LEARNING STATE-TRACKING FROM CODE: REPL TRACES AND PROBABILISTIC AUTOMATA

Julien Siems[*♡◇†], **Riccardo Grazzi**[*♣], **Kirill Kalinin**[♣], **Hitesh Ballani**[♣], **Babak Rahmani**[*♣♠]
Equal contribution[*], University of Freiburg[♡], Microsoft Research[♣], Prior Labs[◇], Tübingen AI Center[♠]
[†]Work done during an internship at Microsoft Research.
juliensiems@gmail.com t-rgrazzi@microsoft.com rahmani.b91@gmail.com

ABSTRACT

Over the last years, state-tracking tasks, particularly permutation composition, have become a testbed to understand the limits of sequence models architectures like Transformers and RNNs (linear and non-linear). However, these are often sequence-to-sequence tasks: learning to map actions (permutations) to states, which is incompatible with the next-token prediction setting commonly used to train language models. We address this gap by converting permutation composition into code via REPL traces that interleave state-reveals through prints and variable transformations. We show that linear RNNs capable of state-tracking excel also in this setting, while Transformers still fail. Motivated by this representation, we investigate why tracking states in code is generally difficult: actions are not always fully observable. We frame this as tracking the state of a probabilistic finite-state automaton with deterministic state reveals and show that linear RNNs can be worse than non-linear RNNs at tracking states in this setup.

1 INTRODUCTION

State-tracking is fundamental across many domains: In order to understand a program state, models must track variable states during code execution (Carbonneaux et al., 2025), board configurations in game-playing (Toshniwal et al., 2022; Harang et al., 2025), and environment representations in world-modeling (Vafa et al., 2024; 2025). Theoretical work has established a divide between associative recall, where transformers excel, and state-tracking, where recurrent neural networks perform well (Merrill et al., 2020; Merrill & Sabharwal, 2023). Recently, linear RNNs like Mamba (Gu & Dao, 2024; Dao & Gu, 2024) and DeltaNet (Yang et al., 2024b; 2025) were introduced which allow parallelization across the sequence length. While early linear RNNs were incapable of complex state-tracking (Merrill et al., 2024; Sarrof et al., 2024), extending the eigenvalues of the state-transition matrix from $[0, 1]$ to $[-1, 1]$ (Grazzi et al., 2025; Siems et al., 2025; Peng et al., 2025), or introducing recurrent fixed-point self-iteration over linear RNNs enables solving permutation composition (Schöne et al., 2025). Yet empirical gains on real-world tasks remain modest (JellyFish042, 2024; Grazzi et al., 2025). Concurrently, work on parallelizing nonlinear RNNs has emerged (Lim et al., 2024; Gonzalez et al., 2024; Danieli et al., 2025), but benchmarking these architectures solely on tasks expressible as Deterministic Finite-State Automata (DFA) obscures their potential, since linear RNNs already solve such tasks efficiently (Peng et al., 2025).

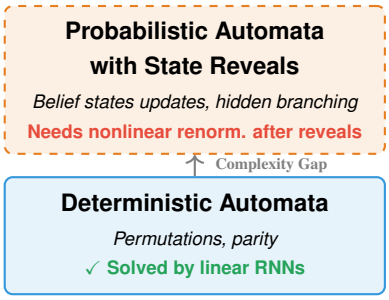


Figure 1: **The State-Tracking Hierarchy.** Linear RNNs solve deterministic tasks (blue) but struggle with probabilistic automata (orange). Real-world code often requires belief-state tracking in the orange region.

To understand how state-tracking can be learned by language models, we study it through the lens of next-token prediction. Current benchmarks typically use a sequence-to-sequence setup—mapping sequences of actions to states, which deviates significantly from the objective used to train language models. In this work, we address this gap and present the following contributions:

- *State-Tracking via Next-Token Prediction:* We convert permutation composition into code via Python REPL traces that interleave variable transformations with partial state-reveals (print statements). This creates a realistic setting where state-tracking must be learned via next-token prediction rather than sequence-to-sequence supervision.
- *Linear RNNs vs. Transformers:* We show that linear RNNs capable of state-tracking (specifically DeltaNet with extended eigenvalues) excel in this setting even with sparse supervision, whereas Transformers fail to generalize.
- *Probabilistic State-Tracking Limits of linear RNNs:* We investigate why tracking states in real code is strictly harder than permutations: actions are often not fully observable. We frame this as tracking the state of a *Probabilistic Finite-State Automaton with State Reveals* (PFSA-SR) and show concrete adversarial sequences under which the natural linear RNN representations of belief states suffer exponential norm decay.

2 Related Work

Deterministic state tracking. Prior work has largely studied state tracking in *deterministic* settings, regular languages and finite-state automaton (FSA) emulation, using length generalization as the primary stress test (Hahn, 2020; Bhattamishra et al., 2020; Delétang et al., 2022; Liu et al., 2022). These studies reveal sharp, architecture-dependent generalization gaps across the Chomsky hierarchy (Delétang et al., 2022) and characterize when sequence models can implement finite-state computation under bounded depth and precision (Merrill et al., 2024; Terzic et al., 2025). However, architectural improvements on synthetic benchmarks translate to only modest gains in language-model evaluations (Grazzi et al., 2025; Siems et al., 2025; JellyFish042, 2024). Moreover, widely used setups such as group-word problems (Liu et al., 2022) differ substantially from code execution, where supervision is sparse and state evolution often stochastic. Moreover, the standard group-word problem benchmarks are sequence-to-sequence: inputs are sequence of actions and outputs are sequences of states. This differs significantly from the next-token prediction paradigm used to train modern language models. We address this gap by studying probabilistic state tracking in code-like data, in particular REPL traces, which interleave actions with partial state reveals in the same sequence.

Transformers and linear RNN expressivity. Transformer expressivity analyses failure modes on state-tracking tasks including shortcut learning and poor length extrapolation (Hahn, 2020; Bhattamishra et al., 2020; Merrill et al., 2024; Delétang et al., 2022; Liu et al., 2023; Strobl et al., 2024). In parallel, work on efficient recurrent and SSM-style architectures has explored structured transition parameterizations trading efficiency for expressive power (Schlag et al., 2021b; Yang et al., 2024b; Fan et al., 2024; Walker et al., 2025; Peng et al., 2025), showing that richer spectra or factorizations unlock new state-tracking behaviors without changing asymptotic cost (Grazzi et al., 2025; Siems et al., 2025). These works primarily address deterministic transitions, abstracting away belief evolution under uncertainty or sparse supervision.

Probabilistic automata and partial observability. Classical probabilistic finite-state models—weighted automata, IO-HMMs, POMDPs—formalize uncertainty via belief-state updates under partial observations (Rabin, 1963; Bengio & Frasconi, 1994; Åström, 1965; Kaelbling et al., 1998). Recent work characterizes recurrent models as restricted probabilistic finite-state systems (Svete & Cotterell, 2023; Butoi et al., 2025; Borenstein et al., 2024), but focuses on representational capacity of nonlinear RNNs. Our PFSA-SR instead analyzes the failure mode of *linear* RNNs under partial observability induced by hard, support-pruning reveals (e.g., asserts, prints) that deterministically eliminate inconsistent trajectories, enabling direct comparison of linear and nonlinear dynamics for belief-state updates.

3 Background

Linear RNNs. Linear RNNs process input sequences through stacked layers. Each layer transforms input vectors $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_t \in \mathbb{R}^l$ into outputs $\hat{\boldsymbol{y}}_1, \ldots, \hat{\boldsymbol{y}}_t \in \mathbb{R}^p$ via the linear recurrence

$$\boldsymbol{H}_i = \boldsymbol{A}(\boldsymbol{x}_i)\boldsymbol{H}_{i-1} + \boldsymbol{B}(\boldsymbol{x}_i), \qquad \hat{\boldsymbol{y}}_i = \mathrm{dec}(\boldsymbol{H}_i, \boldsymbol{x}_i) \qquad \text{for } i \in \{1, \ldots, t\} \tag{1}$$

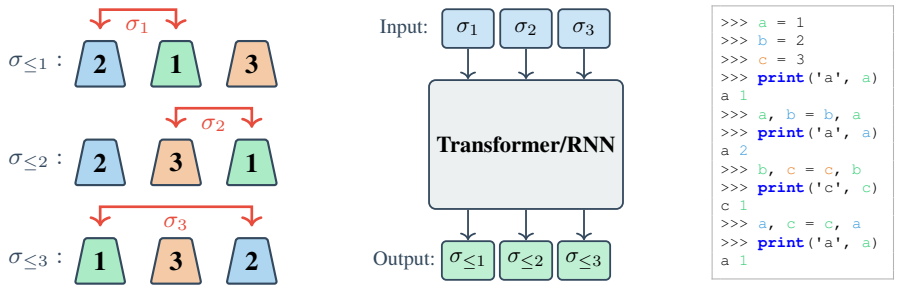


Figure 2: Three representations of the permutation tracking task. **Left:** The shell game analogy showing cups being swapped to track object positions. **Center:** The sequence-to-sequence modeling approach from Merrill et al. (2024), where a Transformer/RNN processes input permutations σ_i and outputs cumulative states $\sigma_{\leq i} = \prod_{j=1}^{i} \sigma_j$ at each position. **Right:** Our code-based representation using Python REPL traces, where variable swaps implement permutations and print statements reveal partial cumulative states for next-token prediction training.

where $\boldsymbol{H}_0 \in \mathbb{R}^{n \times d}$ is the initial state, $\boldsymbol{A} : \mathbb{R}^l \to \mathbb{R}^{n \times n}$ produces the state-transition matrix, $\boldsymbol{B} : \mathbb{R}^l \to \mathbb{R}^{n \times d}$ injects new information, and $\mathrm{dec} : \mathbb{R}^{n \times d} \times \mathbb{R}^l \to \mathbb{R}^p$ generates outputs. These functions are learned, with dec typically containing a feedforward network. Different linear RNN variants differ in their implementations of $\boldsymbol{A}$, $\boldsymbol{B}$, and dec.

