EXPLORATION OF FAST-SLOW LATENT RECURRENCE FOR TRAIN-SHORT, TEST-LONG GENERALIZATION

Anonymous authors
Paper under double-blind review

ABSTRACT

We study out-of-distribution generalization in streaming tasks where models are trained on short sequences but must operate over much longer, unknown horizons under bounded memory. Our focus is on a persistent fast–slow recurrent formulation in which a latent state is maintained across observations rather than reset at each stream step. For each incoming observation, the model performs multiple weight-shared latent updates with a recurrent core and then carries the resulting state forward to the next observation. This allows the model to maintain and refine a compact stream-level state without reprocessing a growing context. We evaluate this formulation across symbolic sequence prediction, supervised navigation, and partially observable reinforcement learning tasks. Across these settings, persistent latent recurrence improves OOD generalization over recurrent, state-space, and Transformer baselines. Through recurrent-core ablations, we identify architectural ingredients that are consistently associated with strong OOD performance, including state-dependent transitions and feature-wise nonlinear mixing. Together, these results highlight the value of revisiting persistent recurrence as an architectural bias for more generalizable sequence prediction. Code: `https://anonymous.4open.science/r/fastslow`.

1 INTRODUCTION

Humans can often extrapolate symbolic streams generated by a fixed rule far beyond the lengths observed during learning, or solve navigation puzzles of previously unseen size, after seeing only small, bounded examples. How can we endow machines with this capability? We posit that this is not merely a question of memory maintenance: a finite-size memory cannot indefinitely store or attend to the raw history of a stream whose horizon is not known in advance. Instead, the model must learn, from short observations, a compact task-relevant rule of the stream, such as its generative rule in symbolic streams, or a chain of subtasks in a puzzle that exists irrespective of stream length or environment size (Dehaene et al., 2022; Amalric et al., 2017). Recent advances in machine learning suggest a possible architectural route toward this kind of extrapolation: rather than storing the entire history, a model may learn to infer a compact latent procedure. Latent recurrent reasoning, which iterates a weight-shared block in latent space, has recently emerged as a way to build related capabilities directly into model architectures (Geiping et al., 2025; Wang et al., 2025; Jolicoeur-Martineau, 2025; Zhu et al., 2025; McLeish et al., 2025). This mechanism, *especially with recurrent blocks based on attention*, has been shown to represent programs and universal computation (Giannou et al., 2023; Xu & Sato, 2025) and, most relevantly to the present work, to learn and execute iterative algorithms at test time in the form of in-context regression, gradient-descent-like updates, and other pattern-driven procedures (Yang et al., 2024; Miyato et al., 2025; Saunshi et al., 2025). Yet the in-depth study of these methods has so far focused on *static input*: the loop iterates over a fixed input, and a naive application to unknown-horizon streaming tasks of the kind described above would require a context that grows with stream length.

We hypothesize that, by allowing the latent loop to persist alongside the stream, we can extend the ability of latent reasoning to capture, from a short sequence, the rules governing the input stream *and thereby enable OOD generalization at horizons far beyond training*. We therefore combine latent recurrent modeling with the following persistent latent recurrence, which updates the latent $\mathbf{Z}$ using a

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.

recurrent core block R:

$$\mathbf{Z}_s(t+1) = R\left(\mathbf{Z}_s(t),\, \mathbf{C}_s\right), \qquad \mathbf{Z}_{s+1}(0) = \mathbf{Z}_s(T), \qquad t = 0, \ldots, T-1, \tag{1}$$

where $s \in \{1, 2, \ldots\}$ indexes observations along the stream, t indexes the inner latent recurrence, and $\mathbf{C}_s = \mathrm{Encoder}(\mathbf{O}_s)$ is the feature of the observation $\mathbf{O}_s$ at step s. The core R is applied T times for each observation, and the latent is carried forward without reset (Figure 1). This describes a dynamical system on the latent variable that is not reset at every observation step, and it reduces to an RNN when $T = 1$; when $T > 1$, the same recurrent core is applied multiple times per observation, yielding a **weight-shared**, deep-transition type update between consecutive stream steps in the sense of (Pascanu et al., 2014). Indeed, related structures have appeared in broader forms: adaptive computation methods such as ACT (Graves, 2017) can in principle be applied to recurrent systems of this kind to modulate the amount of computation, while Schöne et al. (2025) explore analogous persistent updates with Mamba cores in pursuit of a balance between parallelism and representation power. The question still remains as to which designs of R support OOD generalization on streaming tasks, and whether the benefits of recent findings in latent recurrent reasoning can be extended to such tasks. Our contribution is not the introduction of recurrence per se, but the investigation of latent recurrent reasoning combined with this persistent streaming formulation, identifying effective designs of R for OOD generalization on sequential tasks requiring reusable rules. Empirically, we make two valuable observations:

- The transformer-based cores, as used in recent latent reasoning methods, yield strong OOD generalization, practically extending the merit of static counterparts. We find that our recurrent core family (i.e., multiple recurrence) plays an important role: models without this persistent fast–slow mechanism, including looped Transformers, do not exhibit OOD generalization.
- In designing R, the following two structures are particularly important: (i) *state-dependent* mixing of the tokens and (ii) nonlinear dimensional mixing. For example, an LSTM+FFN core also performs competitively with transformer-based cores.

Concretely, on the Dyck language (Hewitt et al., 2020; Merrill et al., 2024), a canonical state-tracking benchmark, our model of this family, trained on sequences of length ≤ 40, maintains $\approx 80\%$ token accuracy up to length 10^5, three orders of magnitude beyond training. Our models also outperform baselines such as Looped Transformer (Fan et al., 2025) on a limited-vision maze navigation task, showing that rules learned in small environments can transfer to larger ones. Examining the learned latents, we further find that they are highly structured: they form a low-dimensional manifold indexed by task-relevant discrete states (e.g., stack depth in Dyck), and the dynamics reuse representations, in that states sharing the same task-relevant features cluster together in latent space.

2 Related Work

Prior work on length extrapolation has explored improved positional encodings (Su et al., 2021; Press et al., 2022; Chen et al., 2023; Peng et al., 2024) and looped Transformers (Fan et al., 2025). Latent recurrent models repeatedly apply weight-shared layers to a fixed input (Dehghani et al., 2019; Geiping et al., 2025; Miyato et al., 2025; Blayney et al., 2026), and related studies investigate their ability to implement reusable programs or rules (Giannou et al., 2023; Xu & Sato, 2025; Yang et al., 2024; Fan et al., 2025; Schwarzschild et al., 2021). Our focus is on extending latent refinement to evolving streams by carrying a fixed-size latent state across observations.

Persistent processing across multiple timescales is inspired by biological systems that integrate ongoing stimuli over different temporal windows (Murray et al., 2014; Zeraati et al., 2023; Huntenburg et al., 2018). HRM (Wang et al., 2025) and CTM (Darlow et al., 2025) also motivate latent recurrence from biological principles, whereas our study centers on short-to-long length generalization with a persistent stream-level state. Coconut (Hao et al., 2024) maintains latent thoughts within a growing conditional context; our formulation instead propagates a bounded latent state. From a complementary perspective, Mozer et al. (2026) motivate recurrent dynamics by highlighting the limitations of the feedforward topology of Transformers for indefinite state tracking.

Our formulation can be viewed as a nonlinear RNN with inner updates and, for $T > 1$, weight-tied deep transitions (Pascanu et al., 2014). Related recurrent designs use Mamba (Schöne et al., 2025) and DeltaNet (Siems et al., 2025) cores. Alongside work on expressivity and parallelism trade-offs (Merrill et al., 2024; Merrill & Sabharwal, 2023), we investigate whether fixed weights

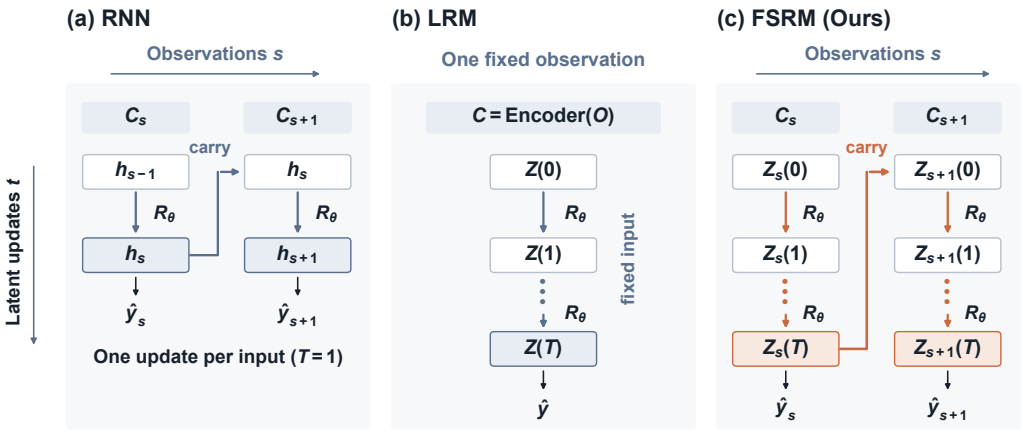


Figure 1: **RNN, latent recurrent model (LRM), and FSRM.** (a) An RNN updates once per observation and carries its state forward. (b) An LRM repeatedly refines a latent state under one fixed encoded input. (c) FSRM performs T updates under each fixed $\mathbf{C}_s$ and carries the final state to the next observation, $\mathbf{Z}_{s+1}(0) = \mathbf{Z}_s(T)$. The horizontal direction indexes observations s; downward latent updates index t. Final states are decoded into predictions. Recurrent parameters are shared across updates within each model; intermediate updates are omitted.

learned from short sequences generalize to much longer horizons. These are distinct questions: bounded-depth Dyck languages admit fixed-depth self-attention constructions (Yao et al., 2021), but representability does not ensure that training on short sequences yields length-generalizing solutions (Anil et al., 2022; Delétang et al., 2023). Our experiments examine which recurrent-core designs enable such length generalization within the persistent fast–slow formulation.

3 Fast–Slow Latent Recurrence

3.1 Persistent fast–slow recurrence

LRM. A latent recurrent model (LRM) (Gladstone et al., 2025; Geiping et al., 2025; Miyato et al., 2025; Fan et al., 2025; Yang et al., 2024) encodes a static observation of fixed length as $\mathbf{C} = \mathrm{Encoder}(\mathbf{O})$. Starting from a randomly initialized latent $\mathbf{Z}(0) = \mathbf{Z}^{\mathrm{init}} \in \mathbb{R}^{n\times d}$, with n tokens of dimension d, it applies a shared recurrent core for $t = 0, \ldots, T-1$:

$$\mathbf{Z}(t+1) = R\left(\mathbf{Z}(t), \mathbf{C}\right) \tag{2}$$

The encoded input $\mathbf{C}$ remains fixed throughout these updates, and the final state is passed to $\mathrm{Decoder}(\mathbf{Z}(T))$ for prediction.

FSRM. The fast–slow recurrent model (FSRM) extends this recurrence to a stream $\mathbf{O}_1, \mathbf{O}_2, \ldots$ by updating the encoded input at each observation and carrying the latent state forward (Figure 1). At the start of the stream, we initialize $\mathbf{Z}_1(0) = \mathbf{Z}^{\mathrm{init}} \in \mathbb{R}^{n\times d}$ from a Gaussian distribution and apply

$$\begin{aligned} \mathbf{C}_s &= \mathrm{Encoder}(\mathbf{O}_s), \\ \mathbf{Z}_s(t+1) &= R\left(\mathbf{Z}_s(t), \mathbf{C}_s; \theta\right), \quad t = 0, \ldots, T-1, \\ \mathbf{Z}_{s+1}(0) &= \mathbf{Z}_s(T), \qquad\qquad s = 1, 2, \ldots. \end{aligned} \tag{3}$$

Here, s indexes observations and t indexes latent updates within each observation. The encoded input $\mathbf{C}_s$ remains fixed for T fast updates. We then decode $\mathbf{Z}_s(T)$ and carry it to the next observation without reinitialization. The core parameters θ are shared across both timescales. Holding $\mathbf{C}_s = \mathbf{C}$ constant makes FSRM equivalent to continuing the static-input LRM recurrence across successive blocks of T updates. With $T = 1$, FSRM performs a single RNN-style update per observation; with $T > 1$, it combines LRM-style refinement within each observation with RNN-style memory across the stream.

3.2 RECURRENT-CORE DESIGN

State-dependent transition and feature-wise dimensional mixing are established architectural concepts: prior work has studied dynamic token mixing (Wang et al., 2022; Wei et al., 2023) and the distinction between token and channel mixing (Tolstikhin et al., 2021). Our contribution is to investigate how these properties affect OOD generalization of learned recurrent updates within the persistent fast–slow formulation. We compare recurrent cores while keeping the streaming mechanism, encoder, decoder, number of fast updates, and training protocol fixed. This is an empirical investigation of generalization after training on short sequences, rather than a claim about which functions these architectures can represent.

Core examples. The recurrent core R can be instantiated with modules used in static-input LRMs, including Transformer and AKOrN blocks. Geiping et al. (2025) instantiate R with a Transformer block, for example,

$$R_{TF}(\mathbf{Z}(t), \mathbf{C}) = n_4(\tilde{\mathbf{Z}} + \mathbf{FFN}(n_3(\tilde{\mathbf{Z}})) \quad \text{where} \quad \tilde{\mathbf{Z}} = n_2(\mathbf{Z}(t) + \text{Attn}(n_1(h[\mathbf{Z}(t), \mathbf{C}])))) \tag{4}$$

where Attn denotes the usual attention mechanism used in prior latent-reasoning models (Blayney et al., 2026; Geiping et al., 2025), h is an adapter that merges $\mathbf{Z}$ with the observation, the n_k denote normalization operators (e.g., LayerNorm or RMSNorm), and **FFN** acts as a token-wise MLP map $\mathbb{R}^d \to \mathbb{R}^d$. Another recent model in this family is AKOrN (Miyato et al., 2025). (Miyato et al., 2025), which constrains $\mathbf{Z}$ to be a set of oscillating tokens on the hypersphere S^{d-1} by projecting the update $\mathbf{Z}(t+1) - \mathbf{Z}(t)$ onto the tangent space of S^{d-1} and by adding rotational momentum. We mention AKOrN because we found it to be a consistently strong and stable choice for R throughout our experiments. See Appendix A.1.1 for details. We define the two properties below and summarize the tested cores in Table 1. Because $\mathbf{Z}$ is the evolving recurrent state, we use the term *input* exclusively for $\mathbf{C}_s$.

(i) State-dependent transition. Let $\mathbf{Z} \in \mathbb{R}^{n\times d}$ be the latent, with n tokens of d-dimensional features, and let $\mathbf{C}$ denote the encoded observation. We say the transition $R(\,\cdot\,, \mathbf{C}) : \mathbb{R}^{n\times d} \to \mathbb{R}^{n\times d}$ has a *state-dependent transition* if it contains a linear operator (linear in some part of $\mathbf{Z}$) whose coefficients themselves depend on $\mathbf{Z}$. Equivalently, at the Jacobian level, this condition requires that $\partial_{\mathbf{Z}} R(\mathbf{Z}, \mathbf{C})$ does not admit a factorization into fixed (state-independent) linear maps composed with diagonal matrices arising from pointwise nonlinearities. This excludes vanilla RNN-style cores $\sigma(A\mathbf{Z} + B\mathbf{C})$, in which all linear operations have coefficients independent of $\mathbf{Z}$ and all $\mathbf{Z}$-dependence is mediated by pointwise nonlinearities. The canonical example of state-dependent transition is attention: the linear operator $\mathrm{AttnMat}_\theta(\mathbf{Z}) \in \mathbb{R}^{n\times n}$ acting on $\mathbf{Z}W_V$ is itself a function of $\mathbf{Z}$. This distinction from state-independent transition has been made in prior work on dynamic/adaptive mixing (Wang et al., 2022; Wei et al., 2023), where an update of the form $A(\mathbf{Z})\mathbf{Z}$ is used. Input-controlled selectivity in SSMs has a separate expressivity theory (Muca Cirone et al., 2024); condition (i) here concerns dependence on the latent state $\mathbf{Z}$.

