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 model like Transformers and RNNs (linear and non-linear). However, current experiments use a sequence-to-sequence setup: learning to map actions (permutations) to states, that does not translate to 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 adversarial sequences exist where linear RNNs cannot guarantee stable probabilistic state-tracking.

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 sequence lengths. 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), 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). 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).

To understand how state-tracking can be learned by language models, we study learning state-tracking from next-token prediction. Current state-tracking benchmarks are sequence-to-sequence where sequences of actions (permutations) are mapped to sequences of states. This setting deviates significantly from next-token prediction used to train language models. To partially bridge this gap, we construct Python REPL traces interleaving variable permutations and (partial) state-reveals where state-tracking can be learned via a next-token prediction objective. We find that linear RNNs excel with sparse supervision while transformers require very frequent reveals. We then formalize why tracking variable states in real code is harder: actions may not be observable, requiring models to handle uncertainty over possible system states. In this case, a model cannot deterministically compute the next state but must maintain a distribution over possible states. While probabilistic state-tracking has been studied for nonlinear RNNs (Svete & Cotterell, 2023), we show that linear

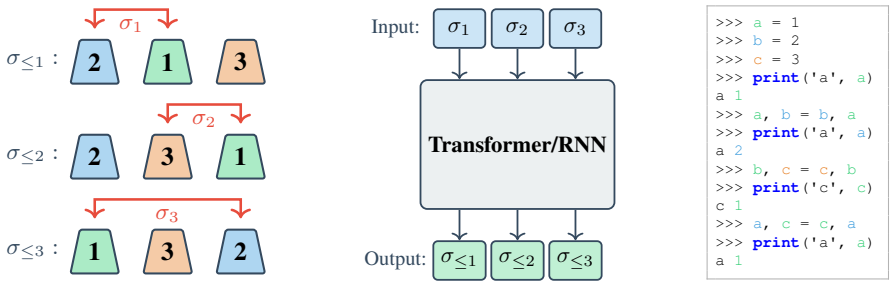

Figure 1: Three representations of the permutation tracking task. **Left:** The shell game analogy showing cups being swapped to track object positions. **Center:** The sequence-to-sequence modeling approach from Merrill et al. (2024), where a Transformer/RNN processes input permutations σ_i and outputs 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.

RNNs cannot robustly track probabilistic beliefs under finite precision without nonlinear renormalization to restore robustness.

2 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

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

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 was 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).

3 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 1 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.

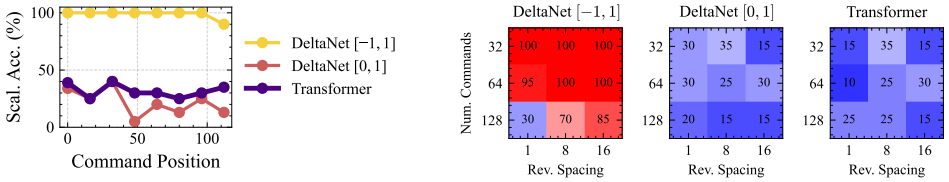

Figure 3: **Left:** Per reveal position accuracy averaged across 5 seeds, reveal spacing 16 and num commands 128. Only DeltaNet $[-1, 1]$ manages to reliably learn to perform state tracking. **Right:** Final reveal position accuracy when increasing the reveal spacing and num. commands beyond training regime, 8 and 64 respectively.

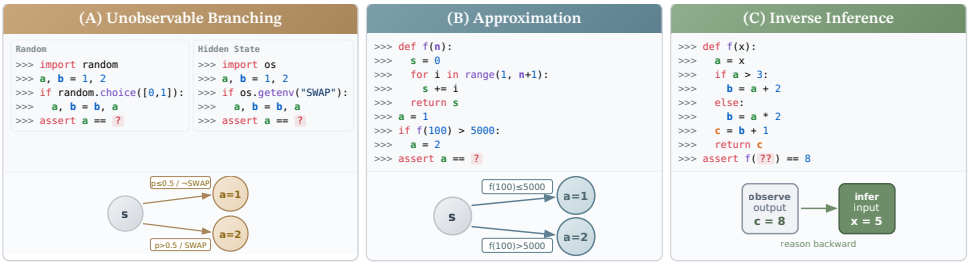

Figure 4: Sources of probabilistic state transitions in code execution: (A) unobservable branching from randomness or hidden state, (B) approximation when skipping expensive computations, (C) inverse inference when recovering inputs from outputs.