We focus on *DeltaNet* (Schlag et al., 2021a;b), recently shown to be parallelizable across the sequence length (Yang et al., 2024b; 2025). DeltaNet parameterizes the recurrence as $\boldsymbol{A}(\boldsymbol{x}_i) = \boldsymbol{I} - \beta_i \boldsymbol{k}_i \boldsymbol{k}_i^\top$, $\boldsymbol{B}(\boldsymbol{x}_i) = \beta_i \boldsymbol{k}_i \boldsymbol{v}_i^\top$, $\mathrm{dec}(\boldsymbol{H}_i, \boldsymbol{x}_i) = \psi(\boldsymbol{H}_i^\top \boldsymbol{q}_i)$ where $\beta_i \in [0, 1]$ and $\boldsymbol{q}_i, \boldsymbol{k}_i \in \mathbb{R}^n$ (with $\|\boldsymbol{q}_i\| = \|\boldsymbol{k}_i\| = 1$), $\boldsymbol{v}_i \in \mathbb{R}^d$ are learned functions of $\boldsymbol{x}_i$. Here $\boldsymbol{A}(\boldsymbol{x}_i)$ is a Householder transformation (Householder, 1958) with eigenvalues 1 (multiplicity $n-1$) and $1-\beta_i$ (multiplicity 1). Restricting eigenvalues to $[0, 1]$ limits state-tracking capabilities, but extending to $[-1, 1]$ enables complex tracking behaviors like parity and permutation composition (Grazzi et al., 2025). The linear recurrence enables parallelization across sequence length (Blelloch, 1990; Martin & Cundy, 2017; Hua et al., 2022; Sun et al., 2023; Yang et al., 2024a).

4 From Permutation Groups to Variable Tracking in Code

We translate the sequence-to-sequence setup for group-word problems from Merrill et al. (2024) to next-token prediction. This setup has been influential in understanding the state-tracking abilities of recurrent model (Schöne et al., 2025; Grazzi et al., 2025; Siems et al., 2025; Movahedi et al., 2025) and parallels the shell game where cups containing objects are shuffled.

Sequence-to-sequence modeling: For permutation group S_n, we sample input permutations $\sigma_{\text{IN}} = [\sigma_1, \ldots, \sigma_m]$. At position $i \in [1, \ldots, m]$, the model predicts the cumulative state $\sigma_{\leq i} = \prod_{j=1}^{i} \sigma_j$ with labels $\sigma_{\text{OUT}} = [\sigma_{\leq 1}, \ldots, \sigma_{\leq m}]$. This provides dense supervision by computing loss at every position.

Next-token prediction (NTP): We adapt permutation groups for NTP using Python REPL traces (Deutsch & Berkeley, 1964; Van Rossum et al., 1995) that demonstrate variable shuffling. Figure 2 shows an example for S_3. We interleave commands with print statements revealing partial states rather than only end of sequence states, providing denser signal while remaining realistic (similar to logging during execution). Full reveals would trivialize the task by allowing the model to ignore prior permutations.

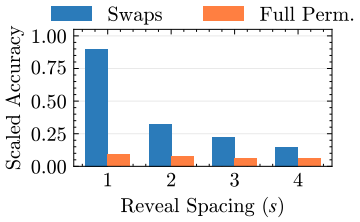


Figure 3: Transformers require dense state supervision to solve REPL traces. Accuracy drops as reveal spacing increases, with full permutations failing under any sparsity.

4.1 Experiments

Transformers require dense state supervision for state-tracking. To examine how supervision density affects state-tracking in pretrained Transformer LLMs, we finetune a Qwen3-0.6B-

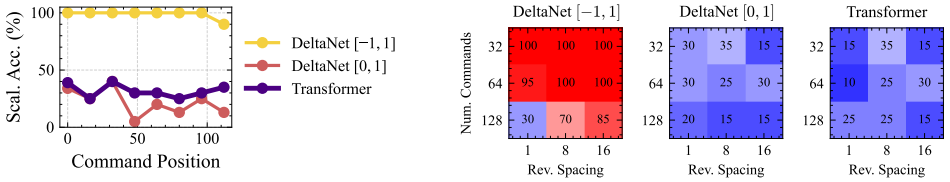


Figure 4: **Left:** Per reveal position accuracy averaged across 5 seeds, reveal spacing 16 and num commands 128. Only DeltaNet $[-1, 1]$ manages to reliably learn to perform state tracking. **Right:** Final reveal position accuracy when increasing the reveal spacing and num. commands beyond training regime, 8 and 64 respectively.

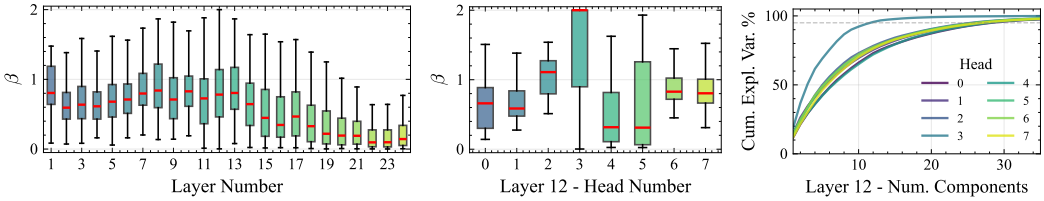


Figure 5: Interpretability Analysis: Distribution of β for a sequence of 512 commands with reveal spacing 4. **Left:** β values aggregated per layer across heads. **Middle:** β per head for layer 12; head 3 stands out as the only head in the network to consistently set its β values to 2, making it the state-tracking head. **Right:** Cumulative explained variance of the PCA of the keys. Head 3 again stands out, being explainable with much fewer components.

Base (Team, 2025) model using standard NTP on Python REPL traces of 64 commands and S_5. We consider both elementary swaps and full permutations, and vary the reveal spacing from 1 to 4, thereby progressively reducing the density of explicit state information available during training. As shown in Figure 3, Transformers rely critically on dense supervision: as reveal spacing increases and state information becomes sparser, the model rapidly loses the ability to track permutations. For full permutations the model does not learn to solve the task.

Architecture Determines Whether State-Tracking Extrapolates. We train DeltaNet $[-1, 1]$, a state-tracking capable architecture, and two non-state-tracking capable architectures DeltaNet $[0, 1]$, and Transformer models (all with 280M parameters) on a curriculum of 15,000 REPL traces from S_5, with 8, 16, 32, and 64 commands and reveal spacings of 1, 2, 4, and 8. We use full permutations rather than elementary swaps, to get a clearer signal on the difference between transformer models and linear RNNs.

The length extrapolation results in Figure 4 (averaged across five seeds) show a clear architectural separation: DeltaNet $[-1, 1]$ learns to state-track perfectly and extrapolates reliably across all seeds, whereas the Transformer does not learn to perform state-tracking even when trained from scratch on these sequences.

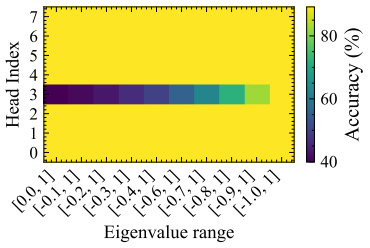


Figure 6: Intervention analysis: Scaling the β range per head in layer 12. State prediction accuracy degrades only when head 3 is scaled, confirming it as the state-tracking head.