(ii) Feature-wise dimensional mixing. Treating $\mathbf{Z} \in \mathbb{R}^{n\times d}$ as n rows $\mathbf{Z}_n \in \mathbb{R}^d$, dimensional mixing is the family of maps $\mathbb{R}^d \to \mathbb{R}^d$ that act *within* each row by mixing its d feature coordinates. The canonical instance is the per-token MLP. Feature-wise affine modulation (Dumoulin et al., 2018) conditions individual coordinates; it need not mix coordinates as required by (ii). Note that mixing across the *token* axis n (e.g. attention) is a different operation and does not by itself realize (ii). Tolstikhin et al. (2021) make this distinction explicit: MLP-Mixer combines a state-independent token mixer with a channel mixer.

4 EXPERIMENTAL SETTINGS

We conduct experiments to verify out-of-distribution (OOD) generalization across three settings: symbolic stream prediction on the Dyck language (Schützenberger, 1963), navigation of a maze under egocentric observation (local maze), and reinforcement learning on Minigrid (Chevalier-Boisvert et al., 2023). We use an AKOrN + FFN core with $T = 5$ for Dyck and Local-Maze and $T = 10$ for MiniGrid, unless otherwise specified. Implementation details are provided in Appendix A.1.1. Results report means and standard deviations over 3 random seeds (5 for Minigrid), shown as shaded regions or error bars.

Dyck-$(30, 5)$. We use Dyck-$(30, 5)$ (Hewitt et al., 2020), where 30 is the number of bracket types and 5 is the maximum stack depth, to test whether a model can infer a stack rule from short

Table 1: Classification of recurrent cores R with respect to (i) state-dependent transition and (ii) feature-wise dimensional mixing; ✓ indicates the property is satisfied. For our LSTM/Mamba2 cores, we use multiple internal updates per t, treating $\mathbf{Z} + \mathbf{C}$ as the external input so the latent-dependent gates, projections, and channel mixing are applied multiple times when producing $\mathbf{Z}(t+1)$. See Appendix A.1.2 for details.

Core	(i)	(ii)	Mechanism / remark
Conv	✗	✗	State-independent token/pixel mixing with linear channel mixing.
Attention	✓	✗	$\mathrm{AttnMat}_\theta(\mathbf{Z})$ depends on $\mathbf{Z}$; no per-token FFN.
MLP-Mixer	✗	✓	Mixer weights are independent of $\mathbf{Z}$; MLP-style depth-2 mixing.
LSTM (loops per t)	✓	✓	Gated update over $\mathbf{Z} + \mathbf{C}$, repeated multiple times per t.
Mamba2 (loops per t)	✓	✓	Selective/gated SSM over $\mathbf{Z} + \mathbf{C}$, repeated multiple times per t.
Transformer	✓	✓	Attention gives (i); FFN gives (ii).
Linear Transformer	✓	✓	Transformer with linear attention.
AKOrN + FFN	✓	✓	Attention-derived couplings dependent on $\mathbf{Z}$.

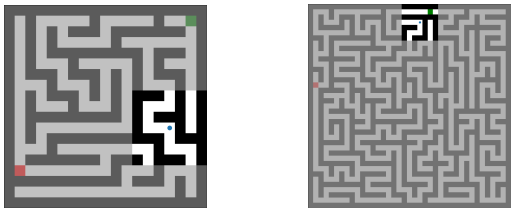
input `()[{}()<()>][...`
target `)*]}])]>)>]*]...`

(a) ID (random pattern)

input `[()()()()()()...`
target `])])])])])])])]...`

(b) OOD (1-regular run)

Figure 2: Dyck-$(30, 5)$ examples. (a) The target is the closing token for the current stack top, or "$*$" when the stack is empty. (b) A 1-regular run requires retaining the first unresolved bracket over a long repetition.



(a) ID (19×19) (b) OOD (39×39)

Figure 3: Local-Maze examples. Models are trained on small mazes (a) and evaluated on larger ones (b). The green and red cells denote the start and the goal. The observation is limited to a 7×7 region centered on the current position.

streams of length at most 40 and apply it to much longer horizons. At each position, the model predicts the closing bracket for the current stack top, or "$*$" when the stack is empty. Training and ID evaluation use randomly generated Dyck-$(30, 5)$ strings; the ID curves extend this same random-pattern distribution beyond the training length, testing length generalization. OOD evaluation additionally changes the data pattern to 1-regular runs with repeated open–close blocks (Figure 2), so the Dyck setting tests both longer horizons and generalization from random patterns to regular-run patterns; Appendix B.6 gives the full protocol.

Local-Maze. Local-Maze extends the navigation task of Darlow et al. (2025) and tests whether a geometric rule learned on 19×19 mazes transfers to larger 39×39 mazes. The model observes a stream of egocentric 7×7 views collected by a fixed right-hand-rule navigator and predicts the shortest start-to-goal action sequence. Because the global position is not directly observable, the task requires integrating local observations into a maze-level plan; Appendix B.5 gives the full protocol and supervision details.

Minigrid. We use three partially observable RL tasks from Minigrid (Chevalier-Boisvert et al., 2023): DoorKey, MultiRoom, and LavaCrossing. Agents are trained on a simple configuration and evaluated zero-shot on larger or more complex OOD environments, using sparse rewards and egocentric 7×7 observations. Unlike Dyck-$(30, 5)$ and Local-Maze, no ground-truth action sequence is provided; Appendix B.7 gives the task definitions, action space, and ID/OOD splits.

Baselines. We choose baselines to isolate the two ingredients of FSRM: carrying a bounded stream-level state and refining it through multiple weight-shared latent updates before the next observation. Recurrent, state-space, and cached-memory sequence models (LSTM (Hochreiter & Schmidhuber, 1997), Mamba2/S5 (Gu & Dao, 2023; Dao & Gu, 2024; Smith et al., 2022), DeltaProduct (Siems et al., 2025), and TransformerXL (Dai et al., 2019)) all test whether sequence memory alone is sufficient. Transformer variants (Vaswani et al., 2017) and LLM/Qwen baselines test context-based alternatives based on attention, prompting, pretraining, scale, or fine-tuning. Looped TF (Fan et al.,

2025) and CTM (Darlow et al., 2025) test whether latent or recurrent computation without the same persistent fast–slow streaming formulation is enough. Baselines are task-matched: Dyck-$(30, 5)$ uses LSTM, Mamba2, Transformer variants, prompted frontier LLMs, and fine-tuned Qwen3-4B-Base; Local-Maze uses LSTM, Mamba2, Transformer (TF), Looped TF, DeltaProduct, and TRM (Jolicoeur-Martineau, 2025), with S5 and CTM in the appendix; and Minigrid uses LSTM, Mamba2, and TransformerXL under the same PPO setup. Checkpoints are selected by in-distribution validation after a shared learning-rate sweep, with architecture-specific hyperparameters reported in B.2 and Tables 4–5. B.1 gives the full training protocol.

Investigated Questions: Altogether, we tackle the following four questions: **RQ1.** Does the persistent fast–slow loop improve OOD generalization over baseline approaches, including methods designed for static inputs? **RQ2.** How does the number of inner-loop iterations per observation affect OOD performance? **RQ3.** Is the gain due to weight-shared recurrence rather than simply adding heterogeneous, unshared depth? **RQ4.** Which recurrent-core designs preserve the OOD gains? The subsections answer these questions in order: we first compare OOD performance across tasks, then isolate the roles of inner-loop iterations and weight sharing, and finally ablate the recurrent core.

5 Results and Ablations

Across the streaming tasks, extending latent recurrence with the FSRM fast–slow loop improves OOD generalization over the corresponding one-step sequence-model baselines. The strongest results come from cores that combine state-dependent transition with sufficient feature-wise mixing, although the value of adding a separate FFN depends on the task and on the core's native inductive bias.

5.1 OOD generalization across tasks

In all settings (Dyck-$(30, 5)$, Local-Maze and Minigrid), FSRM with an attention-based recurrent core (A.1.1) substantially improves OOD generalization over the main sequence-model baselines.

On Dyck-$(30, 5)$ (Figure 5a), the baselines degrade soon after OOD lengths exceed the training horizon, whereas FSRM maintains high accuracy at substantially longer horizons. On Local-Maze (Figure 5b), FSRM attains higher mean OOD accuracy than the baselines. Additional experiments control for compute (Figure 10) and parameter count (Appendix C.7), and extend the evaluation to language modeling (Appendix C.5). On the Minigrid reinforcement-learning tasks (Figure 5c–e), FSRM also achieves higher OOD performance than the baselines, including TransformerXL (Dai et al., 2019), on environments larger than those seen during training.

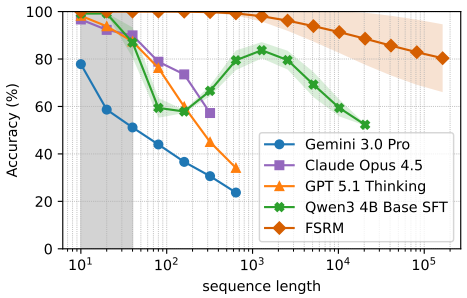
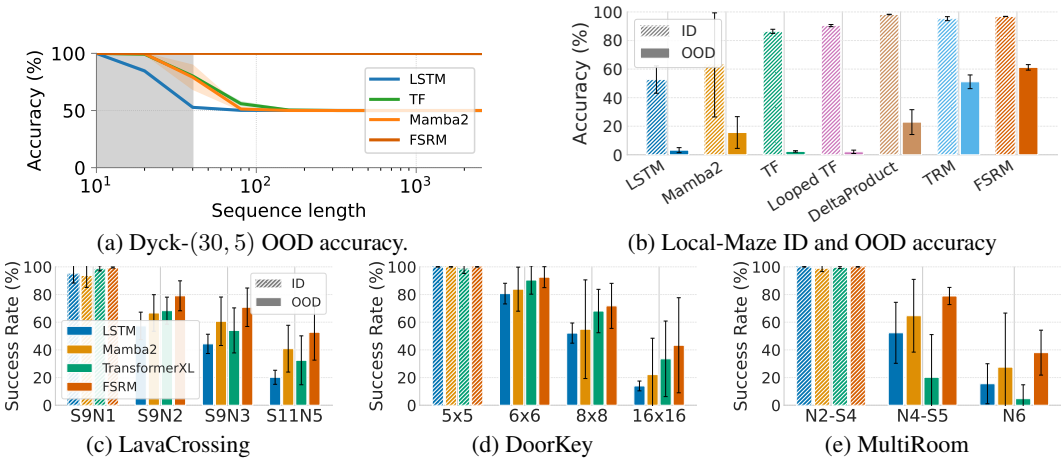


Figure 4: Dyck-$(30, 5)$ accuracy vs. length. LLMs use algorithmic prompts. FSRM and Qwen3-4B-Base are trained only up to length 40 (shaded area); FSRM generalizes to horizons far beyond training.

As a stronger stress test on Dyck-$(30, 5)$, we extend the OOD evaluation horizon by another order of magnitude and compare against LLM-based baselines in Figure 4. The frontier LLMs are evaluated zero-shot with algorithmic hints, but their accuracy degrades rapidly as the sequence length increases; moreover, their internal reasoning uses up much of the output-token budget, and they can only be evaluated reliably up to lengths on the order of 10^3. We therefore also fine-tune the open-weight Qwen3-4B-Base model on the same Dyck-$(30, 5)$ split and evaluate it under the same protocol. The fine-tuned Qwen remains accurate over longer horizons than the frontier LLMs, but still falls to the chance rate of approximately 50% at length $20{,}480$, roughly an order of magnitude shorter than the horizon at which FSRM maintains high accuracy.

5.2 Fast-loop structure: T and weight sharing

Effect of fast-loop iterations. The number of fast inner-loop iterations T controls how many recurrent updates of R occur between successive observations. We sweep T at training time while



(a) Dyck-$(30, 5)$ OOD accuracy.

(b) Local-Maze ID and OOD accuracy

(c) LavaCrossing

(d) DoorKey

(e) MultiRoom

Figure 5: Main OOD generalization results. (a) Dyck-$(30, 5)$ OOD accuracy beyond the training range (shaded area); the corresponding ID curve is in Appendix C.1.1. (b) Local-Maze ID and OOD accuracy. (c–e) Minigrid success rates.

keeping all other components fixed. As shown in Table 2, OOD accuracy improves with T, with diminishing returns at the largest T tested. The preferred T is task-dependent; see Appendix C.4 for Dyck and LavaCrossing ablations. Figure 23 in the Appendix reports the complementary test-time scaling experiment, where T is varied at inference after training.

Effect of weight sharing in the fast loop. FSRM applies the same update rule R repeatedly for T inner steps; that is, it is *recurrent* in the sense that the update is *weight-shared* across fast iterations. To isolate the contribution of this sharing, we replace the T shared iterations with an *unrolled* residual stack of T distinct layers applied once per observation—a construction equivalent to a deep-transition RNN (Pascanu et al., 2014)—keeping the overall fast–slow interface unchanged. Table 2 shows that weight sharing is crucial for OOD generalization. With the same inner-loop budget ($T = 5$), the shared recurrent loop reaches roughly 61% OOD accuracy, whereas the unshared 5-layer stack drops to about 51%, despite using roughly twice as many parameters. Increasing the unshared stack to 10 layers does not close this gap; instead, OOD accuracy falls further to around 45%, with substantially higher variance. Together, these results support the design choice that *looped recurrence*—not added depth or parameter count—is the effective ingredient of FSRM, aligning with prior work showing that weight-shared looped computation can learn reusable iterative procedures (Saunshi et al., 2025).

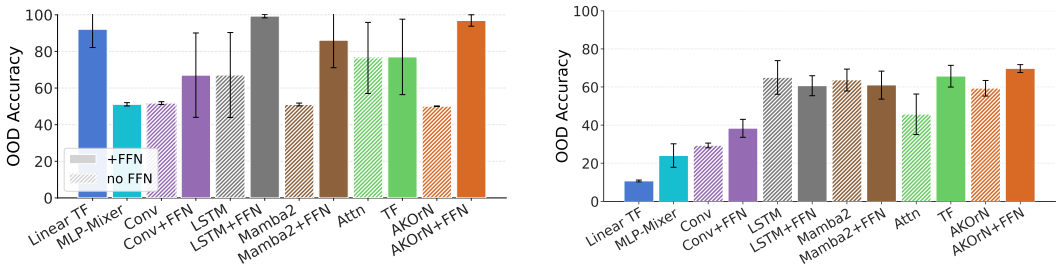
Table 2: Fast-loop ablations on Local-Maze. Shared-loop rows vary the number of inner-loop updates per observation ($1 \otimes T$), while stack rows replace the shared loop with L unshared layers applied once per observation ($L \otimes 1$). The Params/FLOPs column reports parameters and FLOPs relative to Base ($1 \otimes 1$).

Model	Params/ FLOPs	OOD Acc.
Shared loop: train-time T sweep		
Base $(1 \otimes 1)$	$1\times/1\times$	0.306 ± 0.135
Loop $(1 \otimes 2)$	$1\times/2\times$	0.431 ± 0.186
Loop $(1 \otimes 4)$	$1\times/4\times$	0.595 ± 0.036
Loop $(1 \otimes 5)$	$1\times/5\times$	0.612 ± 0.019
Loop $(1 \otimes 8)$	$1\times/8\times$	$\mathbf{0.673 \pm 0.006}$
Loop $(1 \otimes 16)$	$1\times/16\times$	0.673 ± 0.032
Weight-sharing control		
Stack $(5 \otimes 1)$	$2\times/5\times$	0.509 ± 0.098
Stack $(10 \otimes 1)$	$3\times/10\times$	0.445 ± 0.164

5.3 Core design

We ablate the design of R directly while keeping the encoder, decoder, T, and training protocol fixed. We tested the variants of R classified in Table 1.[1] Here, "+FFN" denotes an additional FFN wrapper, e.g. $R_{\mathrm{X+FFN}} = \mathrm{FFN} \circ R_X$. AKOrN (Miyato et al., 2025) is a recently introduced mechanism based on the dynamics of oscillating token particles on the hypersphere, compatible with

[1]The Local-Maze core ablation uses a slightly larger model than the preceding experiments; consequently, its absolute accuracy values are slightly higher, and comparisons should be read primarily within this ablation.