3.1 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-Base (Team, 2025) model using standard next-token prediction (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 2, 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.

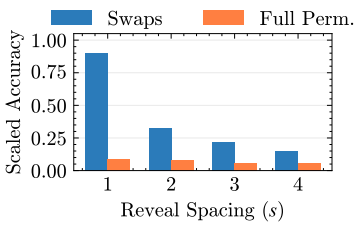

Figure 2: Transformers require dense state supervision to solve REPL traces. Accuracy drops as reveal spacing increases, with full permutations failing under any sparsity.

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 3 (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. See Section B.2 for a mechanistic interpretability study of DeltaNet$[-1, 1]$.

4 From Deterministic to Probabilistic State-Tracking

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 (§3.1). 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 affords such transparency. Figure 4

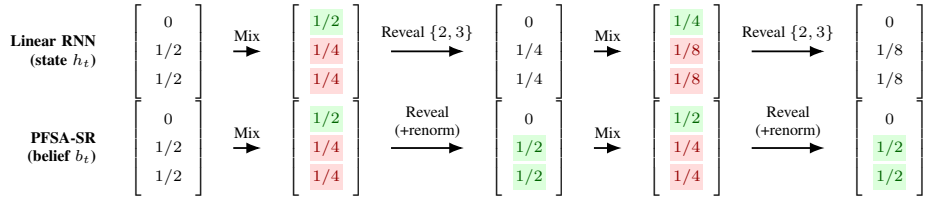

Figure 5: Survival-product accumulation. A mixing step moves half the mass into an absorbing state; the reveal keeps $\{2, 3\}$. The linear RNN's state norm decays geometrically ($1/2 \to 1/4 \to 1/8 \to \cdots$), eventually underflowing. PFSA-SR renormalizes after each reveal, keeping beliefs bounded away from zero.

illustrates three representative sources of *transition uncertainty*. **(A) Unobservable branching** occurs when code branches on conditions inaccessible to the model—explicit randomness, hidden environment state, or external API calls. **(B) Approximation** arises when models skip expensive operations like loops, trading exact computation for efficiency but introducing uncertainty about intermediate states. **(C) Inverse inference** tasks, such as CRUXEval-Input (Gu et al., 2024), require recovering inputs from outputs, which often admits multiple valid solutions. In all cases, a model must maintain a *distribution* over states rather than tracking a single deterministic state.

4.1 Probabilistic Finite-State Automata with State Reveals

We formalize this setting as a *Probabilistic Finite-State Automaton with State Reveals* (PFSA-SR): a tuple $(\mathcal{Q}, \Sigma, \delta, \rho, q_0)$ where $\mathcal{Q}$ is a finite set of states, $\delta : \mathcal{Q} \times \Sigma \to \mathrm{Dist}(\mathcal{Q})$ is a probabilistic transition kernel, and $\rho : \Sigma \to 2^{\mathcal{Q}}$ is a reveal function mapping inputs to observable state subsets. At each time-step, the environment provides a symbol σ_t such that $q_t \in \rho(\sigma_t)$ (a partial observation), then the next state is sampled: $q_{t+1} \sim \delta(q_t, \sigma_t)$. Since the true state is hidden, we track a *belief* b_t—the conditional distribution $p(q_t \mid \sigma_1, \ldots, \sigma_t)$—represented as a vector in the simplex Δ_m. Let $Z_\sigma \in \mathbb{R}^{m \times m}$ be a diagonal reveal matrix with $(Z_\sigma)_{ii} = 1$ if $i \in \rho(\sigma)$, and let T_σ be the column-stochastic transition matrix. The belief update proceeds in two stages: first zero out states inconsistent with the reveal, then propagate through the transition: $b_{t+1} = f(T_{\sigma_t} Z_{\sigma_t} b_t), \quad \text{where } f(x) = x/\|x\|_1$. PFSA-SR generalizes DFAs and weighted automata while differing from IO-HMMs (Bengio & Frasconi, 1994) and POMDPs (Åström, 1965; Kaelbling et al., 1998) in its use of hard, support-pruning reveals that eliminate states outright rather than soft likelihood-weighted observations. PFSA-SR reduces to a DFA when δ is deterministic and reveals are uninformative ($\rho(\sigma) = \mathcal{Q}$). Compared to HMMs and POMDPs, the key difference is the emphasis on *hard, support-pruning reveals* that eliminate states rather than soft likelihood-weighted observations. See Section C for the specific case of permutation composition.