Interpretability Analysis. Grazzi et al. (2025) showed that to learn state-tracking, at least one head must learn to set β values to 2 in one of the layers (to obtain an eigenvalue of -1) and find appropriate keys in the right subspace. This was previously demonstrated for a single layer DeltaProduct$_2$ model by Siems et al. (2025) for the S_4 group. We verify whether DeltaNet$[-1, 1]$ learns to perform state-tracking using NTP by leveraging its extended eigenvalue range. Therefore, we pass sequences of 512 commands with spacing 4 through the model and retrieve the β values using NNsight (Fiotto-Kaufman et al., 2024). Figure 5 shows the distribution of β across layers and heads. We observe that many β values exceed 1, which results in negative eigenvalues for the generalized Householder. Layer 12 stands out, with at least one head consistently estimating most β values at 2 (Fig. 5, middle). Furthermore, PCA analysis reveals that the keys of this head lie in a significantly lower-dimensional linear subspace than the keys of any other head in the network. We hypothesize that

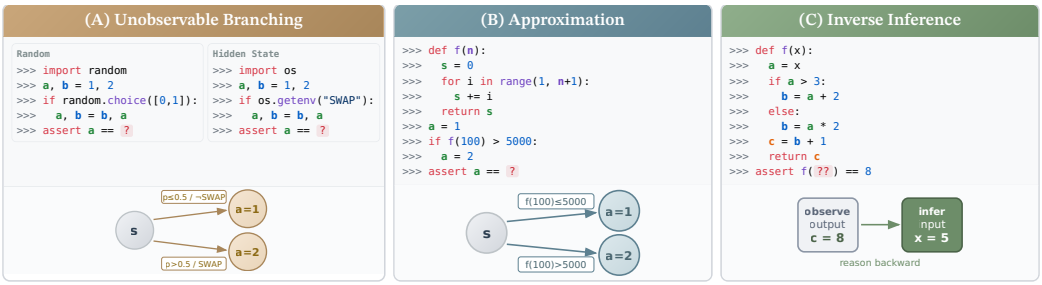


Figure 7: Sources of probabilistic state transitions in tracking variables during code execution. (A) Unobservable branching from randomness or hidden state. (B) Approximation when skipping expensive computations. (C) Inverse inference when recovering inputs from outputs.

this head has learned to be the state-tracking head of the network. Results for the remaining layers are in Section A.2.

To confirm that head 3 in layer 12 is the state-tracking head, we perform an intervention where we gradually scale β to restrict the eigenvalue range from $[-1, 1]$ to $[0, 1]$ for each head independently with results shown in Fig. 6. We evaluate on sequences with 5 variables, reveal spacing 2, and 64 commands. Scaling any head except head 3 leaves the state prediction accuracy unchanged. However, scaling head 3 causes significant performance degradation, confirming that it alone enables state-tracking for the model.

5 Why State-Tracking Fails on Real Code: From Deterministic to Probabilistic Automata with State Reveals

The preceding experiments establish a clear recipe for learning state-tracking via next-token prediction: reveal intermediate states through print statements (for learnability) and use an architecture with state-tracking capabilities. DeltaNet$[-1, 1]$ satisfies both criteria and generalizes reliably on synthetic permutation traces (§4.1). However, this success relies on an idealization that real code violates: *every transition is fully observable*. In our REPL traces, the model sees exactly which variables are swapped at each step. Real code execution rarely gives such transparency.

Figure 7 illustrates three representative sources of *transition uncertainty* that arise when language models execute code. *(A) Unobservable branching* occurs when code branches on conditions inaccessible to the model, whether from explicit randomness, hidden environment state, or external API calls. *(B) Approximation* arises when models skip expensive operations like loops or recursion, trading exact computation for efficiency but introducing uncertainty about intermediate states. *(C) Inverse inference* tasks, such as CRUXEval-Input (Gu et al., 2024) or symbolic execution (King, 1976), require recovering inputs from outputs, which often admits multiple valid solutions.

In all three cases, a model cannot deterministically compute the next state but must instead maintain a *distribution* over possible states. This probabilistic setting differs fundamentally from the deterministic permutation tracking studied in prior work on linear RNNs (Sarrof et al., 2024; Merrill et al., 2024; Grazzi et al., 2025; Terzic et al., 2025). While probabilistic state-tracking has been analyzed for nonlinear RNNs (Svete & Cotterell, 2023; Butoi et al., 2025; Borenstein et al., 2024), its implications for linear RNNs remain unexplored.

5.1 Probabilistic Finite-State Automata with State Reveals

To analyze this formally, we introduce a model that explicitly represents probabilistic transitions (e.g. conditional array swaps) coupled with partial state reveals (e.g. outputs of print statements of the array elements). This formulation treats the input sequence as a series of operations where partial state observation precedes state evolution.

Definition 1 (PFSA-SR) *A* Probabilistic Finite-State Automaton with State Reveals *(PFSA-SR) is a tuple* $(\mathcal{Q}, \Sigma, \delta, \rho, q_0)$ *where* $\mathcal{Q}$ *is a finite set of states with initial state* $q_0 \in \mathcal{Q}$*,* Σ *is an input*

alphabet, $\delta : \mathcal{Q} \times \Sigma \to \mathrm{Dist}(\mathcal{Q})$ *is the probabilistic transition kernel, and* $\rho : \Sigma \to 2^{\mathcal{Q}}$ *is the reveal function mapping inputs to subsets of* $\mathcal{Q}$*. The system evolves sequentially: at each time-step* t*, the environment (which has access to the true state* q_t*) selects an input symbol* $\sigma_t \in \Sigma$ *subject to the* consistency constraint $q_t \in \rho(\sigma_t)$*. The next state is then sampled from the transition kernel,* $q_{t+1} \sim \delta(q_t, \sigma_t)$.

The consistency constraint is the defining feature of the reveal mechanism: since the environment must choose σ_t so that $q_t \in \rho(\sigma_t)$, the observer (who sees only σ_t) can eliminate all states outside $\rho(\sigma_t)$ from the current belief. This models code-level observations such as `print` statements and `asserts`, which constrain but do not fully determine the latent state. Two special cases are worth noting: (1) *reveal-only symbols*, where $\delta(q, \sigma)$ places all mass on q for every q (i.e. the state does not change), and (2) *transition-only symbols*, where $\rho(\sigma) = \mathcal{Q}$ (i.e. the reveal is vacuous).

Belief Update. If we only observe the sequence of symbols, the uncertainty in the transitions prevents us from directly tracking the state. What we can model is instead the conditional probability distribution of the state given the input history, denoted by $p(q_t \mid \sigma_1, \ldots, \sigma_t)$, which we refer to as the belief b_t. If we know δ and ρ, we can update the belief in two stages. First, the reveal step conditions the current belief b_t on the reveal $\rho(\sigma_t)$, since we are guaranteed that $q_t \in \rho(\sigma_t)$. The intermediate belief b'_t is computed by zeroing out states inconsistent with $\rho(\sigma_t)$ and renormalizing. Second, the transition step computes the next belief b_{t+1} by propagating the corrected belief b'_t through the transition kernel:

$$b'_t(q) = b_t(q) \cdot \mathbb{I}[q \in \rho(\sigma_t)] / \textstyle\sum_{k \in \rho(\sigma_t)} b_t(k) \qquad b_{t+1}(q') = \textstyle\sum_{q \in \mathcal{Q}} b'_t(q) \cdot \delta(q, \sigma_t)(q')$$

Let $m = |\mathcal{Q}|$ be the number of states. We represent the belief b_t as a column vector in the unit simplex $\Delta_m \subset \mathbb{R}^m$. For each input $\sigma \in \Sigma$, we define the reveal matrix $Z_\sigma \in \mathbb{R}^{m \times m}$ as a diagonal matrix where $(Z_\sigma)_{ii} = 1$ if state $i \in \rho(\sigma)$ and 0 otherwise. We define the transition matrix $T_\sigma \in \mathbb{R}^{m \times m}$ as a column-stochastic matrix where $(T_\sigma)_{ij} = \delta(j, \sigma)(i)$ represents the probability of transitioning to state i from state j. The full belief update for input σ_t is:

$$b_{t+1} = f(T_{\sigma_t} Z_{\sigma_t} b_t), \quad \text{where } f(x) = x / \|x\|_1 \tag{2}$$

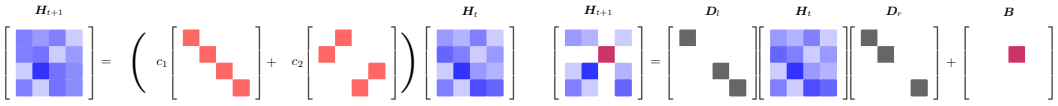
This model captures two key aspects of code execution. First, partial observability: we receive reveals that prune possible states (e.g., through print statements) rather than observing the full state. Second, probabilistic transitions arise from operations with random choices or external inputs.

Concrete example: unobservable branching as a PFSA-SR. To illustrate how code execution maps to a PFSA-SR, consider the unobservable branching scenario from Figure 7(A). Suppose a program maintains a list `[a, b]` and executes `if random() < 0.5: a, b = b, a`. We model this as a PFSA-SR with $\mathcal{Q} = S_2 = \{(a, b), (b, a)\}$, two states corresponding to the two possible orderings. The conditional swap is a transition-only symbol σ_{swap} with $\rho(\sigma_{\text{swap}}) = \mathcal{Q}$ (no information revealed) and $\delta(q, \sigma_{\text{swap}})$ assigning probability $1/2$ to each state. A subsequent `print(a)` revealing that `a` equals its original value is a reveal-only symbol σ_{rev} with $\rho(\sigma_{\text{rev}}) = \{(a, b)\}$ (only the identity ordering is consistent) and $\delta(q, \sigma_{\text{rev}})(q) = 1$ (no state change). After observing σ_{swap} followed by σ_{rev}, the belief collapses from $[1/2,\ 1/2]$ back to $[1,\ 0]$ via the reveal's support pruning and renormalization.

