Views Beyond Looped Transformers. Viewing loop transitions as optimization updates connects our framework to work on neural computation beyond looped Transformers. LISTA unfolds iterative shrinkage and thresholding for sparse coding into a trainable network (Gregor & LeCun, 2010). Li et al. (2018) interprets feedforward computation with shared linear transformations as gradient descent. They use accelerated optimization algorithms to motivate network architectures. Learned optimizers use recurrent networks to learn parameter-update rules (Andrychowicz et al., 2016). Studies of linear regression show that Transformers can implement gradient-based in-context learning in their forward pass (Akyürek et al., 2023; von Oswald et al., 2023). Test-Time Training (TTT) layers represent the recurrent hidden state as model parameters. As tokens are processed, gradient descent updates these parameters using a self-supervised objective (Sun et al., 2024). Our optimization framework derives depth-axis loop transitions from a projection, a local objective, and an optimizer update rule. This local objective defines the forward transition and differs from the language-modeling loss used for end-to-end training.

B Pseudocode for OperLoop

Algorithm 1 gives a common implementation of HyperLoop and OperLoop. Blue and orange denote the HyperLoop and OperLoop transition rules, respectively. Both initialize the recurrent state by duplicating the Prelude output into parallel streams. At each loop, OperLoop computes a state-dependent input map, decay operator, and causal step size. These determine the read, write, and residual maps. The residual map incorporates the delta correction in Eq. 16. After R recurrent steps, the streams are averaged and passed through the Coda and language-model head.

Algorithm 1 Common implementation of HyperLoop (Zeitoun et al., 2026) and OperLoop (**ours**).

Require: Input tokens $\mathbf{x}$; recurrent-step index l; number of streams r; hidden dimension d; number of loops R; Trainable bias $\boldsymbol{e}_l \in \{\boldsymbol{e}_l\}_{l=0,\cdots,R-1}$; Trainable weights $\{\mathbf{W}_l^\tau, \boldsymbol{b}_l^\tau, \boldsymbol{a}_l^\tau\}_{l=0,\cdots,R-1}$, $\tau \in \{\text{res, pre, post}\}$ or $\tau \in \{\text{wd, in, lr}\}$.

Ensure: Language-model head output $\mathrm{LMHead}_\theta(\mathbf{h}_{\text{out}})$.

$\mathbf{h} \leftarrow \mathrm{Embedding}_\theta(\mathbf{x})$

$\mathbf{h} \leftarrow \mathrm{Prelude}_\theta(\mathbf{h})$

$\mathbf{Y}_0 \leftarrow \mathrm{Duplicate}_r(\mathbf{h})$ ▷ $\in \mathbb{R}^d \to \mathbb{R}^{r\times d}$

$\eta_{-1} \leftarrow \mathbf{1}$

for $l \leftarrow 0$ **to** $R-1$ **do**

  $\mathbf{Z}_l \leftarrow \mathrm{RMSNorm}(\mathrm{flatten}(\mathbf{Y}_l))$ ▷ $\in \mathbb{R}^{r\times d} \to \mathbb{R}^{rd}$

  $\eta_l \leftarrow \eta_{l-1} \cdot \sigma(a^{\text{lr}} \cdot (\mathbf{W}^{\text{lr}}\mathbf{Z}_l) + b^{\text{lr}})$ ▷ $\in \mathbb{R}$

  $\mathbf{H}_l^{\text{in}} \leftarrow \sigma(\boldsymbol{a}^{\text{in}} \cdot (\mathbf{W}^{\text{in}}\mathbf{Z}_l) + \boldsymbol{b}^{\text{in}})$ ▷ $\in \mathbb{R}^{1\times r}$

  $\mathbf{\Lambda}_l \leftarrow \mathrm{Diag}\left(\sigma\left(\boldsymbol{a}^{\text{wd}} \cdot (\mathbf{W}^{\text{wd}}\mathbf{Z}_l) + \boldsymbol{b}^{\text{wd}}\right)\right)$ ▷ $\in \mathbb{R}^{r\times r}$

  $\mathbf{H}_l^{\text{pre}} \leftarrow \sigma\left(\boldsymbol{a}_l^{\text{pre}} \cdot (\mathbf{W}_l^{\text{pre}}\mathbf{Z}_l) + \boldsymbol{b}_l^{\text{pre}}\right)$ or $\mathbf{H}_l^{\text{in}}$ ▷ $\in \mathbb{R}^{1\times r}$