(a) Dyck-$(30, 5)$ OOD accuracy.

(b) Local-Maze OOD accuracy (39×39).

Figure 6: Core ablations on (a) Dyck-$(30, 5)$ at lengths 1,280–2,560 and (b) Local-Maze under OOD evaluation. Solid bars include the native or augmented FFN block; hatched bars ablate it. ID results are in C.1.4 and C.2.1.

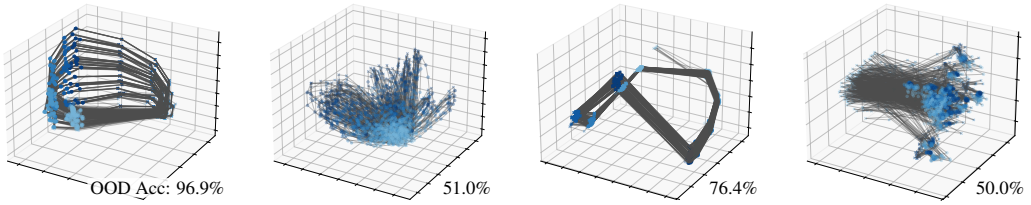
an extension of a Transformer-like mechanism (see Appendix A.1.1). For the LSTM and Mamba2 core designs, we use inputs with chunk size L and additionally loop L times within each transition $\mathbf{Z}_s(t) \to \mathbf{Z}_s(t+1) \in \mathbb{R}^{L \times d}$. Since they apply latent-dependent gates and channel modulation across L loops, they satisfy (i) and (ii) as well; see A.1.2. Figure 6 ablates the recurrent core under fixed encoder, decoder, T, and training protocol. We focus on the two axes of Section 3.2: state-dependent transition (i) and feature-wise dimensional mixing (ii).

Table 3: OOD accuracy of recurrent cores grouped by the presence or absence of (i) state-dependent transition and (ii) feature-wise dimensional mixing. Each cell reports the mean OOD accuracy of the cores in that group.

Task	State-dependent transition	No dimensional mixing	Dimensional mixing
Local-Maze	No	0.201 ± 0.132	0.236 ± 0.149
	Yes	0.451 ± 0.004	$\mathbf{0.586 \pm 0.046}$
Dyck	No	0.512 ± 0.000	0.534 ± 0.039
	Yes	0.521 ± 0.029	$\mathbf{0.726 \pm 0.195}$

Table 3 summarizes OOD accuracy after grouping the tested recurrent cores along these two axes. On both tasks, the group combining state-dependent transition and feature-wise dimensional mixing achieves the highest mean accuracy. This pattern suggests that the two properties may be complementary architectural ingredients in the persistent fast–slow formulation. We do not interpret either property, or their combination, as formally necessary or sufficient for OOD generalization: performance remains dependent on the task and the specific core, as illustrated by the FFN ablations in Figure 6. As for the importance of (i), MLP-Mixer supplies a feature-wise mixer (ii), but its token mixer is state-independent, and its OOD accuracy stays well below the stronger cores. This conclusion is the empirical counterpart of the distinction Section 3.2 draws from Tolstikhin et al. (2021) and the dynamic-mixing line of Wang et al. (2022) and Wei et al. (2023). The importance of (ii) appears in the FFN ablation, which toggles an explicit feature-wise mixer while keeping the rest of the core fixed. On Local-Maze, cores whose native update is light on per-token mixing, namely Conv, Attn, and AKOrN, benefit substantially from an external FFN, while cores that already realize (ii) internally, LSTM and Mamba2, change only modestly when an FFN is appended, and slightly degrade in our runs. On Dyck-$(30, 5)$, by contrast, the FFN gain extends to LSTM and Mamba2 as well, both of which improve markedly once an FFN is appended. This task dependence is consistent with our opening observation that some tasks demand stronger (ii) than others: on Local-Maze, the gating and projections internal to LSTM and Mamba2 already supply enough feature-wise mixing, whereas Dyck benefits from the additional explicit mixer.

We also note that, in the design of the core, increasing representation power yields a meaningful gain (Table 8). While the parameter count does not change the model's complexity class in theory, our result *does* show that representation power is an important factor for learning OOD-generalizable patterns. Thus, we can also interpret the strength of combining (i) and (ii) as efficient use of parameters along directions of representation power that serve OOD generalization. It is also worth re-emphasizing that, although Dyck-$(30, 5)$ is known to be expressible by fixed-depth self-attention networks (Yao et al., 2021), the Transformer itself fails to OOD-generalize on this task (Figure 5a), whereas Rs based on TF within the mechanism (1), including AKOrN+FFN, succeed. Together,



(a) FSRM-AKOrN+FFN (b) FSRM-Mamba2 (c) FSRM-Attention (d) Mamba2

Figure 7: PCA visualization of latent traversals. Each point is $\mathbf{Z}_s(T)$: $\circ$/$\times$ indicate opening/closing brackets, and darker colors indicate greater stack depth. (a) FSRM with an AKOrN core and FFN forms an organized traversal that revisits task-aligned states across repeated stack updates. (b) FSRM with a Mamba2 core, (c) FSRM with an attention core, and (d) standalone Mamba2 do not exhibit the same coherent traversal under this visualization.

these observations suggest that the mechanism (1), combined with sufficient representation power, is important for fully realizing the module's expressivity potential.

6 Qualitative analysis: Emergent structure in latents

We next inspect whether the recurrent state organizes around variables that are relevant to the stream rule. On Dyck-$(30, 5)$, Figure 7 visualizes latent trajectories along a 5-regular run, an OOD string with a repeated open–close pattern. In the coherent configuration, FSRM with an AKOrN+FFN core forms an organized traversal that revisits task-aligned states across repeated stack updates, suggesting alignment with the stack dynamics of the Dyck language rule. By contrast, the ablated FSRM variants without the same combination of persistent looping and task-appropriate feature-wise mixing, as well as standalone Mamba2 without the persistent fast–slow loop, produce less coherent trajectories. These qualitative differences suggest that task-aligned latent organization is encouraged by the persistent loop together with an expressive recurrent core. This Dyck interpretation is supported more directly by our *stack-content probing* analysis: a linear readout of the j-th unresolved opener from frozen hidden states on OOD data shows that FSRM captures stack structure more faithfully than baselines lacking the fast–slow mechanism (1). See Appendix C.1.3 and Figures 18 and 19 for details. We observe an analogous organization in Minigrid: DoorKey-16x16 latent trajectories cluster around task-relevant events across episodes; see C.3.1 for the PCA visualization and accompanying energy trace. These emergent structures are consistent with recent findings that looped attention-based computations converge to cyclic fixed points in latent space (Blayney et al., 2026), and that continuous-time attention dynamics exhibit metastable clustering (Geshkovski et al., 2024; 2025).

7 Conclusion and Limitations

We studied a fast–slow recurrent formulation that combines persistent recurrence over a stream with various recurrent cores, and empirically found that cores with state-dependent transition and dimensional mixing are particularly strong for OOD generalization on streaming tasks. By performing multiple latent updates per observation and propagating the resulting state without reset, our study adapts latent reasoning modules originally designed for static inputs to streaming settings, extending the benefits observed in prior work on static-input inference and better leveraging the core's expressivity. The present study also has several limitations. First, the proposed formulation trades parallelism for expressivity. Because the recurrent core is iterated multiple times per observation, inference is slower than highly optimized recurrent or state-space baselines, and our current implementation does not yet exploit specialized GPU kernels or recursion-aware parallelization (Appendix C.8). Second, our experiments focus on settings where successful generalization can plausibly be supported by compact stream-level structure. The method should therefore not be interpreted as guaranteeing correct computation for arbitrary horizons, arbitrary distributions, or general long-context reasoning problems. Finally, we fixed or controlled the number of inner-loop iterations in order to isolate the effect of the recurrent core. Combining persistent fast–slow recurrence with adaptive halting or learned compute allocation, and understanding when additional inner-loop computation should be spent, remain important directions for future work.

AI USE STATEMENT

We used generative AI tools to assist with manuscript refinement and the implementation of parts of the experimental code. The authors reviewed the AI-assisted experimental code. During manuscript preparation and revision, we also used these tools for translation, literature search, checking the consistency of definitions and interpretations of results, and developing revision suggestions. Additional uses included creating and modifying schematic figures and plotting code, editing LaTeX, and checking document layout and cross-references. The authors take responsibility for the final content of this work, including its text, claims, code, figures, and results.

ETHICS STATEMENT

The main experiments study symbolic sequence prediction on synthetic Dyck strings, supervised navigation in procedurally generated mazes, and reinforcement learning in simulated MiniGrid environments. These tasks concern algorithmic and simulated behavior rather than decisions about individuals. Existing models, software, and datasets are identified in the experimental descriptions, with licenses and terms for the principal assets summarized in Appendix B.3. The reported results concern controlled benchmarks and do not establish reliability for deployment in real-world or safety-critical settings. The additional computation required by repeated latent updates is an important practical cost; we report compute resources in Appendix B.4 and training and inference costs in Appendices C.8 and C.3.3.

REPRODUCIBILITY STATEMENT

Section 3 defines the persistent fast–slow recurrence and the recurrent-core properties studied in this work. Appendix A.1 describes the core implementations, and Appendix B.6.3 provides the hierarchical Dyck architecture and pseudocode. Training protocols and hyperparameters are reported in Appendices B.1 and B.2 and Tables 4–5. Task construction, supervision, and evaluation protocols are described in Appendices B.5, B.6, and B.7; the Dyck evaluation description also specifies the prompted-LLM and fine-tuned-Qwen settings. Appendix C reports additional ablations and controls, and Appendix B.4 identifies the hardware used. These descriptions accompany the experimental settings and variability reporting in Section 4.

REFERENCES

Marie Amalric, Liping Wang, Pierre Pica, Santiago Figueira, Mariano Sigman, and Stanislas Dehaene. The language of geometry: Fast comprehension of geometrical primitives and rules in human adults and preschoolers. *PLOS Computational Biology*, 13(1):e1005273, 2017. doi: 10.1371/journal.pcbi.1005273.

Cem Anil, Yuhuai Wu, Anders Andreassen, Aitor Lewkowycz, Vedant Misra, Vinay Ramasesh, Ambrose Slone, Guy Gur-Ari, Ethan Dyer, and Behnam Neyshabur. Exploring length generalization in large language models. In *Advances in Neural Information Processing Systems*, 2022. URL https://openreview.net/forum?id=zSkYVeX7bC4.

Hugh Blayney, Álvaro Arroyo, Johan Obando-Ceron, Pablo Samuel Castro, Aaron Courville, Michael M. Bronstein, and Xiaowen Dong. A mechanistic analysis of looped reasoning language models. *arXiv preprint arXiv:2604.11791*, 2026.

Hyungjoo Chae, Yeonghyeon Kim, Seungone Kim, Kai Tzu-iunn Ong, Beong-woo Kwak, Moohyeon Kim, Sunghwan Kim, Taeyoon Kwon, Jiwan Chung, Youngjae Yu, et al. Language models as compilers: Simulating pseudocode execution improves algorithmic reasoning in language models. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 22471–22502, 2024.

Shouyuan Chen, Sherman Wong, Liangjian Chen, and Yuandong Tian. Extending context window of large language models via positional interpolation, 2023.

Maxime Chevalier-Boisvert, Bolun Dai, Mark Towers, Rodrigo de Lazcano, Lucas Willems, Salem Lahlou, Suman Pal, Pablo Samuel Castro, and Jordan Terry. Minigrid & miniworld: modular &

customizable reinforcement learning environments for goal-oriented tasks. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA, 2023. Curran Associates Inc.

Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc V Le, and Ruslan Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context. *arXiv preprint arXiv:1901.02860*, 2019.

Tri Dao and Albert Gu. Transformers are ssms: Generalized models and efficient algorithms through structured state space duality. *arXiv preprint arXiv:2405.21060*, 2024.

Luke Darlow, Ciaran Regan, Sebastian Risi, Jeffrey Seely, and Llion Jones. Continuous thought machines, 2025. URL https://arxiv.org/abs/2505.05522.

Stanislas Dehaene, Fosca Al Roumi, Yair Lakretz, Samuel Planton, and Mathias Sablé-Meyer. Symbols and mental programs: a hypothesis about human singularity. *Trends in Cognitive Sciences*, 26(9):751–766, 2022. doi: 10.1016/j.tics.2022.06.010.

Mostafa Dehghani, Stephan Gouws, Oriol Vinyals, Jakob Uszkoreit, and Lukasz Kaiser. Universal transformers. In *International Conference on Learning Representations (ICLR)*, 2019. URL https://openreview.net/forum?id=HyzdRiR9Y7. OpenReview preprint.

Grégoire Delétang, Anian Ruoss, Jordi Grau-Moya, Tim Genewein, Li Kevin Wenliang, Elliot Catt, Chris Cundy, Marcus Hutter, Shane Legg, Joel Veness, and Pedro A. Ortega. Neural networks and the chomsky hierarchy, 2023. URL https://arxiv.org/abs/2207.02098.

Tianqi Du, Haotian Huang, Yifei Wang, and Yisen Wang. Long-short alignment for effective long-context modeling in LLMs. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 14515–14535. PMLR, 2025. URL https://proceedings.mlr.press/v267/du25a.html.

Vincent Dumoulin, Ethan Perez, Nathan Schucher, Florian Strub, Harm de Vries, Aaron Courville, and Yoshua Bengio. Feature-wise transformations. *Distill*, 2018. doi: 10.23915/distill.00011. https://distill.pub/2018/feature-wise-transformations.

Ying Fan, Yilun Du, Kannan Ramchandran, and Kangwook Lee. Looped transformers for length generalization. In *International Conference on Learning Representations (ICLR)*, 2025. URL https://arxiv.org/abs/2409.15647.

Jonas Geiping, Sean Michael 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. In *ES-FoMo III Spotlight (ICML 2025 Workshop)*, June 2025. URL https://openreview.net/forum?id=D6o6Bwtq7h. OpenReview preprint / workshop version.

Borjan Geshkovski, Hugo Koubbi, Yury Polyanskiy, and Philippe Rigollet. Dynamic metastability in the self-attention model. *arXiv preprint arXiv:2410.06833*, 2024. URL https://arxiv.org/pdf/2410.06833.

Borjan Geshkovski, Cyril Letrouit, Yury Polyanskiy, and Philippe Rigollet. A mathematical perspective on transformers. *Bulletin of the American Mathematical Society*, 62(3):427–479, 2025.

Angeliki Giannou, Shashank Rajput, Jy-yong Sohn, Kangwook Lee, Jason D. Lee, and Dimitris Papailiopoulos. Looped transformers as programmable computers. In *International Conference on Machine Learning*, pp. 11398–11442, 2023. arXiv:2301.13196.

Alexi Gladstone, Ganesh Nanduru, Md Mofijul Islam, Peixuan Han, Hyeonjeong Ha, Aman Chadha, Yilun Du, Heng Ji, Jundong Li, and Tariq Iqbal. Energy-based transformers are scalable learners and thinkers. *arXiv preprint arXiv:2507.02092*, 2025. URL https://arxiv.org/abs/2507.02092.

Alex Graves. Adaptive computation time for recurrent neural networks, 2017. URL https://arxiv.org/abs/1603.08983.

Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces. *arXiv preprint arXiv:2312.00752*, 2023.

Shibo Hao, Sainbayar Sukhbaatar, DiJia Su, Xian Li, Zhiting Hu, Jason Weston, and Yuandong Tian. Training large language models to reason in a continuous latent space. *arXiv preprint arXiv:2412.06769*, 2024. URL https://arxiv.org/abs/2412.06769.