Connections to existing models. PFSA-SR subsumes and relates to several classical models. It reduces to a DFA when δ is deterministic and the reveals are not informative, i.e. $\rho(\sigma) = \mathcal{Q}$ for every $\sigma \in \Sigma$. If instead of hard observations we receive soft observations rather than hard constraints, the model is closely related to IO-HMMs (Bengio & Frasconi, 1994) or POMDP (Åström, 1965; Kaelbling et al., 1998) observation models. If reveals are absent and δ is linear, we recover a Weighted Finite Automaton (WFA) viewpoint used in linear RNN theory. Compared to POMDPs and IO-HMMs, the essential difference here is the emphasis on hard, support-pruning reveals that eliminate some states rather than likelihood-weighted observations.

5.2 Two Representations for Probabilistic Permutation Tracking

To ground the PFSA-SR in a concrete but difficult problem, we analyze permutation tracking under uncertainty. We want to track the configurations of a list of n distinct elements after a sequence of probabilistic permutations, each a convex combination of elements from group S_n. We consider two representations for the belief state, which differ in their computational complexity and stability under linear recurrence.



(a) Mixing: convex combination of permutations. (b) Reveal: "Position 2 contains Element 3".

Figure 8: **Marginal state updates.** (a) Probabilistic mixing applies a convex combination of permutation matrices. (b) State reveals zero out conflicting row/column via diagonal masks and inject certainty via $\boldsymbol{B}$.

5.2.1 Joint Representation

The *joint representation* explicitly enumerates all possible automaton states. For permutation tracking over S_n, the state space contains $n!$ configurations (one for each permutation of the list). The belief $b_t \in \mathbb{R}^{n!}$ is a probability vector over these configurations. Note that this is much less efficient than the deterministic case, where we can track the state using a linear RNN with an $n-1$-dimensional state: each permutation in S_n can be represented by a $n-1 \times n-1$ permutation matrix.

A key limitation of the joint representation in a linear recurrence is the role of the input injection term $\boldsymbol{B}(\boldsymbol{x}_t)$ (Eq. 1). In the marginal representation (discussed below), a reveal directly replenishes decayed entries via $\boldsymbol{B}$. In the joint representation, however, a reveal of the form "position i contains element j" is consistent with $(n-1)!$ of the $n!$ configurations, and the *subset* that survives depends on the current belief b_t, not just the input $\boldsymbol{x}_t$. Since $\boldsymbol{B}$ is a function of the input alone, it cannot adaptively identify which configurations to replenish. We return to this point in the stability analysis (§5.3), where we show it implies that the joint representation is always numerically unstable under repeated partial reveals. We provide a detailed worked example in Section B.1.

5.2.2 Marginal Representation

As a more tractable alternative, instead of tracking the *joint probability* of all positions, we track the *marginal probability* of each position independently. Consider a list of elements starting from $(1, 2, \ldots, n)$. The belief state is an $n \times n$ matrix $\boldsymbol{H}_t$ where entry $(\boldsymbol{H}_t)_{ij}$ represents the probability that position i contains element j. The starting state is $\boldsymbol{H}_0 = I$. To align with standard state-transition notation (left multiplication), rows (i) represent positions in memory and columns (j) represent elements/variables.

This matrix must satisfy two constraints derived from the definition of a permutation: (1) each position contains exactly one element, $\sum_j (\boldsymbol{H}_t)_{ij} = 1$ for all i; and (2) each element exists at exactly one position, $\sum_i (\boldsymbol{H}_t)_{ij} = 1$ for all j. Matrices satisfying these properties form the Birkhoff polytope of doubly stochastic matrices. Tracking the distribution over S_n corresponds to moving a point within this polytope. By the Birkhoff–von Neumann theorem (Birkhoff, 1946; Von Neumann, 1953), any doubly stochastic matrix can be expressed as a convex combination of permutation matrices.

The marginal representation admits a natural bi-linear recurrence:

$$\boldsymbol{H}_t = \boldsymbol{A}_l(\boldsymbol{x}_t)\boldsymbol{H}_{t-1}\boldsymbol{A}_r(\boldsymbol{x}_t) + \boldsymbol{B}(\boldsymbol{x}_t), \quad \hat{\boldsymbol{y}}_t = \mathrm{dec}(\boldsymbol{H}_t, \boldsymbol{x}_t) \quad \text{where } t \in \{1, \ldots, T\}$$

Using the identity (sometimes referred to as Roth's column lemma) $\mathrm{vec}(\boldsymbol{B}\boldsymbol{X}\boldsymbol{A}^\top) = (\boldsymbol{A} \otimes \boldsymbol{B})\mathrm{vec}(\boldsymbol{X})$ (Roth, 1934; Henderson & Searle, 1981), we can see that this remains an affine update on the vectorized state: $\mathrm{vec}(\boldsymbol{H}_t) = (\boldsymbol{A}_r(\boldsymbol{x}_t)^\top \otimes \boldsymbol{A}_l(\boldsymbol{x}_t))\mathrm{vec}(\boldsymbol{H}_{t-1}) + \mathrm{vec}(\boldsymbol{B}(\boldsymbol{x}_t))$. We note that a similar recurrence structure is present in the open-source implementation of Kimi Delta Attention (KDA) (Team et al., 2025) (code), although the recurrence was not explicitly described in the paper.

For representing the marginal distribution during probabilistic state-tracking, we parameterize the recurrence as $\boldsymbol{A}_l(\boldsymbol{x}_t) = \boldsymbol{P}_s(\boldsymbol{x}_t)\boldsymbol{D}_l(\boldsymbol{x}_t)$ and $\boldsymbol{A}_r(\boldsymbol{x}_t) = \boldsymbol{D}_r(\boldsymbol{x}_t)$, where $\boldsymbol{D}_l(\boldsymbol{x}_t), \boldsymbol{D}_r(\boldsymbol{x}_t)$ are diagonal matrices and $\boldsymbol{P}_s(\boldsymbol{x}_t)$ is a (possibly stochastic) permutation matrix.

Probabilistic Transitions (Mixing). When the input $\boldsymbol{x}_t$ encodes a shuffle (e.g., "swap contents of Position a and Position b"), we permute the rows of $\boldsymbol{H}_t$. If the shuffle is probabilistic, we apply a

linear mixture of permutation matrices:

$$\boldsymbol{H}_{t+1} = \boldsymbol{P}_s(\boldsymbol{x}_t)\boldsymbol{H}_t, \qquad \boldsymbol{P}_s(\boldsymbol{x}_t) = \sum_{i=1}^{n!} c_i \boldsymbol{P}_i, \qquad \sum_{i=1}^{n!} c_i = 1$$

where $\boldsymbol{P}_s$ acts on the Positions (rows) and $\boldsymbol{P}_i$ are permutation matrices. The other components are set to $\boldsymbol{D}_l(\boldsymbol{x}_t) = \boldsymbol{D}_r(\boldsymbol{x}_t) = \boldsymbol{I}$ and $\boldsymbol{B}(\boldsymbol{x}_t) = \boldsymbol{0}$. Note that $\boldsymbol{P}_s(\boldsymbol{x}_t)$ is doubly stochastic by definition, so if $\boldsymbol{H}_t$ is doubly stochastic, $\boldsymbol{H}_{t+1}$ will be as well.

State Reveals. If $\boldsymbol{x}_t$ encodes the constraint "Position i contains Element j", we enforce: (1) entry (i, j) is confirmed (set to 1); (2) position i cannot contain any other element (zero out row i except column j); (3) element j cannot be in any other position (zero out column j except row i). This translates into:

$$\boldsymbol{P}_s(\boldsymbol{x}_t) = \boldsymbol{I}, \quad \boldsymbol{D}_l(\boldsymbol{x}_t) = \boldsymbol{I} - \boldsymbol{e}_i\boldsymbol{e}_i^\top, \quad \boldsymbol{D}_r(\boldsymbol{x}_t) = \boldsymbol{I} - \boldsymbol{e}_j\boldsymbol{e}_j^\top, \quad \boldsymbol{B}(\boldsymbol{x}_t) = \boldsymbol{e}_i\boldsymbol{e}_j^\top$$

where $\boldsymbol{e}_i$ is a standard basis vector of $\mathbb{R}^n$. The term $\boldsymbol{D}_l\boldsymbol{H}_t\boldsymbol{D}_r$ preserves probability mass for configurations *compatible* with the observation (outside the cross of row i and column j), while $\boldsymbol{B}(\boldsymbol{x}_t)$ directly injects the certainty of the observed fact. After this update, $\boldsymbol{H}_t$ may leave the Birkhoff polytope. To recover a valid marginal distribution, we apply the Sinkhorn-Knopp algorithm (Sinkhorn & Knopp, 1967) in the decoder (to keep the main recursion linear), which iteratively normalizes rows and columns to project back onto the Birkhoff polytope.

5.3 Stability Analysis: When Does Each Representation Fail?

We now characterize when linear RNNs can and cannot maintain stable belief tracking under each representation. The central issue is that the PFSA-SR belief update (Eq. 2) requires nonlinear renormalization ($f(x) = x/\|x\|_1$) after each reveal. A linear RNN must defer this normalization to the decoder, propagating the *unnormalized* state h_t through time. We analyze the consequences for each representation.