  $\mathbf{H}_l^{\text{post}} \leftarrow 2 \cdot \sigma\left(\boldsymbol{a}_l^{\text{post}} \cdot (\mathbf{W}_l^{\text{post}}\mathbf{Z}_l) + \boldsymbol{b}_l^{\text{post}}\right)$ or $\eta_l \mathbf{H}_l^{\text{in}}$ ▷ $\in \mathbb{R}^{1\times r}$

  $\mathbf{H}_l^{\text{res}} \leftarrow \mathrm{Diag}\left(\sigma\left(\boldsymbol{a}_l^{\text{res}} \cdot (\mathbf{W}_l^{\text{res}}\mathbf{Z}_l) + \boldsymbol{b}_l^{\text{res}}\right)\right)$ or $\mathbf{I} - \eta_l \cdot \left(\mathbf{\Lambda}_l + \left(\mathbf{H}_l^{\text{in}}\right)^\top \mathbf{H}_l^{\text{in}}\right)$ ▷ $\in \mathbb{R}^{r\times r}$

  $\mathbf{Y}_{l+1} \leftarrow \mathbf{H}_l^{\text{res}}\mathbf{Y}_l + (\mathbf{H}_l^{\text{post}})^\top \cdot (\mathrm{RecurrentBlock}_\theta(\mathbf{H}_l^{\text{pre}}\mathbf{Y}_l) + \boldsymbol{e}_l)$ ▷ $\in \mathbb{R}^{r\times d}$

end for

$\mathbf{h}_{\text{loop}} \leftarrow \mathbf{Y}_R.\mathrm{Mean}(-2)$ ▷ $\in \mathbb{R}^{r\times d} \to \mathbb{R}^d$

$\mathbf{h}_{\text{out}} \leftarrow \mathrm{Coda}_\theta(\mathbf{h}_{\text{loop}})$

return $\mathrm{LMHead}_\theta(\mathbf{h}_{\text{out}})$

State layout and module correspondence. All HyperLoop and OperLoop experiments use $r = 4$ parallel streams. The state $\mathbf{Y}_l \in \mathbb{R}^{r\times d}$ represents one token, with batch and token indices omitted. Flattening and RMSNorm produce the controller input $\mathbf{Z}_l \in \mathbb{R}^{rd}$. The map $\mathbf{H}_l^{\text{in}} \in \mathbb{R}^{1\times r}$ reads the streams into the block input. The decay operator $\mathbf{\Lambda}_l \in \mathbb{R}^{r\times r}$ acts on the stream dimension. Each $\mathrm{RecurrentBlock}_\theta$ call represents the shared stack Blocks_θ of K Transformer blocks.

Implementation details. HyperLoop and OperLoop use loop-specific transition parameters. Their loop indices are omitted in the OperLoop expressions. Both retain HyperLoop's trainable output bias $\boldsymbol{e}_l$. In the implementation, it shifts the predicted target to $\boldsymbol{t}_l = \mathrm{Blocks}_\theta(\boldsymbol{o}_l) + \boldsymbol{e}_l$. The optimizer-derived formulas omit this term for brevity. This matches HyperLoop's bias setting when assessing performance gains from the optimizer-guided loop transition. Parcae uses shared transition parameters and initializes its recurrent state with Gaussian noise. All recurrent steps are fully backpropagated. Further details of the local optimization interpretation appear in Appendix E.

Causal step-size schedule. Let γ_l denote the sigmoid factor in the step-size parameterization in § 4. With $\eta_{-1} = 1$, the causal recurrence gives $\eta_l = \gamma_l\eta_{l-1} = \prod_{j=0}^{l}\gamma_j$. Since $0 < \gamma_l < 1$ for finite sigmoid inputs, the step size decreases monotonically across recurrent depth. Here, "causal" refers to dependence on the previous loop's step size. The non-causal ablation uses $\eta_l = \gamma_l$, and the unit-step ablation fixes $\eta = 1$. Both retain the delta objective in Table 6.

C Model and Training Configuration

Table 8 summarizes the architecture of the 18- and 32-layer baselines. Both use the same hidden width, attention configuration, and MoE design. Sliding-window and full attention follow a repeating 3:1 pattern. The first two layers are dense, and the remaining layers use MoE. Parameter estimates refer to trainable non-vocabulary parameters.

Loop layouts and compute scaling. Table 9 details the middle-loop configurations in Tables 3 and 4. Each layout contains i Prelude blocks, K shared blocks, and j Coda blocks. Repeating the shared blocks for R steps gives an effective depth of $i+RK+j$. The number of distinct Transformer blocks is $i + K + j$. Parameter and FLOP totals also depend on the block contents and transition overhead. Loop3 and Loop6 are trained separately and evaluated at their respective training depths. Their comparison therefore changes both training and inference computation.