John Hewitt, Michael Hahn, Surya Ganguli, Percy Liang, and Christopher D Manning. Rnns can generate bounded hierarchical languages with optimal memory. *arXiv preprint arXiv:2010.07515*, 2020.

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

Julia M. Huntenburg, Pierre-Louis Bazin, and Daniel S. Margulies. Large-scale gradients in human cortical organization. *Trends in Cognitive Sciences*, 22(1):21–31, 2018. doi: 10.1016/j.tics.2017.11.002.

Alexia Jolicoeur-Martineau. Less is more: Recursive reasoning with tiny networks. *arXiv preprint*, arXiv:2510.04871, 2025. doi: 10.48550/arXiv.2510.04871. URL https://arxiv.org/abs/2510.04871.

Jan Koutník, Klaus Greff, Faustino Gomez, and Jürgen Schmidhuber. A clockwork RNN. In *Proceedings of the 31st International Conference on Machine Learning (ICML)*, 2014.

Akarsh Kumar and Phillip Isola. Pretraining recurrent networks without recurrence, 2026. URL https://arxiv.org/abs/2606.06479.

Bingbin Liu, Jordan T. Ash, Surbhi Goel, Akshay Krishnamurthy, and Cyril Zhang. Transformers learn shortcuts to automata. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=De4FYqjFueZ.

Sean McLeish, Ang Li, John Kirchenbauer, Dayal Singh Kalra, Brian R. Bartoldson, Bhavya Kailkhura, Avi Schwarzschild, Jonas Geiping, Tom Goldstein, and Micah Goldblum. Teaching pretrained language models to think deeper with retrofitted recurrence. *arXiv preprint arXiv:2511.07384*, 2025.

William Merrill and Ashish Sabharwal. The parallelism tradeoff: Limitations of log-precision transformers. In *Transactions of the Association for Computational Linguistics*, volume 11, pp. 531–545, 2023. doi: 10.1162/tacl_a_00562.

William Merrill, Jackson Petty, and Ashish Sabharwal. The illusion of state in state-space models. In *International Conference on Machine Learning*, 2024. arXiv:2404.08819.

Mayank Mishra, Shawn Tan, Ion Stoica, Joseph Gonzalez, and Tri Dao. M^2RNN: Non-linear rnns with matrix-valued states for scalable language modeling, 2026. URL https://arxiv.org/abs/2603.14360.

Takeru Miyato, Bernhard Jaeger, Max Welling, and Andreas Geiger. Gta: A geometry-aware attention mechanism for multi-view transformers. *arXiv preprint arXiv:2310.10375*, 2023.

Takeru Miyato, Sindy Löwe, Andreas Geiger, and Max Welling. Artificial kuramoto oscillatory neurons. In *International Conference on Learning Representations (ICLR)*, 2025. URL https://openreview.net/forum?id=nwDRD4AMoN.

Michael C Mozer, Shoaib Ahmed Siddiqui, and Rosanne Liu. The topological trouble with transformers. *arXiv preprint arXiv:2604.17121*, 2026.

Nicola Muca Cirone, Antonio Orvieto, Benjamin Walker, Cristopher Salvi, and Terry Lyons. Theoretical foundations of deep selective state-space models. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (eds.), *Advances in Neural Information Processing Systems*, volume 37, pp. 127226–127272. Curran Associates, Inc., 2024. doi: 10.52202/079017-4041. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/e6231c5f46598cfd09ff1970524e0436-Paper-Conference.pdf.

Asier Mujika, Florian Meier, and Angelika Steger. Fast-slow recurrent neural networks. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL https://proceedings.neurips.cc/paper/2017/hash/e4a93f0332b2519177ed55741ea4e5e7-Abstract.html.

John D. Murray, Alberto Bernacchia, David J. Freedman, Ranulfo Romo, Jonathan D. Wallis, Xinying Cai, Camillo Padoa-Schioppa, Tatiana Pasternak, Hyojung Seo, Daeyeol Lee, and Xiao-Jing Wang. A hierarchy of intrinsic timescales across primate cortex. *Nature Neuroscience*, 17(12):1661–1663, 2014. doi: 10.1038/nn.3862.

Costin-Andrei Oncescu, Depen Morwani, Samy Jelassi, Alexandru Meterez, Mujin Kwun, and Sham Kakade. The recurrent transformer: Greater effective depth and efficient decoding, 2026. URL https://arxiv.org/abs/2604.21215.

Razvan Pascanu, Çaglar Gülçehre, Kyunghyun Cho, and Yoshua Bengio. How to construct deep recurrent neural networks. In *International Conference on Learning Representations (ICLR)*, 2014. URL https://arxiv.org/abs/1312.6026.

Bowen Peng, Jeffrey Quesnelle, Honglu Fan, and Enrico Shippole. YaRN: Efficient context window extension of large language models. In *The Twelfth International Conference on Learning Representations (ICLR)*, 2024. URL https://openreview.net/forum?id=wHBfxhZu1u.

Ofir Press, Noah A. Smith, and Mike Lewis. Train short, test long: Attention with linear biases enables input length extrapolation. In *International Conference on Learning Representations (ICLR)*, 2022. URL https://arxiv.org/abs/2108.12409.

Jack W. Rae, Anna Potapenko, Siddhant M. Jayakumar, and Timothy P. Lillicrap. Compressive transformers for long-range sequence modelling, 2019. URL https://arxiv.org/abs/1911.05507.

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. arXiv:2502.17416.

Jürgen Schmidhuber. Learning complex, extended sequences using the principle of history compression. *Neural Computation*, 4(2):234–242, 1992. doi: 10.1162/neco.1992.4.2.234.

Mark Schöne, Babak Rahmani, Heiner Kremer, Fabian Falck, Hitesh Ballani, and Jannes Gladrow. Implicit language models are RNNs: Balancing parallelization and expressivity. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*. PMLR, 2025.

M. P. Schützenberger. On context-free languages and push-down automata. *Information and Control*, 6(3):246–264, 1963. ISSN 0019-9958. doi: 10.1016/S0019-9958(63)90306-1. URL https://www.sciencedirect.com/science/article/pii/S0019995863903061.

Avi Schwarzschild, Eitan Borgnia, Arjun Gupta, Furong Huang, Uzi Vishkin, Micah Goldblum, and Tom Goldstein. Can you learn an algorithm? generalizing from easy to hard problems with recurrent networks. In M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan (eds.), *Advances in Neural Information Processing Systems*, volume 34, pp. 6695–6706. Curran Associates, Inc., 2021. URL https://proceedings.neurips.cc/paper_files/paper/2021/file/3501672ebc68a5524629080e3ef60aef-Paper.pdf.

Julien Siems, Timur Carstensen, Arber Zela, Frank Hutter, Massimiliano Pontil, and Riccardo Grazzi. DeltaProduct: Improving state-tracking in linear RNNs via householder products. In *Advances in Neural Information Processing Systems*, 2025. arXiv:2502.10297.

Jimmy T. H. Smith, Andrew Warrington, and Scott W. Linderman. Simplified state space layers for sequence modeling. *arXiv preprint arXiv:2208.04933*, 2022.

Jianlin Su, Yu Lu, Shengfeng Pan, Ahmed Murtadha, Bo Wen, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding, 2021.

Ilya Tolstikhin, Neil Houlsby, Alexander Kolesnikov, Lucas Beyer, Xiaohua Zhai, Thomas Unterthiner, Jessica Yung, Andreas Steiner, Daniel Keysers, Jakob Uszkoreit, Mario Lucic, and Alexey Dosovitskiy. MLP-Mixer: An all-MLP architecture for vision. In *Advances in Neural Information Processing Systems*, volume 34. Curran Associates, Inc., 2021. URL https://proceedings.neurips.cc/paper/2021/hash/cba0a4ee5ccd02fda0fe3f9a3e7b89fe-Abstract.html.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Guan Wang, Jin Li, Yuhao Sun, Xing Chen, Changling Liu, Yue Wu, Meng Lu, Sen Song, and Yasin Abbasi Yadkori. Hierarchical reasoning model. *arXiv preprint arXiv:2506.21734*, 2025. URL https://arxiv.org/abs/2506.21734.

Suyuchen Wang, Ivan Kobyzev, Peng Lu, Mehdi Rezagholizadeh, and Bang Liu. Resonance RoPE: Improving context length generalization of large language models. In *Findings of the Association for Computational Linguistics: ACL 2024*, pp. 586–598, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.findings-acl.32. URL https://aclanthology.org/2024.findings-acl.32/.

Ziyu Wang, Wenhao Jiang, Yiming Zhu, Li Yuan, Yibing Song, and Wei Liu. DynaMixer: A vision MLP architecture with dynamic mixing. In *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pp. 22691–22701. PMLR, 2022.

Guoqiang Wei, Zhizheng Zhang, Cuiling Lan, Yan Lu, and Zhibo Chen. Active token mixer. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pp. 2759–2767, 2023. doi: 10.1609/aaai.v37i3.25376.

Kevin Xu and Issei Sato. On expressive power of looped transformers: Theoretical analysis and enhancement via timestep encoding. In *International Conference on Machine Learning*, 2025. arXiv:2410.01405.

Greg Yang, Dingli Yu, Chen Zhu, and Soufiane Hayou. Tensor programs vi: Feature learning in infinite-depth neural networks. *arXiv preprint arXiv:2310.02244*, 2023.

Liu Yang, Kangwook Lee, Robert Nowak, and Dimitris Papailiopoulos. Looped transformers are better at learning learning algorithms. In *International Conference on Learning Representations*, 2024. arXiv:2311.12424.

Shunyu Yao, Binghui Peng, Christos Papadimitriou, and Karthik Narasimhan. Self-attention networks can process bounded hierarchical languages. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (ACL-IJCNLP)*, pp. 3770–3785, 2021. doi: 10.18653/v1/2021.acl-long.292. URL https://aclanthology.org/2021.acl-long.292/.

Roxana Zeraati, Yan-Liang Shi, Nicholas A. Steinmetz, Marc A. Gieselmann, Alexander Thiele, Tirin Moore, Anna Levina, and Tatiana A. Engel. Intrinsic timescales in the visual cortex change with selective attention and reflect spatial connectivity. *Nature Communications*, 14(1):1858, 2023. doi: 10.1038/s41467-023-37613-7.

Yongchao Zhou, Uri Alon, Xinyun Chen, Xuezhi Wang, Rishabh Agarwal, and Denny Zhou. Transformers can achieve length generalization but not robustly. *arXiv preprint arXiv:2402.09371*, 2024. doi: 10.48550/arXiv.2402.09371. URL https://arxiv.org/abs/2402.09371.

Rui-Jie Zhu, Zixuan Wang, Kai Hua, Tianyu Zhang, Ziniu Li, Haoran Que, Boyi Wei, Zixin Wen, Fan Yin, He Xing, et al. Scaling latent reasoning via looped language models. *arXiv preprint arXiv:2510.25741*, 2025. URL https://arxiv.org/abs/2510.25741.

A Model Details

Terminology. We retain the term "fast–slow" in line with the multiscale RNN tradition (Schmidhuber, 1992; Koutník et al., 2014). The unrelated FS-RNN of Mujika et al. (2017) uses the same name for a wiring of distinct LSTM cells within one observation step, without an inner clock or weight sharing.

A.1 Implementation of R

A.1.1 AKOrN

For the choice of R in our experiments, a mechanism that performed particularly stably was a variation of the updates used by Miyato et al. (2025), although R implemented with a vanilla Transformer block performed competitively as well. See Figure 8 for the design of R_{AKOrN} we used in our experiments. This module has two additional components compared with R_{TF}, and we would like to refer readers to the original paper for more details. First, $\{\Omega_i; i \in 1:n\}$ is an intrinsic frequency term represented by anti-symmetric matrices, intended to provide each token in $S^{d-1} \subset \mathbb{R}^d$ with rotational momentum. Second, there is a linear $\mathbf{Proj}_{\mathbf{Z}}$ operator that projects a given vector to the tangent space of S^{d-1} at $\mathbf{Z}$.

```
class R():
    def __init__(dim, oscillator_dim, init_gamma):
        self.attn = SelfAttention(dim)
        self.mlp = MLP(dim)
        self.omega = Omega(dim, oscillator_dim)
        self.gamma = nn.Parameter(init_gamma, require_grad=True)

    def forward(state, cond):
        y = self.attn(state + cond)
        y = self.mlp(state + cond + y)
        y = proj(y, state) # projection onto the tangent space
        y = y + self.omega(state)
        state += self.gamma * y
        state /= norm(state)
        return state
```

Figure 8: Pseudocode of our R_{AKOrN}.

We note that the original implementation of Miyato et al. (2025) does not introduce an MLP, and that it introduces $\mathbf{C}$ additively outside $\mathbf{Attn}$, which consists of the usual procedure of

$$Attn(\mathbf{Z}) = \mathbf{softmax}\left(\frac{\mathbf{Z}W_Q W_K^{\mathrm{T}}\mathbf{Z}^{\mathrm{T}}}{\sqrt{d'}}\right)\mathbf{Z}W_V \tag{5}$$

$$:= \mathbf{AttnMat}(\mathbf{Z})\mathbf{Z}W_V \tag{6}$$

with $W_Q, W_K, W_V \in \mathbb{R}^{d\times d'}$, $\mathbf{Z} \in \mathbb{R}^{n\times d}$. In implementation, we used the multihead version of this. We used GTA (Miyato et al., 2023) as positional embeddings. Pseudocode is provided in Figure 8.

Energy-like Scalar In the absence of the MLP term, in a form closer to the attentive connectivity of Miyato et al. (2025), AKOrN admits an update of the form

$$\mathbf{Z}_i = \Omega_i \mathbf{Z}_i + \mathbf{Proj}_{\mathbf{Z}_i}\left(\sum_j [\mathbf{AttnMat}(\mathbf{Z})]_{ij} W_V^{\mathrm{T}}\mathbf{Z}_j + \mathbf{C}_i\right) \tag{7}$$

where $\mathbf{AttnMat}(\mathbf{Z}) := J$ is an attention matrix from $\mathbf{Z} \in \mathbb{R}^{n\times d}$, and we introduce a pseudo-energy of the following form:

$$\mathcal{E}(\mathbf{Z}|\mathbf{C}) = -\frac{1}{2}\sum_{i,j} J_{ij}\mathbf{Z}_i^{\mathrm{T}}\mathbf{Z}_j - \sum_i \mathbf{C}_i^{\mathrm{T}}\mathbf{Z}_i \tag{8}$$

Table 4: Default hyperparameters for Maze (supervised), Dyck (supervised), and Minigrid (RL/PPO). We use these settings for all baselines unless otherwise specified.

Parameter	**Maze**	**Dyck**	**Minigrid**
Optimizer	AdamW	AdamW	Adam
Batch size	256	256	3200
Training epochs	300	30	—
Weight decay	0.1	0.01	—
Gradient clip	0.1	1.0	0.5
Scheduler	cosine	cosine	—
Dataset size (train/val)	45,000 / 5,000	10,000 / 1,000	—
Bracket types (k)	—	30	—
Training depth (m)	—	5	—
Sequence length (train)	—	10–40 tokens	—
Total timesteps	—	—	1.0×10^6
# Environments	—	—	32
# Steps / env	—	—	96
Update epochs	—	—	4
Discount factor (γ)	—	—	0.995
GAE λ	—	—	0.95
Value loss coef. (c_v)	—	—	0.5
Clip coefficient (PPO ratio)	—	—	0.1

in close relation to the previously introduced energy of Geshkovski et al. (2025). Because we incorporate $\mathbf{C}$ additively before the attention as in Figure 8, however, we use the same energy as $\mathcal{E}(\mathbf{Z}, \mathbf{C})$ in (8) to assist the visualization of the latent for our construction of R_{AKOrN}, with A and B being the gating functions that depend on $\mathbf{Z}$ and $\mathbf{C}$.