4.2 Linear RNNs are Numerically Unstable

The normalization $f(\cdot)$ is essential for stable belief tracking but breaks linearity. Consider a linear recurrence that defers normalization to the decoder: $h_{t+1} = T_{\sigma_t} Z_{\sigma_t} h_t; b_t = h_t/\|h_t\|_1$. Define the *survival probability* $s_t := \|Z_{\sigma_t} b_t\|_1 \in (0, 1]$ as the belief mass consistent with reveal $\rho(\sigma_t)$. Then $\|h_t\|_1 = \prod_{k=0}^{t-1} s_k$. If $s_k < 1$ repeatedly, $\|h_t\|_1$ shrinks exponentially, and the linear recurrence must preserve relative ratios across components despite exponentially vanishing magnitude. Crucially, this occurs even when each per-step survival is bounded away from zero, because the recurrence accumulates the *product* of all survival probabilities. This parallels the well-known underflow problem in HMM forward messages (Rabiner, 1989; Murphy, 2002), motivating per-step scaling or log-domain implementations—both of which break linearity. Figure 5 illustrates: a linear RNN's state decays geometrically while PFSA-SR's renormalized belief remains stable.

When linear RNNs suffice. Linear RNNs handle two regimes: (1) *deterministic automata*, where belief remains one-hot and normalization is trivially 1, and (2) *full state reveals*, which reset belief to one-hot via $h_{t+1} = 0 \cdot h_t + \mathbf{c}$. Frequent full reveals prevent exponential decay by periodically stabilizing the recurrence.

5 Conclusion

We studied state-tracking through the lens of code execution, introducing REPL traces as a testbed for next-token prediction. Linear RNNs with extended eigenvalue spectra (DeltaNet$[-1, 1]$) learn

and generalize reliably with sparse supervision, while Transformers fail even with dense reveals. However, realistic code involves probabilistic transitions from hidden branching, approximation, and inverse inference. We formalized this via PFSA-SR and showed that linear RNNs cannot maintain stable beliefs: partial reveals cause exponential norm decay. This motivates probabilistic state-tracking as a benchmark for evaluating nonlinear RNNs and hybrid architectures.

REFERENCES

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

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.

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.

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.

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.

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

Yongyi Yang and Jianyang Gao. mhc-lite: You don't need 20 sinkhorn-knopp iterations. *arXiv preprint arXiv:2601.05732*, 2026.

A Related Work

Deterministic state tracking Prior work on state tracking has largely focused on *deterministic* settings, studying regular languages and finite-state automaton (FSA) emulation with length generalization as the primary stress test (Hahn, 2020; Bhattamishra et al., 2020; Delétang et al., 2022; Liu et al., 2022). Empirical studies systematize these tasks via the Chomsky hierarchy and reveal sharp, architecture-dependent generalization gaps (Delétang et al., 2022), while complementary analyses characterize when modern sequence models can or cannot implement finite-state computation under bounded depth and finite precision (Merrill et al., 2024). Recent work further examines how selective or structured SSM parameterizations affect regular-language expressivity and length extrapolation (Terzic et al., 2025). Despite this progress, evidence that improved FSA emulation yields broad practical gains remains mixed: architectural modifications that reliably improve synthetic state-tracking benchmarks often translate to only modest gains on standard language-model evaluations (Grazzi et al., 2025; Siems et al., 2025; JellyFish042, 2024). Moreover, widely used generators, such as group-word problems (Liu et al., 2022), differ substantially from tracking variable states during code execution, where supervision is sparse and state evolution is often stochastic. Our work addresses this gap by studying partially observed and probabilistic state tracking under code-like supervision.