Table 8: Architecture configuration used in the experiments. The sliding–sliding–sliding–full pattern repeats every four layers.

Component	Setting	Value
Model family	Architecture	Decoder-only Transformer
Numerics	Parameter / router / LM-head precision	BF16 / FP32 / FP32
Representation	Hidden size / vocabulary size	1024 / 128,896
Attention	Heads / head dimension / GQA groups	16 / 128 / 4
Context	Maximum sequence length	4096
Attention pattern	Repeated every four layers	Sliding–Sliding–Sliding–Full
Sliding attention	Window / QK RoPE dimension / RoPE theta	512 / 128 / 1,000
Full attention	QK RoPE dimension / RoPE theta	64 / 10,000
Feed-forward	Dense intermediate size / activation	4096 / SwiGLU
MoE	Type / experts per MoE layer / router top-k	Marco-style / 256 / 8
MoE experts	Routed / shared intermediate size	384 / 384
Routing	Router / load balancing	Sigmoid / auxiliary-loss-free
Routing	Routed scaling / router bias update rate / coefficient	$2.5 / 5 \times 10^{-3} / 10^{-3}$
Initialization	Standard deviation / LayerNorm epsilon	$0.006 / 10^{-5}$
Initialization	RMSNorm gamma initialization	Zero
18-layer model	Layers / dense layers / MoE layers	18 / 0–1 / 2–17
18-layer model	Full-attention layers / estimated parameters	3, 7, 11, 15 / 4.974B
32-layer model	Layers / dense layers / MoE layers	32 / 0–1 / 2–31
32-layer model	Full-attention layers / estimated parameters	3, 7, 11, 15, 19, 23, 27, 31 / 9.296B

Table 9: The number of distinct blocks is $i + K + j$, and the effective depth is $i + RK + j$.

Setting	Prelude i	Shared K	Loops R	Coda j	Effective Depth	Distinct Blocks
18-layer comparison (Loop3)	4	4	3	2	18	10
32-layer comparison (Loop3)	4	8	3	4	32	16
Compute scaling (Loop6)	4	4	6	2	30	10

Training settings. Table 10 reports the training settings. Both scales use the same batch configuration, weight decay, gradient clipping, and cosine learning-rate schedule. Muon updates eligible QKV and FFN linear weights, while AdamW updates the remaining parameters. The peak learning rate, warmup duration, and number of updates vary by scale.

Training budgets. Nominal token counts are calculated from the sequence length, global batch size, and number of updates. A global batch of 1024 sequences of length 4096 gives 4,194,304 nominal token positions per update. The 67,000 and 120,000 updates correspond to 281.018B and 503.316B nominal tokens, respectively. The main text uses the approximate budget labels 300B and 500B. Comparisons at matched compute use training FLOPs, including transition overhead. Parameter ratios are normalized and rounded; they do not imply exact parameter equality.

D Evaluation Details

We evaluate perplexity and downstream performance with `lm-eval-harness` v0.4.13 (Gao et al., 2024). Each benchmark uses its task-specific default prompts, few-shot settings, answer filters, normalization rules, and generation parameters. Table 3 summarizes the few-shot and chain-of-thought (CoT) settings. All compared models use the same evaluation configuration.

We report perplexity on Wikitext (Merity et al., 2016) and Lambada (Radford et al., 2019). Commonsense scores cover Lambada) (Radford et al., 2019), ARCc and ARCe (Clark et al., 2018), HellaSwag (Zellers et al., 2019), WinoGrande (Sakaguchi et al., 2019), and PIQA (Bisk et al., 2019).

For GSM8K (Cobbe et al., 2021), we report 5-shot evaluation and 8-shot evaluation with CoT. HumanEval (Chen et al., 2021) and HumanEval+ (Liu et al., 2023) use 0-shot evaluation and report

Table 10: Training configuration. Nominal token counts are computed from the sequence length, global batch size, and number of updates.