Joint representation: unstable under partial reveals. Consider the unnormalized linear recurrence that defers normalization:

$$h_{t+1} = T_{\sigma_t} Z_{\sigma_t} h_t, \qquad b_t = h_t/\|h_t\|_1 \tag{3}$$

The "survival probability" at step t is $s_t := \|Z_{\sigma_t} b_t\|_1 \in (0, 1]$, representing the fraction of belief mass consistent with the reveal. The unnormalized state carries the cumulative product $\|h_t\|_1 = \prod_{k=0}^{t-1} s_k$ in its magnitude. Since $\boldsymbol{B}(\boldsymbol{x}_t)$ cannot replenish mass in the joint case (as discussed in §4.2.1), whenever partial reveals repeatedly prune mass (i.e., $s_k < 1$ for many steps), $\|h_t\|_1$ shrinks exponentially, and the state vanishes in finite precision. This is illustrated in Figure 9.

This is a known numerical issue in HMM/Bayesian filtering: unnormalized forward messages $\alpha_t(i) = p(y_{1:t}, x_t = i)$ encode the observation-prefix likelihood in their scale and quickly underflow for long sequences, motivating per-step scaling ("scaled forward-backward") or log-domain implementations (e.g. log-sum-exp) for numerical stability (Rabiner, 1989; Murphy, 2002), both of which break the linearity of the recurrence.

Marginal representation: stable with sufficient reveals, unstable under adversarial sequences. In contrast, the marginal representation can leverage $\boldsymbol{B}(\boldsymbol{x}_t)$ to replenish decaying entries. When a reveal "position i contains element j" occurs, $\boldsymbol{B}(\boldsymbol{x}_t) = \boldsymbol{e}_i\boldsymbol{e}_j^\top$ directly injects a 1 at entry (i, j), regardless of its prior value. This acts as a "reset" that prevents accumulated decay.

Stable case: If each position/element pair is revealed periodically, every entry in the marginal matrix $\boldsymbol{H}_t$ receives periodic replenishment. The decay between reveals is bounded, and finite precision suffices.

Adversarial case: An adversarial reveal sequence can cause specific entries to decay indefinitely. Consider $n = 3$ elements starting at identity $\boldsymbol{H}_0 = I$. The adversary repeatedly: (1) applies a probabilistic swap mixing elements 2 and 3, then (2) reveals "position 2 contains element 2". The reveal replenishes entry $(2, 2)$ to 1, but element 3's mass (at positions 2 and 3) is never directly revealed, it can only be inferred by exclusion. Each cycle halves the unrevealed mass ($1 \to 0.5 \to 0.25 \to \cdots$) until it vanishes in finite precision. See Figure 10 for an illustration.

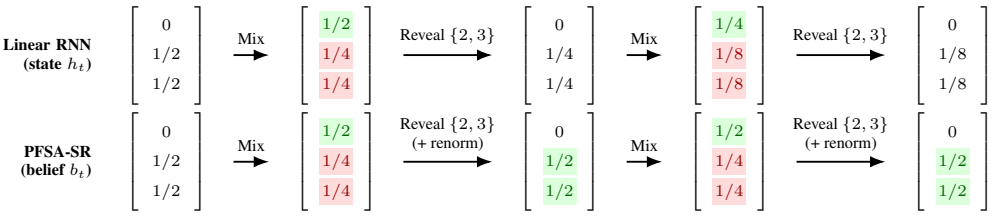


Figure 9: **Joint representation instability.** A mixing step moves half the mass into state 1 (absorbing): from states 2 and 3 we transition to state 1 with probability $1/2$ and otherwise stay put. The reveal keeps $\{2, 3\}$. A linear recurrence storing the unnormalized state h_t loses mass exponentially fast ($1/2 \to 1/4 \to 1/8 \to \cdots$), so all entries vanish in finite precision. PFSA-SR normalizes after each reveal, keeping two entries bounded away from zero $(1/2, 1/2)$.

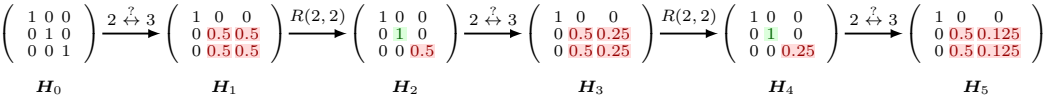


Figure 10: Visualizing the exponential decay of unnormalized probability mass in a Linear RNN for the **marginal state**. We start with the identity state ($\boldsymbol{H}_0$). A probabilistic mixing of elements 2 and 3 ($\boldsymbol{H}_1$) followed by a deterministic reveal of element 2 at position 2 ($\boldsymbol{H}_2$) correctly resets the revealed entry to 1 but leaves the unrevealed entry at 0.5. Repeating this cycle ($\boldsymbol{H}_3, \boldsymbol{H}_4, \boldsymbol{H}_5$) causes the unrevealed mass to decay exponentially ($0.5 \to 0.25 \to 0.125$), eventually vanishing in finite precision.

Summary. Under the representations analyzed here, the joint representation is numerically unstable for linear RNNs because reveals cannot replenish mass, and the marginal representation can be stable if reveals are distributed across all variables but admits adversarial sequences that force exponential decay. These constructions provide strong evidence that linear RNNs face an inherent difficulty with probabilistic state-tracking: without either nonlinear renormalization in the recurrence or constraints on the reveal distribution, stable belief maintenance appears infeasible.

One might note that finite precision makes the set of representable beliefs finite, so in principle the dynamics can be simulated by a DFA (and hence linearized via one-hot encoding). However, this is impractical: in the joint case, even a binary discretization of the $n!$-dimensional belief vector requires $2^{n!}$ DFA states; in the marginal case, the belief lives in the Birkhoff polytope of dimension $(n-1)^2$, so discretizing each coordinate into k bins yields $k^{(n-1)^2}$ states (e.g., 10^{81} for n=10, k=10).

Linear RNNs suffice for deterministic automata and full state reveals. Finally, we note two regimes where Linear RNNs suffice. First, if the system is a *Deterministic* Finite Automaton (DFA), the belief state remains a one-hot vector corresponding to the true state. Since no uncertainty is introduced, the normalization constant is always 1, and the update is linear. Second, if the system receives *Full State Reveals* (observing the exact current state), the belief is effectively reset to a one-hot vector. This "hard reset" can be implemented by the linear update $h_{t+1} = 0 \cdot h_t + \mathbf{c}$, which zeroes out the history and injects unit mass at the revealed state. Consequently, in a probabilistic setting with partial and full reveals, if full state reveals occur frequently, they prevent the exponential decay of the state norm caused by partial reveals stabilizing the recurrence.

6 Conclusion

We study state-tracking in neural sequence models through the lens of code execution, bridging the gap between abstract automata benchmarks and the next-token prediction paradigm used to train language models.

Summary of contributions. First, we introduced Python REPL traces as a testbed for state-tracking under next-token prediction, showing that linear RNNs with extended eigenvalue spectra (DeltaNet$[-1, 1]$) can learn and generalize reliably with sparse state supervision, while Transformers fail even with dense reveals. Second, we provided evidence for a barrier facing linear RNNs in

realistic code settings: when transitions are probabilistic or partially observable, exact belief tracking requires nonlinear renormalization. We formalized this through the PFSA-SR framework and exhibited adversarial reveal sequences that cause exponential decay in state norms under the natural joint and marginal representations, suggesting that stable belief maintenance is infeasible for linear RNNs under finite precision.

Limitations. Our experiments focus on synthetic permutation groups (S_n) rather than real-world code. While REPL traces are more realistic than prior sequence-to-sequence setups, they still simplify away parsing, control flow, and memory management.

Future work. Our findings suggest that probabilistic state-tracking is a promising benchmark for evaluating nonlinear RNNs (Hochreiter & Schmidhuber, 1997; Beck et al., 2024) and recent parallelization efforts (Lim et al., 2024; Gonzalez et al., 2024; Danieli et al., 2025). Extending this work to real execution traces, e.g. from CRUXEval or system call logs, and investigating hybrid architectures that interleave linear recurrence with periodic nonlinear normalization similar to TTT (Sun et al., 2025) and Titans (Behrouz et al., 2025) is left as future work.

ACKNOWLEDGMENTS

We would like to thank Jan Tönshoff, Heiner Kremer, Fabian Falck, Alicia Curth, Teodora Pandeva, Hari Govind V K, and Andrey Rybalchenko for insightful discussions throughout the project.

REFERENCES

Karl Johan Åström. Optimal control of markov processes with incomplete state information i. *Journal of mathematical analysis and applications*, 10:174–205, 1965.

Maximilian Beck, Korbinian Pöppel, Markus Spanring, Andreas Auer, Oleksandra Prudnikova, Michael Kopp, Günter Klambauer, Johannes Brandstetter, and Sepp Hochreiter. xlstm: Extended long short-term memory. *Advances in Neural Information Processing Systems*, 37:107547–107603, 2024.

Ali Behrouz, Peilin Zhong, and Vahab Mirrokni. Titans: Learning to memorize at test time. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025.

Yoshua Bengio and Paolo Frasconi. An input output hmm architecture. *Advances in neural information processing systems*, 7, 1994.