Probabilistic automata and partial observability. Classical probabilistic finite-state models, including probabilistic or weighted automata, IO-HMMs, and POMDPs, formalize uncertainty through belief-state updates under partial observations (Rabin, 1963; Bengio & Frasconi, 1994; Åström, 1965; Kaelbling et al., 1998). Recent NLP work makes this connection explicit by characterizing recurrent sequence models as restricted families of probabilistic finite-state systems (Svete & Cotterell, 2023; Butoi et al., 2025; Borenstein et al., 2024). Our PFSA-SR aligns with this prior work but differs in the observation model: we focus on partial observability induced by *hard, support-pruning reveals* (e.g., asserts, prints, return codes) that deterministically eliminate inconsistent latent trajectories. This perspective emphasizes belief compression under sparse supervision and enables a direct comparison between linear and nonlinear recurrent dynamics for belief-state updates.

Transformers and linear RNN expressivity A line of work analyzes the expressivity limits of Transformers on algorithmic state-tracking tasks, including FSA emulation and periodic languages, identifying failure modes such as 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, substantial effort has focused on enabling state tracking in efficient recurrent and SSM-style architectures by modifying the transition operator, yielding a range of structured parameterizations that trade computational efficiency for expressive power (Schlag et al., 2021b; Yang et al., 2024b; Fan et al., 2024; Walker et al., 2025; Peng et al., 2025). Notably, allowing richer spectra or transition factorizations can unlock qualitatively new state-tracking behaviors in linear RNNs without changing asymptotic cost (Grazzi et al., 2025; Siems et al., 2025). These works primarily address deterministic state transitions and largely abstract away belief evolution under uncertainty or sparse supervision.

B Motivation

B.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

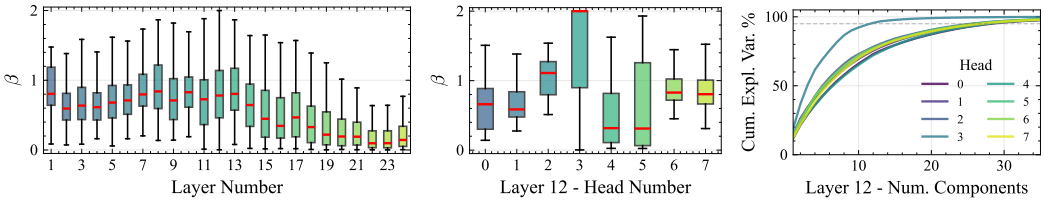

Figure 6: Interpretability Analysis: Distribution of β for a sequence of 512 commands with reveal spacing 4. **Left:** β values aggregated per layer across heads. **Middle:** β per head for layer 12; head 3 stands out as the only head in the network to consistently set its β values to 2, making it the state-tracking head. **Right:** Cumulative explained variance of the PCA of the keys. Head 3 again stands out, being explainable with much fewer components.

spacing (S):

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

B.2 Interpretability

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 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 6 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. 6, 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 this head has learned to be the state-tracking head of the network. Results for the remaining layers are in Section B.2.

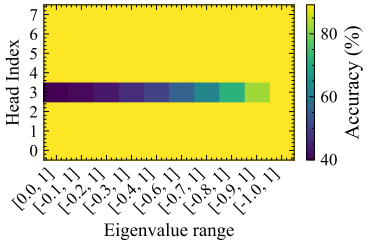

Figure 7: Intervention analysis: Scaling the β range per head in layer 12. State prediction accuracy degrades only when head 3 is scaled, confirming it as the state-tracking head.

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. 7. 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.