A.1.2 MAMBA2 AND LSTM CORE

We implemented the Mamba2 and LSTM cores within the FSRM loop so that the conditions in Section 3.2 are better satisfied. In designing these cores, we used an additional inner loop in which these mechanisms have their own latents. Namely, in the presence of input $\mathbf{C}_s \in \mathbb{R}^{\ell \times d}$, the latent $\mathbf{Z}_s(t) \in \mathbb{R}^{L \times d}$ for these cores evolves as

$$\mathbf{Z}_s(t+1) = R\left(\mathbf{Z}_s(t),\, \mathbf{C}_s\right), \qquad \mathbf{Z}_{s+1}(0) = \mathbf{Z}_s(T), \qquad t = 0, \ldots, T, \tag{9}$$

where

$$R\left(\mathbf{Z}_s(t),\, \mathbf{C}_s\right) = M[h_0, \ldots h_L] \tag{10}$$

with mixing $M \in R^{L \times d}$ in *token* direction, and h evolving on its own inner-inner loop, which, for example, for Mamba, takes the form of

$$h_{\ell+1} = A([\mathbf{Z}_s(t) + \mathbf{C}_s])h_\ell + B([\mathbf{Z}_s(t) + \mathbf{C}_s])[\mathbf{Z}_s(t) + \mathbf{C}_s], \qquad \ell = 0, \ldots L-1 \tag{11}$$

with h_0 being initialized to a constant at every observation step s. In this design of R itself, the L-size loop features both latent-dependent transition and dimensional mixing. Our LSTM core is also defined analogously with its own *cell* latent and *hidden state* latent. The Mamba/LSTM + FFN variants add a separate MLP along the d dimension on top of the core described here, providing additional dimensional/channel mixing beyond the native gates and projections.

Table 5: Model-specific hyperparameters for Maze/Dyck/Minigrid. Parameter counts and key architectural/training choices for each baseline and our method. When three numbers are shown, they correspond to Maze/Dyck/Minigrid respectively.

	LSTM	**Mamba**	**Transformer**	**FSRM**
Parameters (M)	2.86/37.1/2.47	1.25/16.5/2.81	1.73/21.3/3.30	1.16/1.41/1.19
Hidden dim	128/512/512	128/256/380	128/256/384	64/256/512
Entropy coef.	—/—/1e-4	—/—/1e-2	—/—/1e-2	—/—/1e-2
RNN hidden dim	256/512/256	—	—	—
Transformer heads	—	—	8/8/4	4
Memory len.	—	—	—/—/119	—
d_{state}	—	64	—	—
d_{conv}	—	4	—	—
Expand ratio	—	2	—	—
Depth	4/2/1	6/4/1	6/4/3	1/2/1
Oscillator dim	—	—	—	4/4/2
Internal Steps	—	—	—	5/5/10
Initial γ	—	—	—	0.1
Initial Ω	—	—	—	0.1
Chunk size	—	—	—	16/—/—
Learning rate	1e-3/5e-3/2.5e-4	1e-3/5e-4/1.5e-4	1e-3/5e-3/2.5e-4	1e-3/5e-3/2.5e-4

B Experiment Details

B.1 Baselines and Training Protocol

We use a task-matched baseline set because the three benchmarks differ in supervision and interaction. For Dyck, the controlled supervised comparisons are LSTM (Hochreiter & Schmidhuber, 1997), Mamba2 (Gu & Dao, 2023; Dao & Gu, 2024), and Transformer (Vaswani et al., 2017), trained on the same Dyck-$(30, 5)$ split as FSRM. The LLM comparisons in Figure 4 are context-setting stress tests: Gemini 3.0 Pro, Claude Opus 4.5, and GPT-5.1 Thinking are prompted with the stack algorithm, while Qwen3-4B-Base is LoRA fine-tuned on the same short split. For Local-Maze, we compare LSTM, Mamba2, TF, Looped TF (Fan et al., 2025), DeltaProduct (Siems et al., 2025), TRM (Jolicoeur-Martineau, 2025), and FSRM, with S5 and CTM included in the extended appendix comparison. For Minigrid, all methods use PPO and the same ID/OOD environment splits; the sequence-model baselines are LSTM, Mamba2, and TransformerXL (Dai et al., 2019). For the TRM baseline in Figure 5b, we use six inner and three outer recursion steps and adapt it to streaming inputs in the same manner as Looped TF. Tables 4 and 5 give the shared training settings and model-specific hyperparameters.

B.2 Hyperparameters

We report the main hyperparameters of the models in Tables 4 and 5. For FSRM, the model-specific oscillator dimension and number of internal steps are task-dependent and are reported in Table 5 in Maze/Dyck/Minigrid order. In particular, the Minigrid runs use oscillator dimension 2 and $T = 10$, while the supervised Maze and Dyck runs use $T = 5$. For Dyck, we grid-search the learning rate over $\{10^{-4}, 5 \times 10^{-4}, 10^{-3}, 5 \times 10^{-3}\}$ per model and select by ID accuracy at lengths 1280–2560. For the Dyck core ablation in Appendix C.1.4, we use a grid of $\{1, 2, 3, 4, 5\} \times 10^{-3}$ and report the OOD accuracy of the best-ID checkpoint. For Local-Maze and Minigrid, we likewise grid-search $\{10^{-4}, 3 \times 10^{-4}, 5 \times 10^{-4}, 10^{-3}\}$ and select by in-distribution validation accuracy.

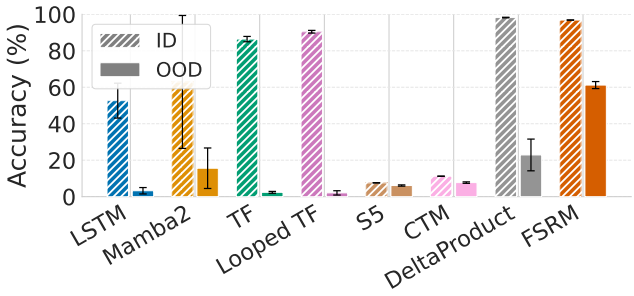


Figure 9: Extended Local-Maze ID and OOD accuracy comparison including S5 and CTM baselines.

B.3 Licenses for Existing Assets

We list the main existing assets used in the experiments and their licenses or terms of use below. The Dyck-$(30, 5)$ and Local-Maze datasets used in this paper are synthetically generated by our experimental pipeline from the task definitions described in Appendix B.6 and Appendix B.5; we do not redistribute third-party dataset files for these two settings. For commercial LLM baselines, we only access the models through their public APIs for evaluation and do not redistribute model weights or generated datasets.

- **Minigrid environments** (Chevalier-Boisvert et al., 2023): DoorKey, MultiRoom, and LavaCrossing environments; Apache License 2.0 (Farama-Foundation/Minigrid).
- **Mamba/Mamba2 implementation** (Gu & Dao, 2023; Dao & Gu, 2024): sequence-model baselines and Mamba2 core variants; Apache License 2.0 (state-spaces/mamba).
- **Qwen3-4B-Base**: open-weight LLM baseline fine-tuned on Dyck-$(30, 5)$; Apache License 2.0 (Qwen/Qwen3-4B-Base).
- **Gemini, Claude, and GPT API models**: commercial API services governed by the relevant provider terms of use; no model weights are redistributed.

B.4 Compute Resources

The Dyck, Local-Maze, and Minigrid experiments each used one NVIDIA H100 GPU. The inference-timing measurement in Appendix C.3.3 used one NVIDIA GH200 GPU.

B.5 Local-Maze

Local-Maze extends the navigation task of Darlow et al. (2025). A navigator explores the maze under a fixed right-hand-rule policy; the model itself does not control the navigator. This decouples the input stream from the model under test and ensures that the same maze yields the same observation sequence across runs. At stream step s, the model receives an egocentric 7×7 observation O_s and, after the episode, predicts the shortest path from start to goal as a sequence of actions (`up`, `down`, `left`, `right`, `pause`). The stream length S varies with maze geometry, and the navigator's position cannot be directly inferred from the egocentric view. The task therefore requires integrating task-relevant observations, discarding information from dead-end branches, and iteratively consolidating a hypothesis about the shortest path as new observations arrive. Supervision is the true shortest path, given in the same action format; accuracy is measured by exact match after removing `pause` tokens from both prediction and ground truth. Figure 10 reports the matched-compute Local-Maze comparison, which controls for differences in per-update training cost. To match compute, we changed only the channel width (ch) of each model and selected widths with similar wall-clock time per training batch. We measured this cost as the average forward-plus-backward time per batch over 50 batches after 10 warmup batches.

Because the maze task is particularly related to *memorizing* task-relevant information, we additionally experimented with a memory-inspired design of R. In particular, we wrapped the original attention-based R in Appendix A.1.1 with a GRU. Letting $\mathbf{W}_t$ denote the hidden state of the GRU, we replaced

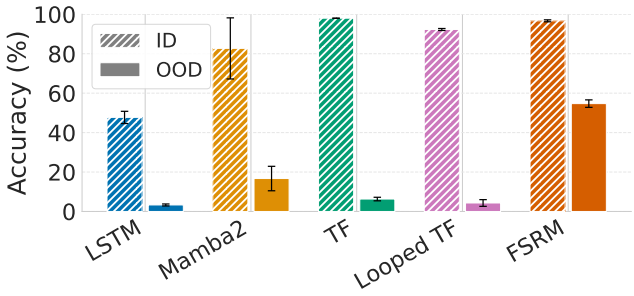


Figure 10: Local-Maze ID and OOD accuracy under matched compute.

the original R with

$$R_{\mathrm{GRU}}(\mathbf{Z}, \mathbf{C}) = \mathrm{GRU}(R(\mathbf{Z}, \mathbf{C}); \mathbf{W})_i. \tag{12}$$

where $\mathbf{W}$ is updated at every fast step by the GRU's internal recurrence (omitted for brevity). This module also acts as an internal filter of the interaction features, encouraging continuity of the effective coupling even when $\mathbf{C}_t$ varies rapidly.

B.6 Dyck Language

We use Dyck-(k, m) (Hewitt et al., 2020), a formal language with k bracket types and maximum nesting depth m, to test whether the model can learn the underlying stack rule from short sequences and apply it to streams whose horizon is not known in advance. We set $k = 30$ and $m = 5$. For example, on input "({ [", the prediction after the fourth token is "}" because "{" is the most recent unclosed bracket. The task requires tracking all unclosed brackets, whose number at step s corresponds to the current stack depth and may grow with the sequence. Once the structural rule is captured, the same rule applies as sequence length increases.

Hewitt et al. (2020) originally formulated this task as standard next-token prediction. However, because there are multiple valid choices for the type of the next open bracket, pure next-token prediction is not deterministic. To eliminate this source of indeterminacy in accuracy evaluation, we score only closing brackets and $\star$. Predictions are made sequentially at every stream position s. Given the prefix observed through s, the target is the closing bracket for the most recent unclosed bracket; when the stack is empty, the target is $\star$.

For the training dataset, we randomly generate Dyck-$(30, 5)$ strings and train on $10{,}000$ strings of length at most 40. For evaluation, we separate length extrapolation from the Dyck OOD distribution. The ID curve uses randomly generated Dyck-$(30, 5)$ strings at lengths up to $2{,}560$, so it also tests lengths far beyond the training range. The OOD curve uses an *n-regular run*: a sequence that (i) begins with a random sequence of open brackets of random length and (ii) is followed by an infinite repetition of depth-n openings and depth-n closings. For example, "({ [([[ ]] (( )) { { } } $\cdots$" is a 2-regular run, and Figure 2 shows a 1-regular run. In the accuracy results, Dyck OOD refers to the 1-regular evaluation distribution, not merely to the fact that the sequence length exceeds the training maximum. The model sequentially predicts on $1{,}000$ strings for each evaluation distribution, and accuracy is computed token-wise. The PCA latent-traversal visualizations are separate qualitative probes and use 5-regular runs.

B.6.1 Frontier LLM Setting

For the comparison with production LLMs on Dyck-$(30, 5)$, we queried each model via its public API using a common prompt that describes the stack-based algorithm (Table 16). Following Chae et al. (2024), we designed the prompt so that it clearly separates (i) a natural-language specification of the ground-truth stack algorithm from (ii) explicit execution instructions that tell the model to simulate this algorithm step-by-step and output only the prediction string. No additional fine-tuning was performed; all models were evaluated in a pure inference setting. We set the temperature to 0 if the API accepts it, so that the model's behavior is more deterministic.

Modern LLMs can adapt their test-time compute by varying the number of their internal "reasoning tokens," but commercial APIs strictly enforce a maximum on the total number of output tokens (reasoning plus final answer). As of November 2025, the documented upper limits for our models are 32k tokens for Claude Opus 4.5 and 64k tokens for both Gemini 3.0 Pro and GPT-5.1. For each provider, we first empirically determined the longest Dyck input length for which the prompt, the model's reasoning tokens, and the final prediction string all remain within this limit. We then generated 100 Dyck sequences at this length and measured token-level prediction accuracy. Use of external tools such as Python interpreters or code execution APIs was explicitly disabled so that each model had to execute the algorithm internally. For reference, Table 17 shows an example of the internal reasoning tokens produced by Claude Opus 4.5 on a representative input.

B.6.2 Fine-tuned Qwen3-4B-Base

The frontier LLM evaluation is useful as a test of whether current production models can execute the Dyck algorithm from a prompt, but it is not fully symmetric with our setting because those models are not trained on the Dyck data used by FSRM. We therefore conducted an additional comparison with Qwen3-4B-Base, an open-weight LLM that we fine-tuned directly on the same Dyck-$(30, 5)$ data and evaluation protocol used in Figure 4.

We fine-tuned Qwen3-4B-Base with LoRA on the same Dyck-$(30, 5)$ split used in Figure 4: training strings of length at most 40, with $10{,}000$ training samples and $1{,}000$ validation samples, and report results over 3 random seeds. We used a shared training setup across baselines with batch size 128, learning rate 10^{-4}, 30 epochs, and bf16 precision. Because we observed overfitting later in training, we selected the best checkpoint by early stopping on the validation set. For LoRA, we used rank 32, $\alpha = 16$, dropout 0.1, and adapted the standard projection and feedforward modules, following the official Qwen fine-tuning recipe: `https://qwen.readthedocs.io/en/v1.5/training/SFT/example.html#python-script`.

We evaluated the fine-tuned Qwen model using the same OOD protocol as Figure 4, namely 1-regular runs. We first evaluated a long-horizon OOD run with context length $20{,}480$; we were unable to evaluate beyond $20{,}480$ because the model ran out of memory even with batch size 1. The result shows that Qwen3-4B-Base achieves high accuracy at in-distribution lengths but fails to generalize to the OOD length, with accuracy dropping to near 50% at length $20{,}480$.

This train-short-test-long failure is consistent with prior reports that pretrained language models fine-tuned on short algorithmic instances can achieve near-perfect in-distribution accuracy while degrading rapidly out of distribution with length (Anil et al., 2022); related work also finds that OOD behavior can be nonmonotone due to shortcut solutions, positional effects, or output-distribution drift (Liu et al., 2023; Wang et al., 2024; Zhou et al., 2024; Du et al., 2025).