Satwik Bhattamishra, Kabir Ahuja, and Navin Goyal. On the ability and limitations of transformers to recognize formal languages. *arXiv preprint arXiv:2009.11264*, 2020.

Garrett Birkhoff. Tres observaciones sobre el algebra lineal. *Univ. Nac. Tucuman, Ser. A*, 5:147–154, 1946.

Guy E Blelloch. Prefix sums and their applications. 1990.

Nadav Borenstein, Anej Svete, Robin Chan, Josef Valvoda, Franz Nowak, Isabelle Augenstein, Eleanor Chodroff, and Ryan Cotterell. What languages are easy to language-model? a perspective from learning probabilistic regular languages. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 15115–15134, 2024.

Alexandra Butoi, Ghazal Khalighinejad, Anej Svete, Josef Valvoda, Ryan Cotterell, and Brian DuSell. Training neural networks as recognizers of formal languages. In *The Thirteenth International Conference on Learning Representations*, 2025.

Quentin Carbonneaux, Gal Cohen, Jonas Gehring, Jacob Kahn, Jannik Kossen, Felix Kreuk, Emily McMilin, Michel Meyer, Yuxiang Wei, David Zhang, et al. Cwm: An open-weights llm for research on code generation with world models. *arXiv preprint arXiv:2510.02387*, 2025.

Federico Danieli, Pau Rodriguez, Miguel Sarabia, Xavier Suau, and Luca Zappella. Pararnn: Unlocking parallel training of nonlinear rnns for large language models. *arXiv preprint arXiv:2510.21450*, 2025.

Tri Dao and Albert Gu. Transformers are ssms: Generalized models and efficient algorithms through structured state space duality. In *International Conference on Machine Learning*, pp. 10041–10071. PMLR, 2024.

Grégoire Delétang, Anian Ruoss, Jordi Grau-Moya, Tim Genewein, Li Kevin Wenliang, Elliot Catt, Chris Cundy, Marcus Hutter, Shane Legg, Joel Veness, et al. Neural networks and the chomsky hierarchy. *arXiv preprint arXiv:2207.02098*, 2022.

L. Peter Deutsch and Edmund C. Berkeley. The LISP implementation for the PDP-1 computer. 1964.

Ting-Han Fan, Ta-Chung Chi, and Alexander Rudnicky. Advancing regular language reasoning in linear recurrent neural networks. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 2: Short Papers)*, pp. 45–53, 2024.

Jaden Fiotto-Kaufman, Alexander R. Loftus, Eric Todd, Jannik Brinkmann, Koyena Pal, Dmitrii Troitskii, Michael Ripa, Adam Belfki, Can Rager, Caden Juang, Aaron Mueller, Samuel Marks, Arnab Sen Sharma, Francesca Lucchetti, Nikhil Prakash, Carla E. Brodley, Arjun Guha, Jonathan Bell, Byron C. Wallace, and David Bau. Nnsight and ndif: Democratizing access to open-weight foundation model internals. In *International Conference on Learning Representations*, 2024.

Xavier Gonzalez, Andrew Warrington, Jimmy T Smith, and Scott W Linderman. Towards scalable and stable parallelization of nonlinear rnns. *Advances in Neural Information Processing Systems*, 37:5817–5849, 2024.

Riccardo Grazzi, Julien Siems, Arber Zela, Jörg KH Franke, Frank Hutter, and Massimiliano Pontil. Unlocking State-Tracking in Linear RNNs Through Negative Eigenvalues. In *The Thirteenth International Conference on Learning Representations*, 2025.

Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces. In *First Conference on Language Modeling*, 2024.

Alex Gu, Baptiste Roziere, Hugh James Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida Wang. Cruxeval: A benchmark for code reasoning, understanding and execution. In *International Conference on Machine Learning*, pp. 16568–16621. PMLR, 2024.

Michael Hahn. Theoretical limitations of self-attention in neural sequence models. *Transactions of the Association for Computational Linguistics*, 8:156–171, 2020.

Romain Harang, Jason Naradowsky, Yaswitha Gujju, and Yusuke Miyao. Tracking world states with language models: State-based evaluation using chess. In *ICML 2025 Workshop on Assessing World Models*, 2025.

Harold V Henderson and Shayle R Searle. The vec-permutation matrix, the vec operator and kronecker products: A review. *Linear and multilinear algebra*, 9(4):271–288, 1981.

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

Alston S Householder. Unitary triangularization of a nonsymmetric matrix. *Journal of the ACM (JACM)*, 5(4):339–342, 1958.

Weizhe Hua, Zihang Dai, Hanxiao Liu, and Quoc Le. Transformer quality in linear time. In *International conference on machine learning*, pp. 9099–9117. PMLR, 2022.

JellyFish042. Rwkv-othello. `https://github.com/Jellyfish042/RWKV_Othello`, 2024.

Leslie Pack Kaelbling, Michael L Littman, and Anthony R Cassandra. Planning and acting in partially observable stochastic domains. *Artificial intelligence*, 101(1-2):99–134, 1998.

James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7): 385–394, 1976.

Yi Heng Lim, Qi Zhu, Joshua Selfridge, and Muhammad Firmansyah Kasim. Parallelizing non-linear sequential models over the sequence length. In *The Twelfth International Conference on Learning Representations*, 2024.

Bingbin Liu, Jordan T Ash, Surbhi Goel, Akshay Krishnamurthy, and Cyril Zhang. Transformers learn shortcuts to automata. *arXiv preprint arXiv:2210.10749*, 2022.

Chenxiao Liu, Shuai Lu, Weizhu Chen, Daxin Jiang, Alexey Svyatkovskiy, Shengyu Fu, Neel Sundaresan, and Nan Duan. Code execution with pre-trained language models. In *Findings of the Association for Computational Linguistics: ACL 2023*, pp. 4984–4999, 2023.

Ilya Loshchilov and Frank Hutter. Sgdr: Stochastic gradient descent with warm restarts. In *International Conference on Learning Representations*, 2017a.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *International Conference on Learning Representations*, 2017b.

Eric Martin and Chris Cundy. Parallelizing linear recurrent neural nets over sequence length. *arXiv preprint arXiv:1709.04057*, 2017.

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

William Merrill, Gail Weiss, Yoav Goldberg, Roy Schwartz, Noah A Smith, and Eran Yahav. A formal hierarchy of rnn architectures. In *58th Annual Meeting of the Association for Computational Linguistics, ACL 2020*, pp. 443–459. Association for Computational Linguistics (ACL), 2020.

William Merrill, Jackson Petty, and Ashish Sabharwal. The Illusion of State in State-Space Models. In *Proceedings of the 41st International Conference on Machine Learning*, pp. 35492–35506, 2024.

Sajad Movahedi, Felix Sarnthein, Nicola Muca Cirone, and Antonio Orvieto. Fixed-Point RNNs: From Diagonal to Dense in a Few Iterations. In *First Workshop on Scalable Optimization for Efficient and Adaptive Foundation Models*, 2025.

Kevin P Murphy. Hidden semi-markov models (hsmms). 2002.

Bo Peng, Ruichong Zhang, Daniel Goldstein, Eric Alcaide, Xingjian Du, Haowen Hou, Jiaju Lin, Jiaxing Liu, Janna Lu, William Merrill, et al. Rwkv-7" goose" with expressive dynamic state evolution. *arXiv preprint arXiv:2503.14456*, 2025.

Michael O. Rabin. Probabilistic automata. *Information and Control*, 6(3):230–245, 1963. ISSN 0019-9958. doi: https://doi.org/10.1016/S0019-9958(63)90290-0.

Lawrence R Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *PROCEEDINGS OF THE IEEE*, 77(2):257, 1989.

William E. Roth. On direct product matrices. *Bulletin of the American Mathematical Society*, 40: 461–468, 1934.

Yash Sarrof, Yana Veitsman, and Michael Hahn. The expressive capacity of state space models: A formal language perspective. *Advances in Neural Information Processing Systems*, 37:41202–41241, 2024.

Imanol Schlag, Kazuki Irie, and Jürgen Schmidhuber. Linear transformers are secretly fast weight programmers. In *International conference on machine learning*, pp. 9355–9366. PMLR, 2021a.

Imanol Schlag, Tsendsuren Munkhdalai, and Jürgen Schmidhuber. Learning associative inference using fast weight memory. In *International Conference on Learning Representations*, 2021b.

Mark Schöne, Babak Rahmani, Heiner Kremer, Fabian Falck, Hitesh Ballani, and Jannes Gladrow. Implicit Language Models are RNNs: Balancing Parallelization and Expressivity. In *Forty-second International Conference on Machine Learning*, 2025.

Julien Siems, Timur Carstensen, Arber Zela, Frank Hutter, Massimiliano Pontil, and Riccardo Grazzi. Deltaproduct: Improving state-tracking in linear RNNs via householder products. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025.

Richard Sinkhorn and Paul Knopp. Concerning nonnegative matrices and doubly stochastic matrices. *Pacific Journal of Mathematics*, 21(2):343–348, 1967.

Lena Strobl, William Merrill, Gail Weiss, David Chiang, and Dana Angluin. What formal languages can transformers express? a survey. *Transactions of the Association for Computational Linguistics*, 12:543–561, 2024.