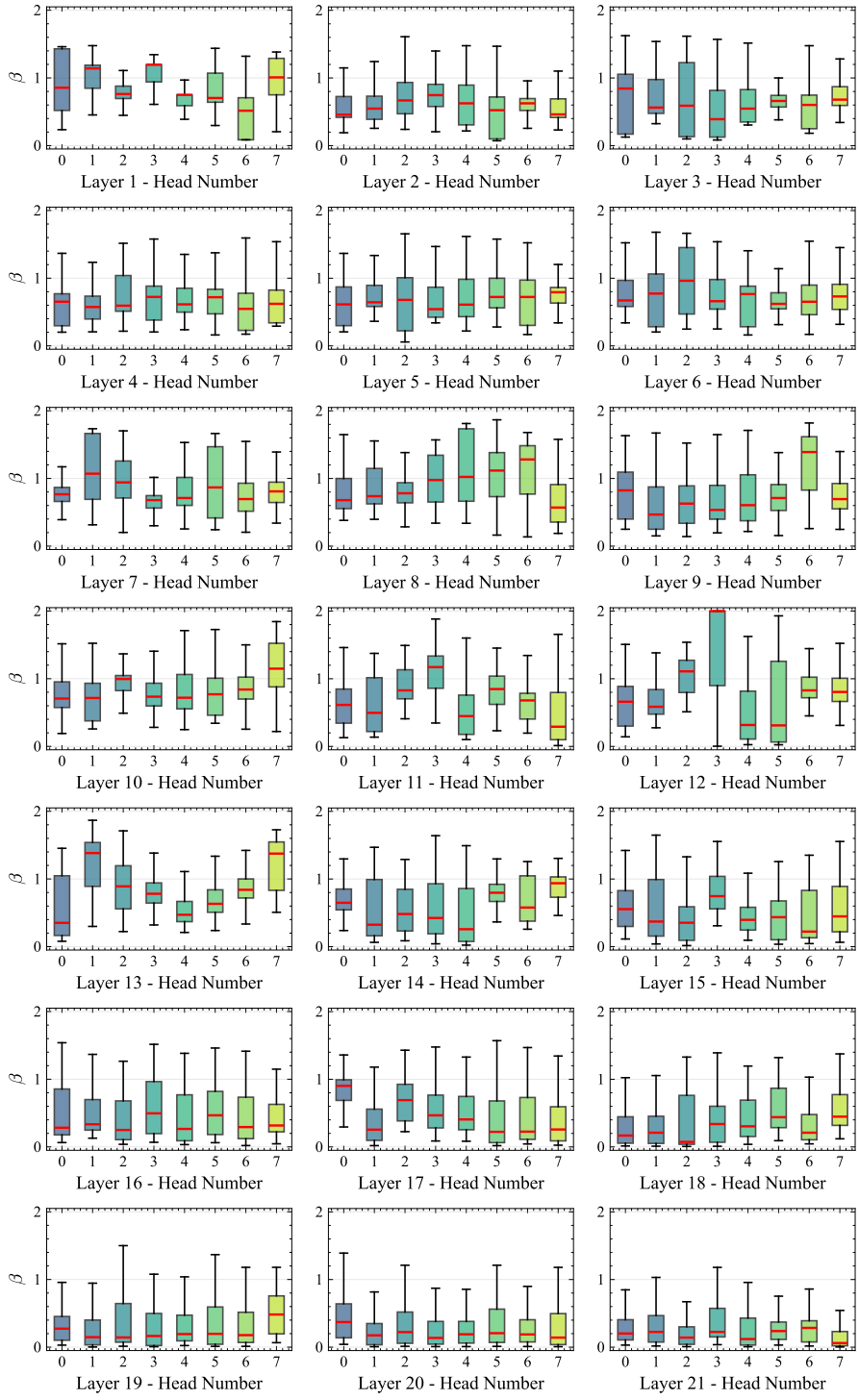

Figure 8: β distribution per layer per head recorded during a forward pass of a REPL trace. Layer 12, Head 3 stands out as the state-tracking head.

C Probabilistic Permutation State-Tracking

To ground the PFSA-SR in a concrete but difficult problem, we analyze permutation tracking under uncertainty. In particular, we want to track the configurations of list of n distinct elements after a