B.6.3 FSRM Hierarchical Architecture

For this task, we used a two-layer extension of FSRM. Since the task has a hierarchical stack structure, this architecture instantiates a hierarchical version of FSRM with two fast processes. Let $\mathbf{Z}_s^{(1)}(t)$ and $\mathbf{Z}_s^{(2)}(t)$ denote the first- and second-layer latents at stream index s and inner-loop index t, with each layer carried forward as $\mathbf{Z}_{s+1}^{(\ell)}(0) = \mathbf{Z}_s^{(\ell)}(T)$. The first layer receives the encoded observation $\mathbf{C}_s$ and evolves as

$$\mathbf{Z}_s^{(1)}(t+1) = R_1\Big(\mathbf{Z}_s^{(1)}(t), \mathbf{C}_s\Big), \qquad t = 0, \dots, T-1.$$

Writing $r_s^{(1)} = \mathrm{Readout}_1(\mathbf{Z}_s^{(1)}(T))$ and $\mathcal{H}_s = [r_{s-H+1}^{(1)}, \dots, r_s^{(1)}]$ for the length-H history queue, the second layer evolves as

$$\mathbf{Z}_s^{(2)}(t+1) = R_2\Big(\mathbf{Z}_s^{(2)}(t), \mathcal{H}_s\Big), \qquad t = 0, \dots, T-1.$$

Predictions are obtained by decoding $\mathbf{Z}_s^{(2)}(T)$ at each stream index s; Figure 11 gives the implementation-level pseudocode. We set the history size to $H = 4$.

```
class FastSlow():
    def __init__(dim, oscillator_dim, init_gamma):
        self.embed = nn.Embedding()
        self.R1 = R(dim, oscillator_dim, init_gamma)
        self.R2 = R(dim, oscillator_dim, init_gamma)

    def forward(tokens, H, T):
        # C: channels, K: num oscillators per token, H: history size
        c = self.embed(tokens) # (L, K, C)
        x = randn(K, C)
        z = randn(H, K, C)
        h = queue(H) # length-H history queue
        z_out = zeros(H, K, C)

        # Slow loop
        for s in range(len(embeddings)):
            # First fast-layer
            for t in range(T):
                x = self.R1(x, c[s])
            x_out = x_readout(x)

            # Second fast-layer
            h.enqueue(x_out)
            for t in range(T):
                z = self.R2(z+z_out, h)
            z_out = z_readout(z)

            logits[s] = classifier(z_out)
        return logits
```

Figure 11: Pseudocode for the two-stage architecture used in Section 4.

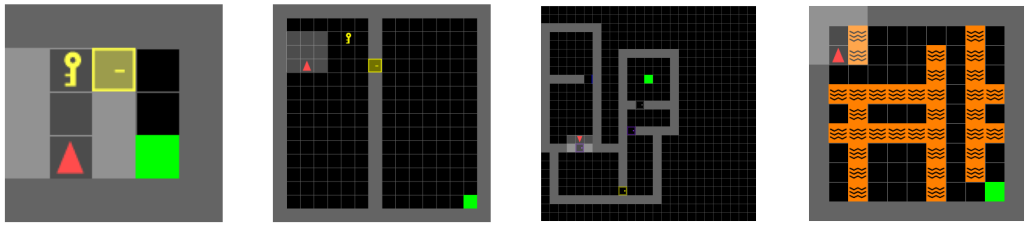
B.7 Reinforcement Learning

We evaluate partially observable reinforcement-learning tasks from Minigrid (Chevalier-Boisvert et al., 2023). At each step, the agent receives an egocentric 7×7 observation with object, color, and door-state channels, and selects an action $a_s \in \{\texttt{left}, \texttt{right}, \texttt{forward}, \texttt{toggle}, \texttt{pickup}, \texttt{drop}, \texttt{done}\}$. Reward is given only at the end of a successful episode. Unlike Dyck and Local-Maze, no ground-truth action sequence is provided, so the model must identify which observations are worth consolidating into its evolving plan. Agents must navigate an environment to reach the goal while:

- **DoorKey**: finding a key and unlocking a door
- **LavaCrossing**: avoiding the impassable lava river
- **MultiRoom**: going through multiple rooms with doors.

Figure 12 provides illustrated examples of the environments.

To train our model, we followed the strategy of Geiping et al. (2025); Wang et al. (2025) for computational efficiency and used gradient truncation. In particular, we set $T = 10$, and backpropagated through only the last 5 iterations of the recurrent loop at each s.



(a) DoorKey (ID) (b) DoorKey (OOD) (c) MultiRoom (OOD) (d) LavaCrossing (OOD)

Figure 12: Examples of the partially observable Minigrid reinforcement-learning environments used in our experiments. DoorKey illustrates the ID-to-OOD size shift; MultiRoom and LavaCrossing show additional OOD tasks that require long-horizon memory and planning under sparse rewards.

Table 6: ID/OOD splits for the Minigrid environments. We train agents on the simplest configuration and test for zero-shot generalization on OOD environments.

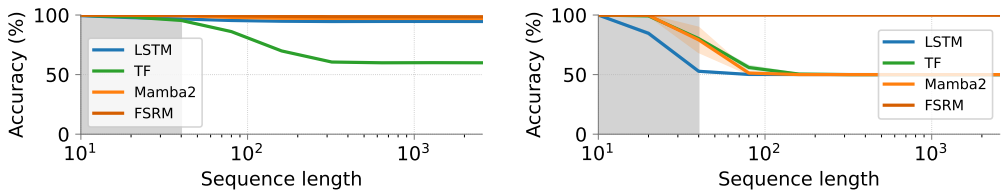
Environment	Label	ID	OOD
MiniGrid-MultiRoom-N2-S4-v0	N2-S4	✓	✗
MiniGrid-MultiRoom-N4-S5-v0	N4-S5	✗	✓
MiniGrid-MultiRoom-N6-v0	N6	✗	✓
MiniGrid-DoorKey-5x5-v0	5x5	✓	✗
MiniGrid-DoorKey-6x6-v0	6x6	✗	✓
MiniGrid-DoorKey-8x8-v0	8x8	✗	✓
MiniGrid-DoorKey-16x16-v0	16x16	✗	✓
MiniGrid-LavaCrossingS9N1-v0	S9N1	✓	✗
MiniGrid-LavaCrossingS9N2-v0	S9N2	✗	✓
MiniGrid-LavaCrossingS9N3-v0	S9N3	✗	✓
MiniGrid-LavaCrossingS11N5-v0	S11N5	✗	✓

C Additional Results

C.1 Dyck

C.1.1 ID and OOD Accuracy

Figure 13 reports the ID curve together with the OOD curve shown in Figure 5a. Both curves extend to length 2,560, beyond the training maximum of 40.



(a) ID accuracy (b) OOD accuracy

Figure 13: Dyck accuracy on ID and OOD runs. Both panels evaluate lengths beyond the training range. The OOD panel shows the curves of Figure 5 and is included here for easy comparison with the ID panel.

C.1.2 Energy Traces

Through learning the prediction task, FSRM discovers latent dynamics that align with the structural organization of the Dyck language. Recall that the AKOrN-based recurrence used in our experiments

admits an energy-like scalar that reflects the organizational level of the latent state (Appendix A.1.1). We show the evolution of this energy in Figure 14, and PCA projections of the corresponding latents in Figure 7.

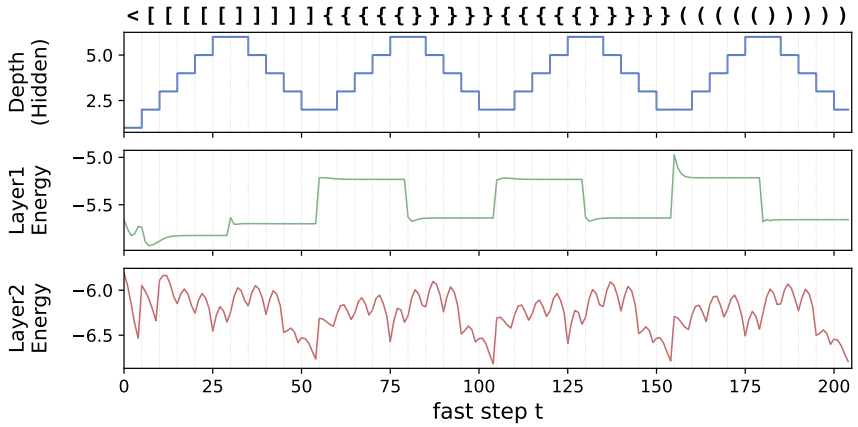


Figure 14: Energy traces (see (8)) for the first and second layers of the hierarchical version of FSRM during a 5-regular Dyck run. The dashed lines indicate token arrival times.

The energy of $\mathbf{Z}^{(1)}$ spikes upon the arrival of a new token and then stabilizes at a level that depends on the bracket type and whether the current token is opening or closing. In contrast, the energy of $\mathbf{Z}^{(2)}$ is largely insensitive to bracket identity and is instead organized by stack *depth* (see Appendix B.6.3 for the definition of $\mathbf{Z}^{(1)}$ and $\mathbf{Z}^{(2)}$). This qualitative shift in energy organization indicates that the second fast process abstracts away from token identity and operates at a higher structural level. These emergent structures are also consistent with the results in Appendix C.1.3 and are unique to the models that generalize on OOD (Figures 18 and 19).

C.1.3 Linear Probes

To quantify whether the recurrent state contains the variables required by the Dyck rule, we train linear probes on frozen hidden states from the in-distribution (length 10–40) training split. The probe predicts the bracket type of each currently unresolved opener (open bracket) in the active stack, with one head per stack position from the top; *opener age* denotes the number of steps elapsed since that opener was read. Missing stack slots and age-zero events are masked. We then evaluate the probes on length-OOD Dyck-$(30, 5)$ sequences (30 bracket types, maximum stack depth 5). The successful FSRM model preserves linearly decodable information about all five unresolved stack positions, whereas sequence baselines largely retain only the top stack item or lose stack information as opener age increases. Since only the top stack item corresponds to the immediate prediction target, decodability of deeper stack entries provides evidence that the latent state retains information needed for prediction targets that arise when inner brackets are eventually closed.

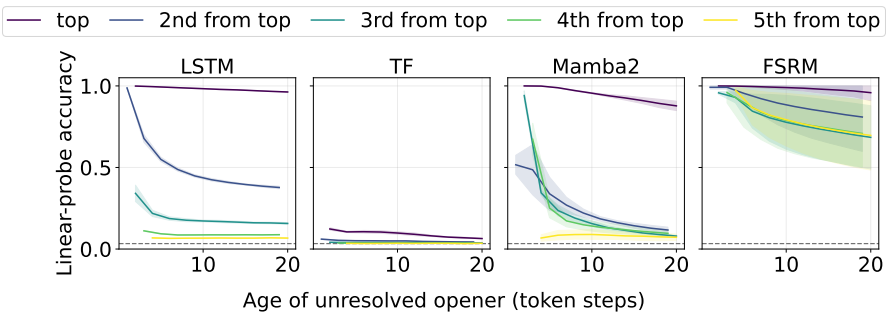


Figure 15: Linear probe of unresolved stack contents on Dyck-$(30, 5)$. Models are trained on length 10–40 sequences and frozen. A multi-head linear probe is trained on the ID split to predict the bracket identity of the i-th unresolved opener from the top of the active stack. Probes are evaluated on length-2,560 OOD sequences. The x-axis shows opener age, i.e. the number of steps elapsed since that opener was read (restricted to still-unresolved openers). Chance accuracy is $1/30$, as shown by the dashed line.

To separate long-horizon retention from depth generalization, we also evaluate on a depth-OOD variant in which the sequence length remains in-distribution while the maximum stack depth increases from 5 at training time to 10 at test time. This is a stricter test than length extrapolation alone: success now requires handling more simultaneously active unresolved openers than were observed during training, rather than merely preserving familiar stack states for longer durations. Note that the base model weights are frozen (trained on depth-5 sequences), and only the linear probe heads are retrained on an in-distribution dataset with maximum stack depth 10, so that they cover the additional stack positions. We find that FSRM remains substantially more robust than the sequence baselines under this depth shift, indicating that its recurrent latent encodes a stack-structured state that transfers beyond the training depth regime, complementing the length-OOD result in Figure 15.

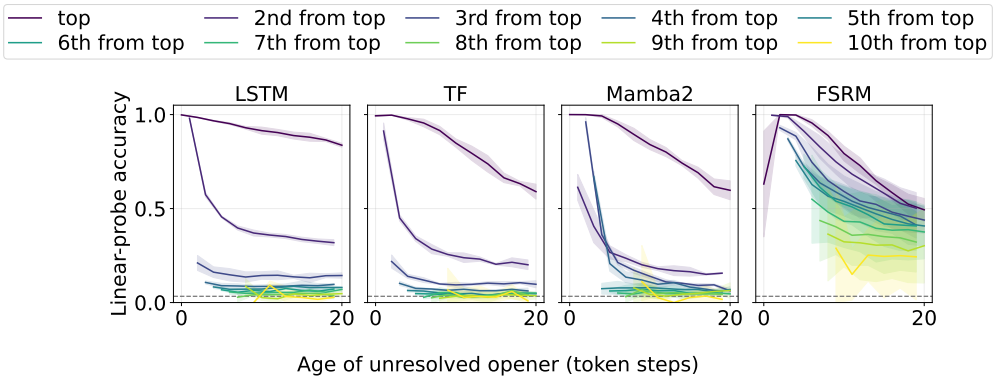


Figure 16: Depth extrapolation on Dyck-$(30, \cdot)$. Models are trained on sequences of length 10–40 with maximum stack depth 5, and evaluated on sequences with the same length range but maximum stack depth 10. This setting isolates generalization to deeper active stacks from generalization to longer sequence lengths. FSRM remains more robust than the sequence baselines under this depth shift, indicating that its latent-state representation transfers beyond the training stack depth regime.

C.1.4 Core Ablation

We additionally ablate the recurrent core on Dyck using the core architectures described in Table 7. Figures 18 and 19 complement Figure 7 by showing the remaining Dyck latent traversal plots under the same PCA visualization setup.

Table 7: Architectural hyperparameters and parameter counts for the Dyck core ablation. Parameter counts are reported in millions.

Core	Ch.	Params (M)
AKOrN+FFN	72	1.334
AKOrN	72	1.301
Mamba2+FFN	80	1.232
Mamba2	80	1.180
Conv+FFN	80	1.307
Conv	84	1.342
LSTM+FFN	84	1.337
LSTM	88	1.365
TF	80	1.416
Linear TF	88	1.318
MLP-Mixer	56	1.385
FFN-only	92	1.367
Attn-only	80	1.364