Category	Setting	Value
Objective	Training objective / loss	Causal LM / vocabulary-parallel cross-entropy
Batching	Global batch / micro batch	1024 / 8 sequences
Batching	Nominal tokens per update	4,194,304
Precision	Training parameter precision	BF16
Optimizer	Muon momentum / Nesterov	0.95 / enabled
Optimizer	Newton–Schulz variant / iterations	Polar Express / 6
Optimizer	AdamW β_1 / β_2 / ϵ	0.9 / 0.95 / 10^{-8}
Regularization	Weight decay / global gradient clipping	0.1 / 1.0
Schedule	Learning-rate schedule / minimum rate	Cosine / 10^{-5}
Schedule	Schedule unit	Iteration
18-layer model	Peak rate / warmup / iterations	7.3×10^{-4} / 1,000 / 67,000
18-layer model	Nominal training tokens / Muon-matched AdamW RMS	281.018B / 0.18
32-layer model	Peak rate / warmup / iterations	5.6×10^{-4} / 2,000 / 120,000
32-layer model	Nominal training tokens / Muon-matched AdamW RMS	503.316B / 0.18

pass@1. MMLU (Hendrycks et al., 2021) uses 5-shot evaluation, with generated answers scored by exact match (EM). BBH (Suzgun et al., 2022) uses 3-shot evaluation with CoT.

Downstream scores are reported as percentages. LL Avg. is the unweighted mean of the six reported commonsense scores. EM Avg. is the unweighted mean of the six reported math, code, and reasoning scores, including code pass@1. The two GSM8K settings and the two HumanEval variants each contribute separate entries. EM & LL Avg. is the mean of the two reported group averages, with equal weight assigned to each group. Accuracy differences and Δ in Table 4 are measured in percentage points. All compared runs use the same training and evaluation seed. Every result is evaluated from the final checkpoint of its run.

E Additional Details of the Optimization View

The following identities detail the local-gradient calculation in Proposition 1. At recurrent step l, consider $\boldsymbol{o}_l = \mathbf{H}_l^{\text{in}}\mathbf{Y}_l + \boldsymbol{b}_l$. For the local derivative with respect to $\mathbf{Y}_l$, hold $\mathbf{H}_l^{\text{in}}$, $\boldsymbol{b}_l$, and the predicted target $\boldsymbol{t}_l$ fixed. The linear and delta objectives then give:

$$\mathcal{L}_l^{\text{lin}} = -\langle \boldsymbol{o}_l, \boldsymbol{t}_l \rangle, \qquad \nabla_{\mathbf{Y}_l}\mathcal{L}_l^{\text{lin}} = -(\mathbf{H}_l^{\text{in}})^\top \boldsymbol{t}_l, \tag{17}$$

$$\mathcal{L}_l^{\text{delta}} = \tfrac{1}{2}\|\boldsymbol{o}_l - \boldsymbol{t}_l\|_2^2, \qquad \nabla_{\mathbf{Y}_l}\mathcal{L}_l^{\text{delta}} = (\mathbf{H}_l^{\text{in}})^\top(\boldsymbol{o}_l - \boldsymbol{t}_l). \tag{18}$$

Holding these quantities fixed is only a convention for deriving the closed-form forward transition. The target and state-dependent maps are recomputed across recurrent steps. End-to-end training does not detach the target, input map, or bias. Gradients propagate through their computation.

For the bias-free projection in Eq. 12, substitute $\boldsymbol{o}_l = \mathbf{H}_l^{\text{in}}\mathbf{Y}_l$ into the delta update. This gives

$$\begin{aligned}\mathbf{Y}_{l+1} &= (\mathbf{I} - \eta_l\mathbf{\Lambda}_l)\mathbf{Y}_l + \eta_l(\mathbf{H}_l^{\text{in}})^\top(\boldsymbol{t}_l - \mathbf{H}_l^{\text{in}}\mathbf{Y}_l) \\ &= \big(\mathbf{I} - \eta_l\big(\mathbf{\Lambda}_l + (\mathbf{H}_l^{\text{in}})^\top\mathbf{H}_l^{\text{in}}\big)\big)\mathbf{Y}_l + \eta_l(\mathbf{H}_l^{\text{in}})^\top\boldsymbol{t}_l.\end{aligned} \tag{19}$$

This recovers the residual-map form in Eq. 16, with $\mathbf{H}_l^{\text{pre}} = \mathbf{H}_l^{\text{in}}$ and $\mathbf{H}_l^{\text{post}} = \eta_l\mathbf{H}_l^{\text{in}}$. The stored write map therefore includes the step-size factor. The HyperLoop alignment study equates the stored input and write maps under a unit step size. OperLoop uses the adaptive step size η_l.

Here, a closed-form transition specifies one update of the looped state. It does not denote a closed-form minimizer of the local objective. The local objective differs from the causal language-modeling loss used to train model parameters. The loop step size η_l differs from the training learning rate. The decay operator $\mathbf{\Lambda}_l$ also differs from the training weight-decay coefficient in Table 10.