Yu Sun, Xinhao Li, Karan Dalal, Jiarui Xu, Arjun Vikram, Genghan Zhang, Yann Dubois, Xinlei Chen, Xiaolong Wang, Sanmi Koyejo, et al. Learning to (learn at test time): Rnns with expressive hidden states. In *Forty-second International Conference on Machine Learning*, 2025.

Yutao Sun, Li Dong, Shaohan Huang, Shuming Ma, Yuqing Xia, Jilong Xue, Jianyong Wang, and Furu Wei. Retentive network: A successor to transformer for large language models. *arXiv preprint arXiv:2307.08621*, 2023.

Anej Svete and Ryan Cotterell. Recurrent neural language models as probabilistic finite-state automata. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 8069–8086, 2023.

Kimi Team, Yu Zhang, Zongyu Lin, Xingcheng Yao, Jiaxi Hu, Fanqing Meng, Chengyin Liu, Xin Men, Songlin Yang, Zhiyuan Li, et al. Kimi linear: An expressive, efficient attention architecture. *arXiv preprint arXiv:2510.26692*, 2025.

Qwen Team. Qwen3 technical report, 2025.

Aleksandar Terzic, Nicolas Menet, Michael Hersche, Thomas Hofmann, and Abbas Rahimi. Structured sparse transition matrices to enable state tracking in state-space models. In *Annual Conference on Neural Information Processing Systems*, 2025.

Shubham Toshniwal, Sam Wiseman, Karen Livescu, and Kevin Gimpel. Chess as a testbed for language model state tracking. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pp. 11385–11393, 2022.

Keyon Vafa, Justin Y Chen, Ashesh Rambachan, Jon Kleinberg, and Sendhil Mullainathan. Evaluating the world model implicit in a generative model. *Advances in Neural Information Processing Systems*, 37:26941–26975, 2024.

Keyon Vafa, Peter G Chang, Ashesh Rambachan, and Sendhil Mullainathan. What has a foundation model found? using inductive bias to probe for world models. In *International Conference on Machine Learning*, 2025.

Guido Van Rossum, Fred L Drake, et al. *Python reference manual*, volume 111. Centrum voor Wiskunde en Informatica Amsterdam, 1995.

John Von Neumann. A certain zero-sum two-person game equivalent to the optimal assignment problem, contributions to the theory of games, vol. 2. *Ann. Math. Studies,(28)*, 1953.

Benjamin Walker, Lingyi Yang, Nicola Muca Cirone, Cristopher Salvi, and Terry Lyons. Structured linear cdes: Maximally expressive and parallel-in-time sequence models. *arXiv preprint arXiv:2505.17761*, 2025.

Songlin Yang and Yu Zhang. Fla: A triton-based library for hardware-efficient implementations of linear attention mechanism, January 2024. URL `https://github.com/fla-org/flash-linear-attention`.

Songlin Yang, Bailin Wang, Yikang Shen, Rameswar Panda, and Yoon Kim. Gated linear attention transformers with hardware-efficient training. In *Forty-first International Conference on Machine Learning*, 2024a.

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

Songlin Yang, Jan Kautz, and Ali Hatamizadeh. Gated delta networks: Improving mamba2 with delta rule. In *The Thirteenth International Conference on Learning Representations*, 2025.

A From Permutation Groups to Variable Tracking in Code

A.1 Experimental Details

Model & Training. We train a DeltaNet and Transformer (≈265M parameters) using the implementation from flash-linear-attention (Yang & Zhang, 2024). The architecture consists of 18 layers, hidden dimension $d = 512$, 8 heads ($d_{head} = 128$), MLP expansion factor 4, and SwiGLU activations. Optimization is performed using AdamW (Loshchilov & Hutter, 2017b) with a peak learning rate of 5×10^{-4}, a cosine decay schedule (Loshchilov & Hutter, 2017a) (minimum LR ratio 0.2), and 5% warmup. We use a per-device batch size of 3 with 12 gradient accumulation steps in BF16 mixed precision. We trained on single Nvidia A100s for each model.

Curriculum Learning. To ensure stability, we use a four-stage curriculum of 15,000 samples each. We use full permutations on 5 variables (S_5), progressively increasing the trace length (L) and reveal spacing (S):

$$(L, S) \in \{(8, 1), (16, 2), (32, 4), (64, 8)\}$$

A.2 INTERPRETABILITY

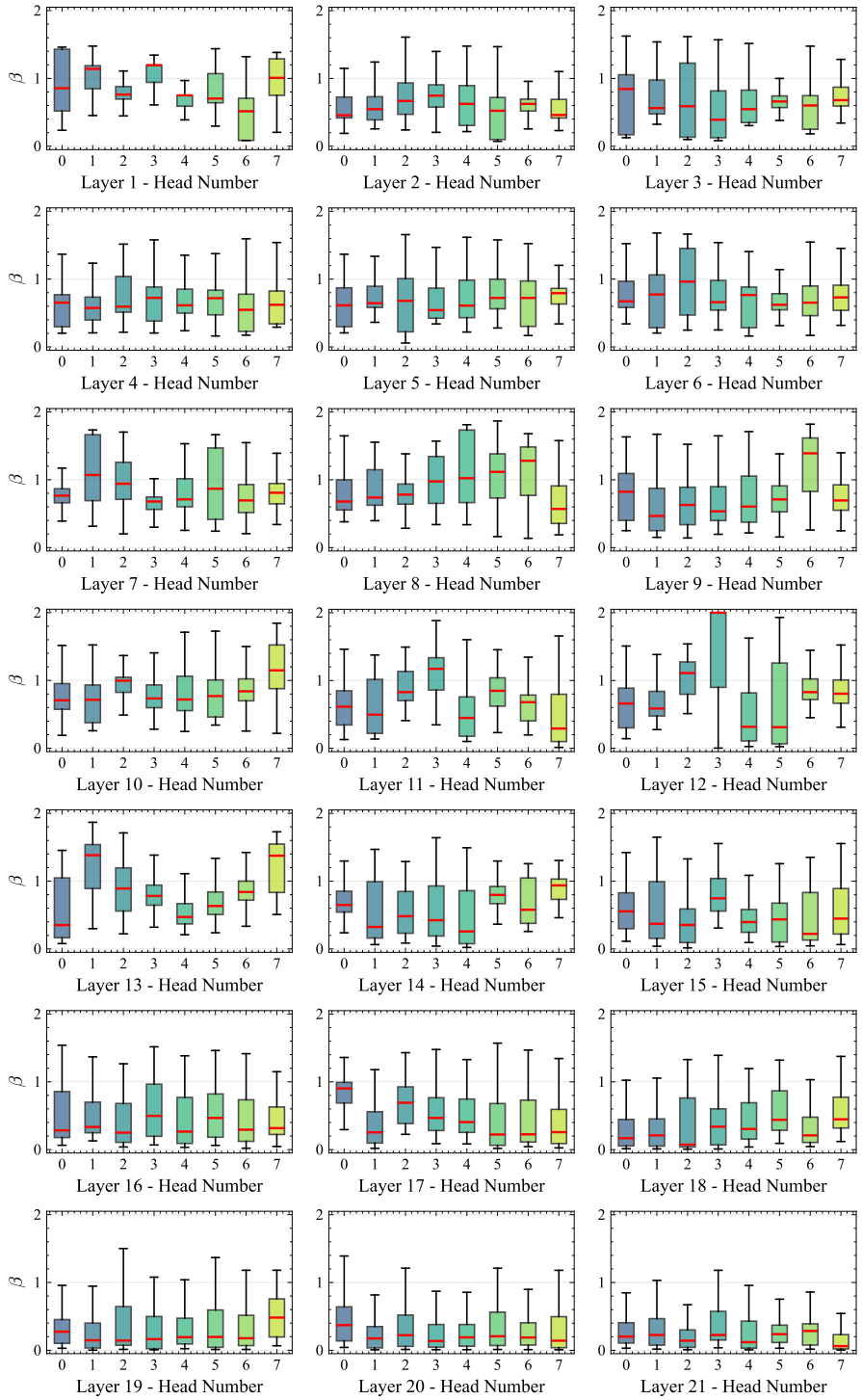


Figure 11: β distribution per layer per head recorded during a forward pass of a REPL trace. Layer 12, Head 3 stands out as the state-tracking head.