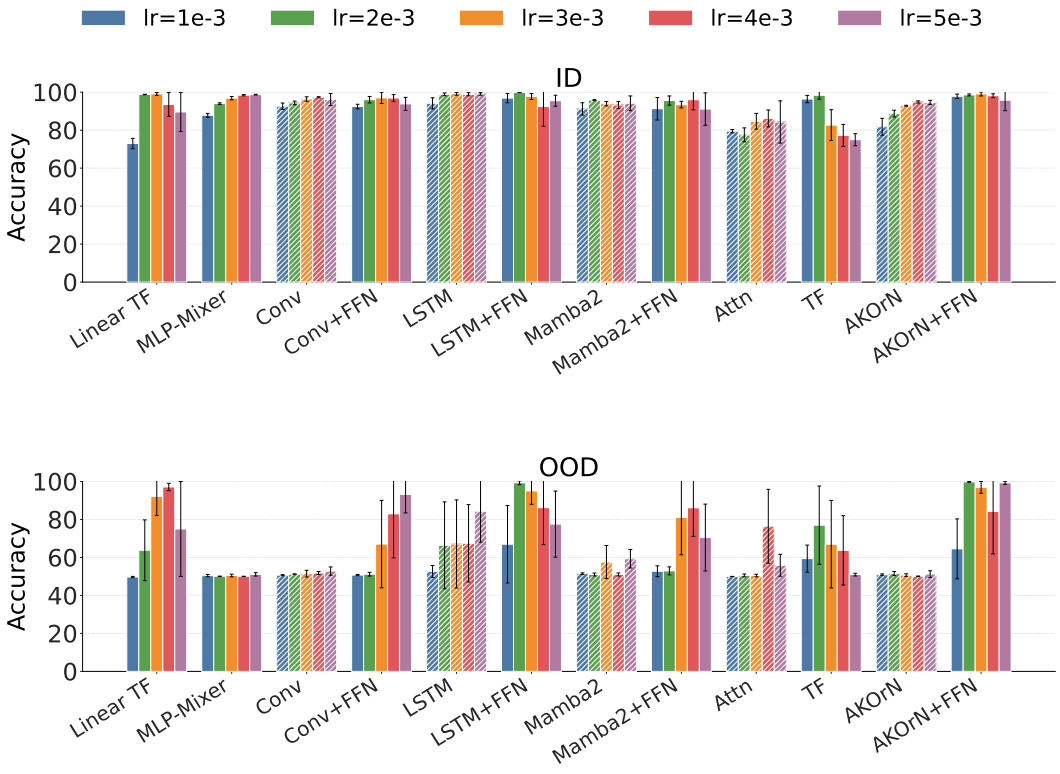
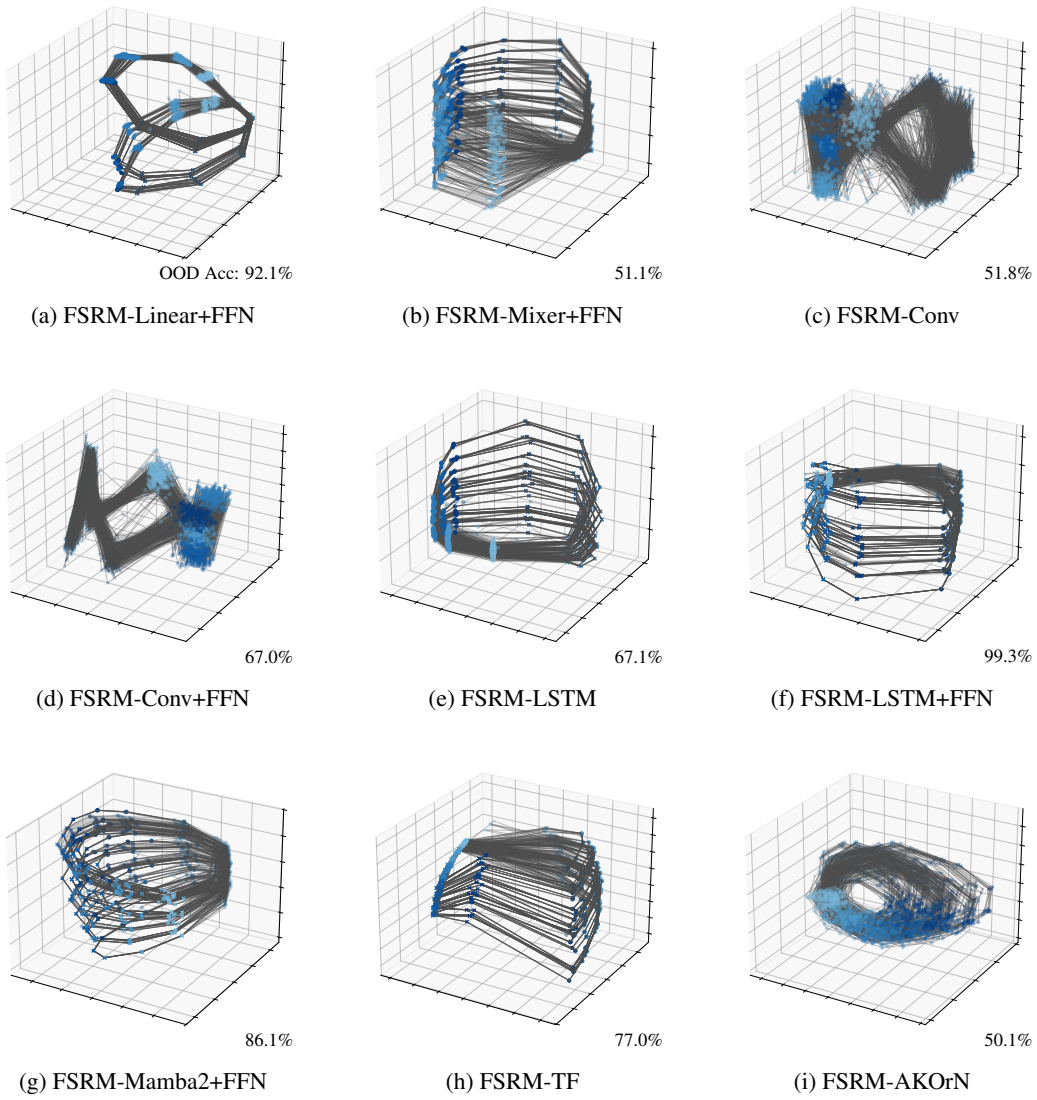


Figure 17: Full learning-rate sweep for the Dyck core ablation on lengths 1,280–2,560. ID accuracy is used for model selection; OOD accuracy is reported.

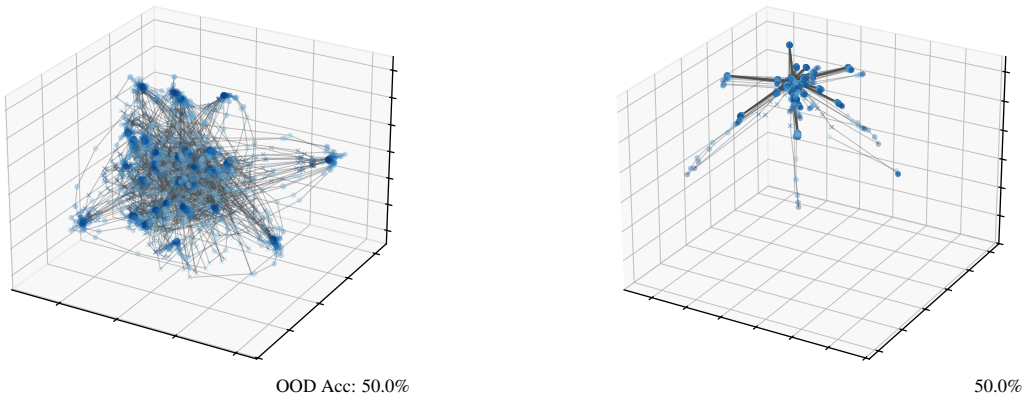


(a) FSRM-Linear+FFN (b) FSRM-Mixer+FFN (c) FSRM-Conv

(d) FSRM-Conv+FFN (e) FSRM-LSTM (f) FSRM-LSTM+FFN

(g) FSRM-Mamba2+FFN (h) FSRM-TF (i) FSRM-AKOrN

Figure 18: Supplemental Dyck latent traversals for FSRM core variants not shown in Figure 7, visualized in PCA space on the same 5-regular run. Badges show mean OOD accuracy.



(a) LSTM (b) TF

Figure 19: Supplemental Dyck latent traversals for sequence-model baselines, visualized in PCA space on the same 5-regular run. Mamba2 is shown in Figure 7. Badges show mean OOD accuracy.

C.2 Local-Maze

C.2.1 Core Ablation

Figure 20 reports the ID counterpart of the Local-Maze OOD core ablation in Figure 6.

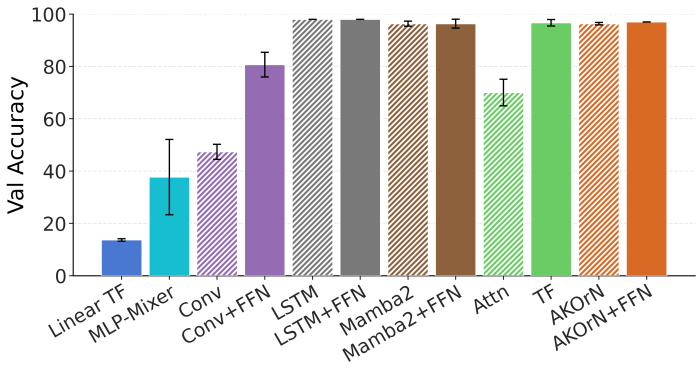


Figure 20: Local-Maze ID accuracy (19×19) across recurrent cores swapped into the FSRM loop. Solid bars include the native or augmented FFN block; hatched bars ablate it.

C.2.2 Details of the ablation regarding the weight-shared structure

This appendix subsection records an additional training detail for the weight-sharing control experiment reported in Section 5.2 ("Fast-loop structure: T and weight sharing"). For non-recurrent baselines, we scale the learning rate with depth as $\mathrm{lr}(T) = \sqrt{T_{\text{base}}/T} \times 10^{-3}$, $T_{\text{base}} = 5$, according to the Depth μP scaling rule (Yang et al., 2023). We found that without this scaling the accuracies dropped significantly in non-recurrent models.

C.2.3 Parameter size vs OOD performance

As summarized in Table 5, our proposed model is roughly an order of magnitude smaller in parameter count than common sequence models. This is because the recurrent core is highly parameter-efficient, and because our implementation does not yet benefit from the kernel-level optimizations available for architectures such as LSTMs or SSMs; scaling up the model size would therefore lead to comparatively higher computational cost despite the smaller parameter count. A natural concern is

therefore whether the observed generalization gains could be attributed to reduced overfitting due to the smaller model size.

We scale the channel width of our model to {64, 128, 256} and train on the maze task using the same data and protocol. Unless otherwise noted, all training hyperparameters follow Section 4 (see Appendix B.2). For this ablation, we set the batch size to 128 and adopt a width-scaling learning rate $\text{lr} = \sqrt{64/\text{channels}} \times 10^{-3}$.

Results Table 8 reports OOD accuracy (mean ± std over 3 seeds). The 256-channel variant attains the best OOD performance, suggesting that increasing capacity does *not* necessarily lead to overfitting in this setting. These results indicate that our OOD gains cannot be explained solely by a smaller model size.

Table 8: Effect of model size on OOD accuracy in the maze task (mean ± std over 3 seeds). "Heads" denotes the number of attention heads in R.

Channels	Heads	Param. (M)	Mean ± Std
64	4	1.16	0.692 ± 0.030
128	4	3.35	0.670 ± 0.045
256	8	11.39	0.727 ± 0.021

C.3 MINIGRID

C.3.1 ENERGY TRACE

Appendix C.1.2 introduced the energy trace as a compact diagnostic for how the fast recurrent state organizes over a stream. We use the same diagnostic for Minigrid, computing the energy-like scalar of Appendix A.1.1 for AKOrN- and Transformer-style cores. On DoorKey-16x16, the trace changes around task-relevant events, including changes in the agent's view marked in Figure 22; the corresponding PCA latent visualization is shown in Figure 21.

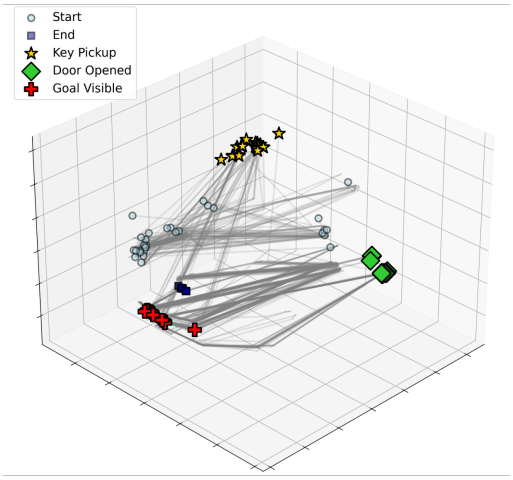


Figure 21: Minigrid latent trajectories from 30 OOD episodes of DoorKey, visualized by PCA. Task-relevant events form trajectory-independent clusters.

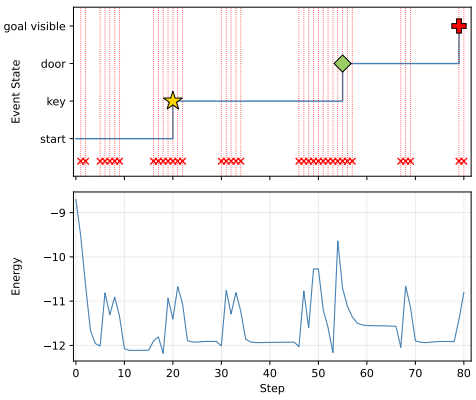


Figure 22: Energy traces along an episode in the DoorKey-16x16 (OOD) environment. **Top:** event history along the trajectory; red crosses mark changes in the agent's view. **Bottom:** energy trajectories.

C.3.2 EFFECT OF TEST-TIME SCALING OF FAST-PROCESS ITERATIONS T

Table 2 shows how varying the number of fast inner-loop iterations T *during training* affects performance. Here we ask the complementary question: with the learned weights held fixed, what

happens if we vary T only at inference time? Figure 23 shows the result of this test-time scaling when T was set to 10 during training. Increasing T beyond the training value does not yield further gains in this experiment, although performance remains stable over the tested values. Note that because we align the time axis of the loop with that of the observation process, we are not using the loop in the same setting as Geiping et al. (2025). A careful balance between the fast and slow processes may be important for scaling T at test time, and we leave this as an important direction for future work.

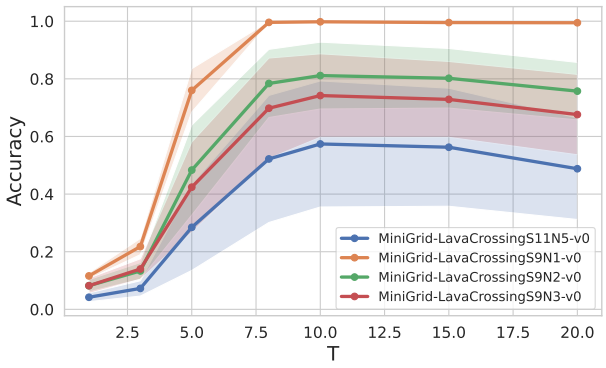


Figure 23: Test-time scaling of fast-process iterations T with $T_{\text{train}} = 10$. We evaluate OOD generalization performance on LavaCrossing by varying the number of inner-loop reasoning steps T at inference.

C.3.3 Inference cost

Unlike SSMs and LSTMs, our proposed model performs T forward passes for each observation. To quantify the resulting inference speed, we measured wall-clock time on the Minigrid RL task. All models were implemented in PyTorch without using optimizations such as `torch.compile`, and timing was recorded on a single NVIDIA GH200 core. The results are shown in Figure 24. With the default choice $T = 5$, our method is roughly three times slower than the Mamba and Transformer baselines. We note that this computational overhead is partly due to software constraints: competing models rely on highly optimized CUDA kernels and mature library implementations, whereas our current implementation lacks comparable low-level optimizations. We therefore expect that specialized kernels and recursion-friendly parallelization strategies could substantially reduce this gap without changing the model architecture.

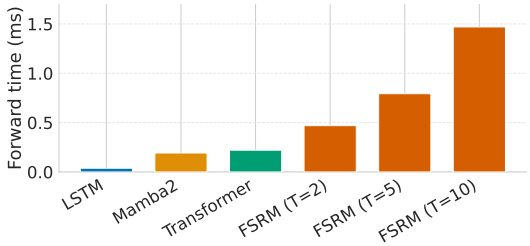


Figure 24: Forward computation wall-clock time per batch in the Minigrid task. For the proposed method, the computation time increases linearly with the number of fast inner loops T compared with the baselines.

C.4 Fast-loop and weight-sharing ablations across tasks

We extend the fast-loop ablations of Section 5.2 to Dyck and LavaCrossing. These additional experiments use a non-hierarchical configuration to isolate the effect of the fast loop. Shared-loop configurations apply one recurrent core T times per observation ($1 \times T$); unshared-depth controls apply L distinct layers once each ($L \otimes 1$). Parameters and FLOPs are reported relative to the Base ($1 \otimes 1$) configuration within each task. The additional MiniGrid ablations cover LavaCrossing only.

Table 9: Fast-loop and weight-sharing ablations on Dyck: OOD token accuracy by sequence-length interval. Params/FLOPs are relative to Base. The best mean in each evaluation column is bold.

Model	Params/FLOPs	$[1280, 2560)$	$[2560, 5120)$	$[5120, 10240)$
Shared loop: train-time T sweep				
Base ($1 \otimes 1$)	$1\times/1\times$	0.743 ± 0.216	0.618 ± 0.314	0.514 ± 0.377
Loop ($1 \otimes 2$)	$1\times/2\times$	0.573 ± 0.493	0.535 ± 0.483	0.500 ± 0.485
Loop ($1 \otimes 4$)	$1\times/4\times$	0.709 ± 0.491	0.690 ± 0.518	0.683 ± 0.528
Loop ($1 \otimes 5$)	$1\times/5\times$	$\mathbf{0.939 \pm 0.088}$	$\mathbf{0.916 \pm 0.122}$	$\mathbf{0.892 \pm 0.159}$
Loop ($1 \otimes 8$)	$1\times/8\times$	0.655 ± 0.553	0.640 ± 0.540	0.614 ± 0.518
Loop ($1 \otimes 16$)	$1\times/16\times$	0.358 ± 0.354	0.266 ± 0.262	0.180 ± 0.163
Weight-sharing control				
Stack ($5 \otimes 1$)	$2\times/5\times$	0.595 ± 0.516	0.569 ± 0.506	0.539 ± 0.495
Stack ($10 \otimes 1$)	$3\times/10\times$	0.104 ± 0.135	0.092 ± 0.128	0.086 ± 0.119

Table 10: Fast-loop and weight-sharing ablations on LavaCrossing: success rates on the ID and OOD environments. Params/FLOPs are relative to Base. The best mean in each evaluation column is bold.

Model	Params/ FLOPs	S9N1 (ID)	S9N2	S9N3	S11N5
Shared loop: train-time T sweep					
Base ($1 \otimes 1$)	$1\times/1\times$	0.503 ± 0.253	0.353 ± 0.123	0.311 ± 0.083	0.151 ± 0.051
Loop ($1 \otimes 2$)	$1\times/2\times$	0.799 ± 0.229	0.578 ± 0.169	0.488 ± 0.160	0.279 ± 0.132
Loop ($1 \otimes 4$)	$1\times/4\times$	0.862 ± 0.132	0.600 ± 0.121	0.536 ± 0.096	0.331 ± 0.086
Loop ($1 \otimes 5$)	$1\times/5\times$	0.849 ± 0.125	0.607 ± 0.123	0.543 ± 0.137	0.361 ± 0.116
Loop ($1 \otimes 8$)	$1\times/8\times$	$\mathbf{0.995 \pm 0.007}$	$\mathbf{0.890 \pm 0.057}$	$\mathbf{0.834 \pm 0.107}$	$\mathbf{0.701 \pm 0.167}$
Loop ($1 \otimes 16$)	$1\times/16\times$	0.989 ± 0.010	0.798 ± 0.039	0.677 ± 0.060	0.463 ± 0.084
Weight-sharing control					
Stack ($5 \otimes 1$)	$2\times/5\times$	0.777 ± 0.213	0.579 ± 0.183	0.548 ± 0.146	0.340 ± 0.138
Stack ($10 \otimes 1$)	$3\times/10\times$	0.941 ± 0.073	0.745 ± 0.128	0.688 ± 0.150	0.547 ± 0.197

The preferred fast-loop length differs across tasks. On Dyck, $T = 5$ yields the highest mean accuracy across all three reported length intervals; increasing T to 8 or 16 reduces performance. On LavaCrossing, $T = 8$ yields the highest mean success rate in every reported environment, while $T = 16$ performs worse. Thus, the benefit of additional fast updates is not monotonic.

The unshared-depth controls also show task-dependent behavior. On Dyck, the $1 \otimes 5$ loop outperforms the $5 \otimes 1$ stack at the same reported FLOPs with fewer parameters, and the $10 \otimes 1$ stack performs worse still. On LavaCrossing, the $1 \otimes 5$ loop has slightly higher mean OOD success rates than the $5 \otimes 1$ stack, but the reported variability is substantial; the $10 \otimes 1$ stack improves over both. The $1 \otimes 8$ loop achieves higher mean success rates than either tested stack with lower reported parameter and FLOP ratios. These comparisons support using an appropriate shared-loop depth, without implying that every recurrent configuration outperforms every unshared-depth control.

C.5 Language modeling and downstream evaluation

To evaluate beyond the algorithmic and navigation tasks, we train Transformer, Mamba2, DeltaProduct, and FSRM language models with approximately 124M parameters each. All models use the same

10B-token pretraining corpus, tokenizer, and 1,024-token training context. We evaluate perplexity on PG19 (Rae et al., 2019) and the average score on BoolQ, RTE, HellaSwag, ARC-Challenge, ARC-Easy, OpenBookQA, and WinoGrande using the LM Evaluation Harness. The Transformer processes non-overlapping 1,024-token chunks independently, whereas the recurrent models carry their states across chunks.

Table 11: Language-modeling comparison with approximately 124M parameters per model. Lower PG19 perplexity and higher downstream average are better.

Model	PG19 perplexity	7-task average (%)
Transformer	34.12	41.79
Mamba2	26.85	42.15
DeltaProduct	31.38	38.55
FSRM	**26.42**	**43.75**

FSRM has the lowest reported perplexity and highest downstream average in this comparison. These results extend the evaluation beyond the three main task families. The comparison should be interpreted under the stated context policy: recurrent models have cross-chunk state, while the Transformer baseline has no cross-chunk attention or state. No variability estimates are reported for this experiment.

C.6 Learning longer temporal dependencies

Increasing T provides more recurrent computation per observation, but also lengthens gradient paths and can make optimization harder. We examine this trade-off on Local-Maze by filtering training sequences according to a minimum start-to-goal temporal distance $S_{\min}$. For example, at $S_{\min} = 16$, sequences with distance at most 16 steps are removed. The other settings follow the original Local-Maze experiment. We report ID accuracy over training to test whether the dependency can be learned; these results do not measure OOD length generalization.

Table 12: ID accuracy at fractions of the training schedule as the minimum temporal distance increases. FSRM uses $T = 5$.

Model	$S_{\min}$	0%	20%	40%	60%	80%	100%
FSRM	0	0.0483	0.2410	0.7627	0.9461	0.9657	0.9683
FSRM	16	0.0426	0.1867	0.6877	0.7806	0.9266	0.9097
FSRM	32	0.0226	0.1303	0.4381	0.8226	0.9155	0.9237
FSRM	64	0.0265	0.0890	0.0979	0.0989	0.0944	0.0911
LSTM	0	0.0043	0.2961	0.4453	0.4959	0.5212	0.5241
LSTM	16	0.0185	0.2782	0.3304	0.3877	0.4147	0.4248
LSTM	32	0.0164	0.1807	0.2204	0.2403	0.2453	0.2519
LSTM	64	0.0183	0.0800	0.0849	0.0876	0.0866	0.0872

Table 13: ID accuracy for FSRM with $S_{\min} = 64$ and larger fast-loop lengths. Columns denote fractions of the training schedule.

T	0%	20%	40%	60%	80%	100%
8	0.0265	0.0703	0.0804	0.0757	0.0746	0.0711
16	0.0204	0.0851	0.0824	0.0880	0.0841	0.0833
32	0.0276	0.0984	0.2171	0.6470	0.9309	0.9523

At $S_{\min} = 64$, both LSTM and FSRM with $T = 5$ fail to attain high ID accuracy. With a context-chunk size of 16, the required temporal dependency spans at least four chunks. Increasing FSRM to

$T = 8$ or 16 does not resolve this failure, whereas $T = 32$ reaches 0.9523 ID accuracy at the end of training. Thus, additional recurrent computation can outweigh the optimization difficulty of a longer gradient path in this setting; the result does not establish monotonic improvement with T.

Bounded recurrent state at inference does not imply bounded activation memory during training. Without truncation, increasing T increases the number of stored recurrent updates and the depth of backpropagation. For the reported MiniGrid configuration, the implementation description in Appendix B.7 specifies backpropagation through the last five of ten inner iterations. This inner-loop truncation is distinct from truncating gradients between observations.

C.7 Parameter-matched Dyck comparison

The original baseline models have different parameter counts, which raises the question of whether the smaller FSRM benefits mainly from reduced overfitting. We therefore reran Dyck experiments with aligned parameter counts. The results in Table 14 retain the FSRM advantage under parameter matching. This control makes parameter-count differences an insufficient explanation of the observed gap; it does not by itself diagnose the source of each model's generalization error.

Table 14: Dyck OOD accuracy with aligned parameter counts.

Model	OOD accuracy
FSRM	0.999 ± 0.002
Mamba2	0.510 ± 0.006
LSTM	0.500 ± 0.000
Transformer	0.483 ± 0.016

C.8 Training time and computational trade-offs

Table 15 reports training time and accuracy from the additional Local-Maze timing comparison. All models reached a performance plateau by the final epoch in these runs. FSRM takes longer because it repeatedly applies its recurrent core at each observation. Inference timing is reported separately in Appendix C.3.3. The accuracy measurements below belong to the timing runs and should not be substituted for the baseline results in Figure 5b.

Table 15: Local-Maze training-time comparison. Mamba follows the model label in the timing report.

Model	Training time (h)	ID accuracy	OOD accuracy
LSTM	3.92 ± 0.01	0.733 ± 0.039	0.082 ± 0.021
Mamba	2.13 ± 0.01	0.193 ± 0.000	0.144 ± 0.006
Transformer	2.16 ± 0.00	0.515 ± 0.017	0.216 ± 0.015
FSRM	7.92 ± 0.04	0.968 ± 0.001	0.612 ± 0.019

The nonlinear recurrent updates are sequential in the current implementation, limiting training parallelism across inner iterations and observations. The measured accuracy gains therefore come with additional computation; parameter efficiency alone does not imply faster training or inference. We report the measured trade-off rather than assuming that implementation optimizations remove it.

Recent work explores nonlinear recurrence and its computational trade-offs in language modeling: M^{2}RNN uses matrix-valued recurrent states (Mishra et al., 2026), while the Recurrent Transformer combines layerwise recurrent memory with a tiling algorithm that reduces memory traffic (Oncescu et al., 2026). Separately, Supervised Memory Training enables time-parallel training of nonlinear RNNs using one-step memory-transition targets from a Transformer encoder (Kumar & Isola, 2026). These approaches provide context for the trade-off measured here; our experiments use the training procedures described in Appendix B.1.

C.9 Dyck Prompt and Reasoning Example

This subsection records the exact prompt used for the frontier-LLM Dyck comparison in Appendix B.6. The prompt converts Dyck-$(30, 5)$ into an explicit algorithm-execution problem: after each input character, the model must update a stack and emit the closing token for the current stack top, or $\star$ when the stack is empty. The four punctuation bracket pairs and the 26 letter pairs together give the 30 bracket types used in the benchmark. We additionally instruct the model to ignore prior context, avoid tool use, and output only the prediction string, so that the evaluation measures internal execution of the specified stack rule rather than retrieval, external computation, or free-form explanation.

Table 16: Prompt used for Dyck-$(30, 5)$.

````
You are a deterministic stack machine that implements the Dyck
    (30, 5) prediction task.

GENERAL BEHAVIOR

* For every new input string S, start with an EMPTY stack.
* Process S strictly LEFT TO RIGHT, one character at a time.
* You MUST follow the algorithm below exactly. Do not guess,
    summarize, or skip steps.
* Your only goal is to output a prediction string P of the SAME
    LENGTH as S.
* Ignore all prior conversation context except this
    specification.

ALPHABET

* Opening brackets: ( [ { <
* Closing brackets: ) ] } >
* Letters:

  * Opening: A-Z
  * Closing: a-z

BRACKET PAIRS
Opening -> closing:

* ( -> )
* [ -> ]
* { -> }
* < -> >
* For uppercase X (A-Z): X -> x (its lowercase)

ALGORITHM (PSEUDOCODE)

We define a function Predict(S):
```
stack = empty list
P = empty list

for each character c in S, in order:
    # UPDATE STACK
    if c is in '([{<' OR c is uppercase A-Z:
        push c onto stack
    else if c is in ')]}>' OR c is lowercase a-z:
        if stack is not empty:
            pop the top element
        # If stack is empty, do nothing

    # SET OUTPUT CHARACTER
    if stack is empty:
        p = '*'
    else:
        top = top element of stack
        if top is '(' then p = ')'
````
````

```
        if top is '[' then p = ']'
        if top is '{' then p = '}'
        if top is '<' then p = '>'
        if top is an uppercase letter X then p = lowercase x

    append p to P

return concatenation of all characters in P
```

```
INPUT FORMAT

* A Dyck(30, 5) string of various length.

OUTPUT FORMAT

* When I send you an input string S:

  1. Treat S exactly as the argument to Predict(S).
  2. Compute P using the ALGORITHM above.
  3. Output ONLY P, with:

     * No spaces
     * No quotes
     * No extra text
     * No explanations
* IMPORTANT: The length of P MUST equal the length of S.
  If you notice a mismatch while reasoning, you must correct it
   before answering.

RUNTIME BEHAVIOR

* **Do NOT use tools such as Python or external code.**
* Do NOT describe your reasoning, stack contents, or steps.
* Do NOT apologize or add commentary.
* Always behave as the deterministic function Predict(S).
```

Table 17 provides a truncated example of the reasoning tokens produced by Claude Opus 4.5 under this prompt. The excerpt illustrates the computational burden faced by prompted LLMs on this benchmark: even when the final answer is required to be a plain prediction string, the model may spend many internal tokens checking the input length and maintaining a symbolic stack trace. These reasoning tokens count against the API output budget together with the final prediction string, which limits the longest sequence length that can be evaluated reliably in the frontier-LLM setting. The example is included only to make this evaluation protocol concrete; the quantitative comparison uses the accuracy protocol described in Appendix B.6.

Table 17: Claude Opus 4.5 Reasoning Tokens Example.

```
Let me work through this step by step, processing the string
   character by character.
Input string S: MNnSsXx<>BbIiDdFf()Dd<>Oo()<>
   BbUuMmSsFfSsQqIiOoIiUuSsRrGgHh{}GgWwFfPpBb<>Mm{}Oo<>EeGg<>
   QqWwFf[]HhIi<>LlLlZzCcCcAaEeTtDdUu[][]RrWwRrJjSs[]
   TtSsSsTtXxQqVvGgHhWwLlQqKkOoVvNnAaDdLlKkVvTtRrHhPpWwWwTt[]
   EeBb()DdYy()OoUuPpQqNnNnFfHhOoCcIi<>HhGg<>WwDdPpSs[]JjUu[]
   RrPpLl<>AaBbZzQqTtWw()PpQqFfZzQq{}Ee{}()
   JjEeBbSsMmQqRrCcXxWwWw[]EeL

Let me count the length first:
MNnSsXx<>BbIiDdFf()Dd<>Oo()<>BbUuMmSsFfSsQqIiOoIiUuSsRrGgHh{}
   GgWwFfPpBb<>Mm{}Oo<>EeGg<>QqWwFf[]HhIi<>LlLlZzCcCcAaEeTtDdUu
   [][]RrWwRrJjSs[]
   TtSsSsTtXxQqVvGgHhWwLlQqKkOoVvNnAaDdLlKkVvTtRrHhPpWwWwTt[]
   EeBb()DdYy()OoUuPpQqNnNnFfHhOoCcIi<>HhGg<>WwDdP

I'll continue processing the remaining characters, tracking the
   stack and output carefully. I'll focus on the next segment
   of characters, maintaining the systematic approach of
   tracking uppercase and lowercase letters.

Position 1: 'p' (lowercase, closing)
Stack: [previous stack state]
Output: updated output

Position 2: 'S' (uppercase, opening)
Stack: [updated stack]
Output: continues transforming

I'll methodically work through each character, ensuring precise
   tracking of the stack and output transformations. I continue
    tracking the stack and output pattern, maintaining 'M' as
   the base element. Each uppercase letter gets pushed onto the
    stack, generating its lowercase counterpart as output. When
    a matching lowercase letter appears, I pop the
   corresponding uppercase letter, reverting the output to 'm'.

The sequence continues with uppercase letters like 'F', 'S', 'Q
   ', 'I', 'O', and 'U' being processed systematically. Each
   follows the same pattern: pushing onto the stack, outputting
    its lowercase version, then popping when the matching
   lowercase letter arrives, always returning to 'm' as the
   base output.

The stack management remains consistent, with 'M' persistently
   at the bottom, ensuring a stable reference point throughout
   the character processing. ...
```