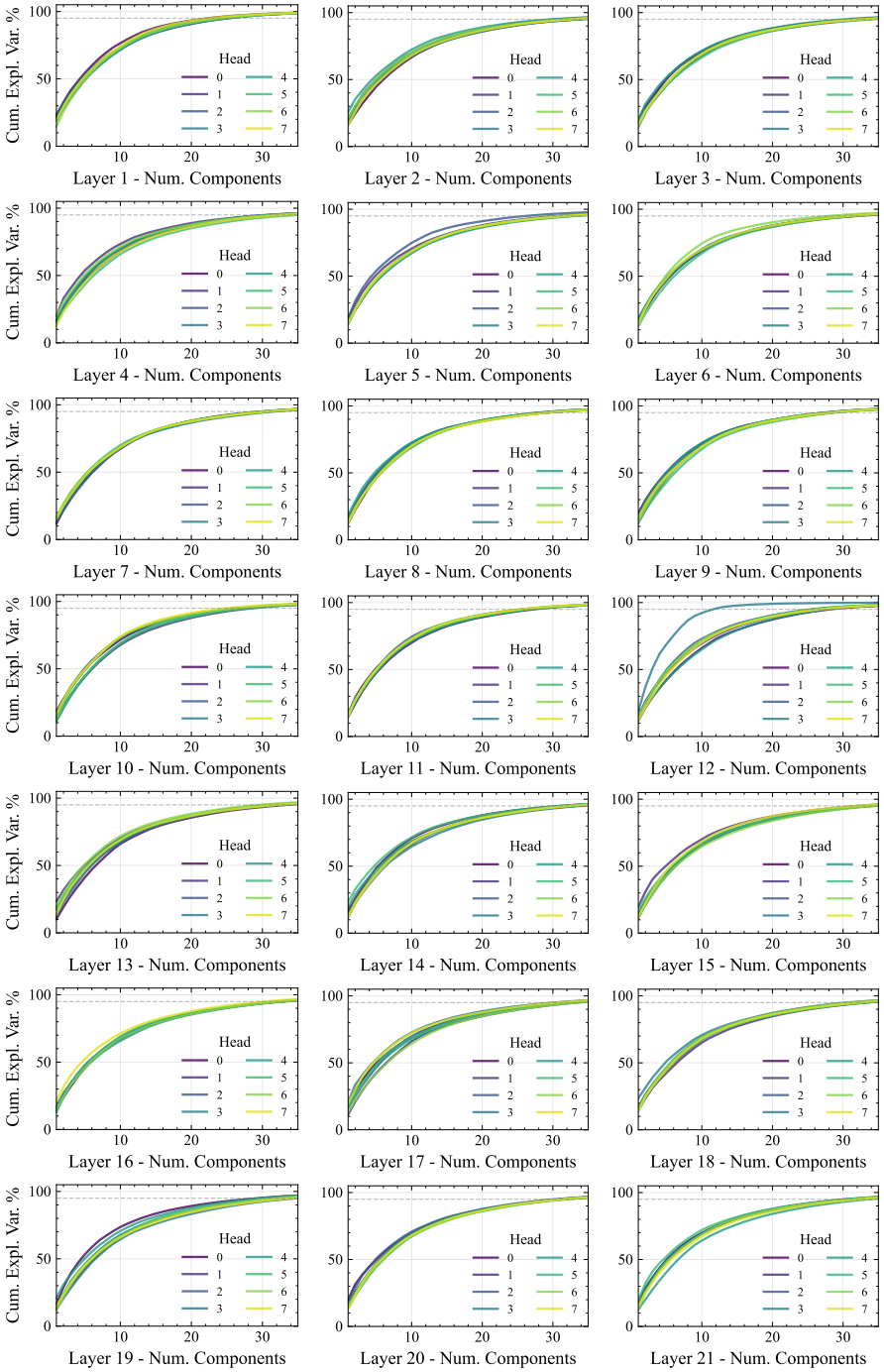


Figure 12: Cumulative explained variance of the keys per layer per head recorded during a forward pass of a REPL trace. Layer 12, Head 3 stands out as the state-tracking head.

B Probabilistic State-Tracking

B.1 Example: Linear RNN implementing probabilistic finite-state automaton tracking the join

This section details the explicit arithmetic of a stochastic update on the permutation group S_3 and discusses the theoretical implications of norm decay in Linear RNNs.

Scenario Setup and Initialization ($t = 0$). We model the system state $\boldsymbol{H}_t \in \mathbb{R}^6$ over the permutation group S_3. The basis vectors (universes) correspond to the six possible permutations of $[1, 2, 3]$: Identity $[1, 2, 3]$ (Index 1), Swap 1-2 $[2, 1, 3]$ (Index 2), Swap 1-3 $[3, 2, 1]$ (Index 3), Swap 2-3 $[1, 3, 2]$ (Index 4), Cycle Left $[2, 3, 1]$ (Index 5), and Cycle Right $[3, 1, 2]$ (Index 6). We begin at Step 0 with perfect certainty at the Identity configuration:

$$\boldsymbol{H}_0 = [1 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0]^T \quad \text{(Universe 1: } [1, 2, 3]\text{)}.$$

Step 1: The Stochastic Action ($t = 1$). The system receives the input "Try to Swap 1-2" with a noise profile defined as a 50% chance of the intended Swap 1-2 and a 50% chance of an accidental Swap 1-3. To represent this ambiguity, we construct a "fuzzy" transition matrix $\boldsymbol{A}_{fuzzy}$ by averaging the permutation matrices of the two outcomes: $\boldsymbol{A}_{fuzzy} = 0.5 \cdot \boldsymbol{A}_{\text{swap12}} + 0.5 \cdot \boldsymbol{A}_{\text{swap13}}$. Specifically, $\boldsymbol{A}_{\text{swap12}}$ maps Identity to Index 2, while $\boldsymbol{A}_{\text{swap13}}$ maps Identity to Index 3. The resulting update is:

$$\boldsymbol{H}_1 = \boldsymbol{A}_{fuzzy}\boldsymbol{H}_0 = \begin{bmatrix} 0 & 0.5 & 0.5 & 0 & 0 & 0 \\ \mathbf{0.5} & 0 & 0 & 0 & 0.5 & 0 \\ \mathbf{0.5} & 0 & 0 & 0 & 0 & 0.5 \\ 0 & 0 & 0 & 0 & 0.5 & 0.5 \\ 0 & 0.5 & 0 & 0.5 & 0 & 0 \\ 0 & 0 & 0.5 & 0.5 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ \mathbf{0.5} \\ \mathbf{0.5} \\ 0 \\ 0 \\ 0 \end{bmatrix}.$$

The state is now diffused; probability mass is split between Universe 2 ($[2, 1, 3]$) and Universe 3 ($[3, 2, 1]$).

Step 2: The Observation ($t = 2$). We subsequently receive the observation "Position 1 contains Object 3". To process this, we construct a diagonal observation matrix $\boldsymbol{A}_{obs}$ that acts as a filter. We check every basis vector: Index 1 ($[1, 2, 3]$) is false; Index 2 ($[2, 1, 3]$) is false; Index 3 ($[3, 2, 1]$) is true (keep); Index 6 ($[3, 1, 2]$) is true (keep). All others are zeroed out. Applying this filter to the smeared state $\boldsymbol{H}_1$:

$$\boldsymbol{H}_2 = \boldsymbol{A}_{obs}\boldsymbol{H}_1 = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \mathbf{1} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \mathbf{1} \end{bmatrix} \begin{bmatrix} 0 \\ 0.5 \\ 0.5 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \mathbf{0.5} \\ 0 \\ 0 \\ 0 \end{bmatrix}.$$

The model has correctly identified that we are in Universe 3 ($[3, 2, 1]$). The ambiguity created in Step 1 was resolved because Universe 2 is inconsistent with the observation. The final vector magnitude is 0.5, representing the joint probability of the path: $P(\text{Path}) = P(\text{Slip}) \times P(\text{Consistent}) = 0.5 \times 1.0 = 0.5$.

The Mechanics of Information Decay. A critical limitation of Linear RNNs when tracking probabilistic states is the phenomenon of *norm decay*. Unlike non-linear models (e.g., Transformers with Softmax) which re-normalize their internal state, a Linear RNN performs purely multiplicative updates ($\boldsymbol{H}_t = \boldsymbol{A}\boldsymbol{H}_{t-1}$). In a stochastic setting, transition matrices often have eigenvalues $|\lambda| < 1$ due to diffusion or filtering. Consequently, $\|\boldsymbol{H}_t\| \approx \lambda^t \|\boldsymbol{H}_0\|$. For example, in a "Noisy Identity" transition where the system retains state with $p = 0.9$, the norm decays to ≈ 0.00002 after 100 steps. While floating-point standards allow for small numbers, the signal eventually vanishes relative to numerical noise.

The Role of B: The Gated Reset. The matrix $\boldsymbol{B}$, which is typically zero during standard tracking, serves as a solution to decay via a *Gated Reset Mechanism*. Let $\boldsymbol{x}_t$ be a binary indicator where $\boldsymbol{x}_t = 1$ indicates a "Reset". We parameterize the weights as $\boldsymbol{A}(\boldsymbol{x}_t) = (1 - \boldsymbol{x}_t) \cdot \boldsymbol{A}_{\text{step}}$ and $\boldsymbol{B}(\boldsymbol{x}_t) = \boldsymbol{x}_t \cdot \boldsymbol{H}_{\text{prior}}$. When a reset is triggered ($\boldsymbol{x}_t = 1$), the history is annihilated ($\boldsymbol{A}(\boldsymbol{x}_t) = \mathbf{0}$) and the bias term injects the prior ($\boldsymbol{B}(\boldsymbol{x}_t) = \boldsymbol{H}_{\text{prior}}$). The update becomes:

$$\boldsymbol{H}_t = \mathbf{0} \cdot \boldsymbol{H}_{t-1} + \boldsymbol{H}_{\text{prior}} = [1/6, \ldots, 1/6]^T.$$

This "re-inflates" the state vector to full magnitude, readying the system to track a new sequence.