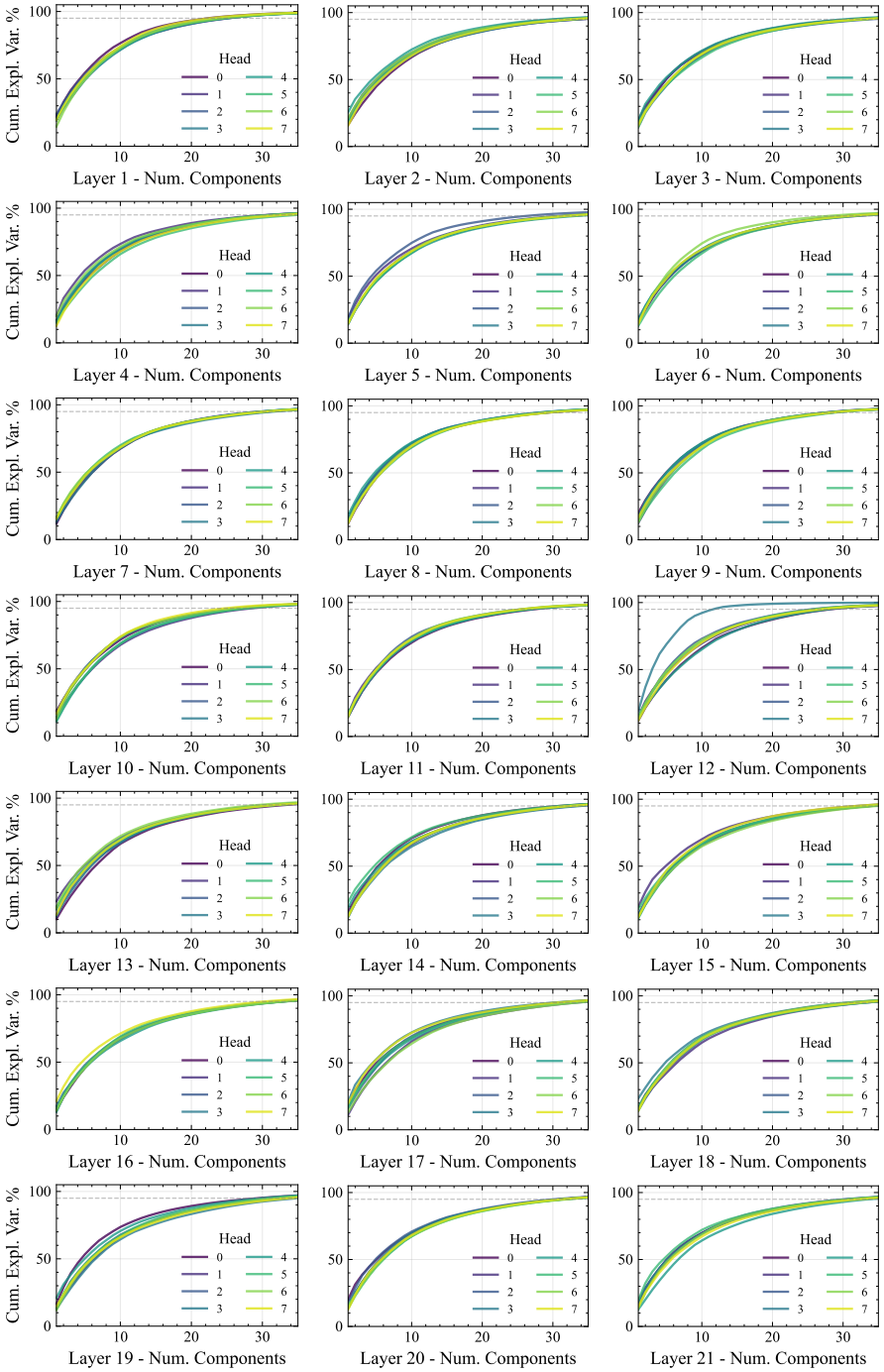

Figure 9: Cumulative explained variance of the keys per layer per head recorded during a forward pass of a REPL trace. Layer 12, Head 3 stands out as the state-tracking head.

sequence of permutations, each an element of group S_n. Note that the corresponding automaton has $n!$ states (one for each configuration of the list) and thus applying the belief update naively would require to store and update a very large $n!$-dimensional belief vector, while in the deterministic case we can track the sate using a linear RNN with an $n-1$-dimensional state, because each permutation in S_n can be represented by a $n-1 \times n-1$ permutation matrix. We provide an example of state-tracking the full permutation automaton in Section C.1.

As a more tractable alternative, instead of tracking the *joint probability* of all positions in the list, we focus on tracking the *marginal probability* of each position. Since we are permuting n elements this means that we can represent this belief probability as an $n \times n$ matrix where each element ij represent the probability that position i contains element j. In this scenario each transitions is a convex combiniation of the $n \times n$ permutation matrices. Note that this guarantees that both the belief and the transition matrices are doubly stochastic. To adhere to the REPL traces described above we focus on reveals of the form "position 2 contains element 5", which can be modeled by a bilinear operation.

C.1 Example: Linear RNN implementing probabilistic finite-state automaton tracking the joint

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:

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

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 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.

C.2 Tracking the Marginal Distribution via the Panini Recurrence

The marginal state: a doubly stochastic matrix Consider the list of elements which starts from $(1, 2, \ldots, n)$. We define the state of our system as an $n \times n$ matrix M, where entry M_{ij} contains the probability that the i-th list position contains element j. The starting state would be $M = I$. To align with standard state-transition notation (left multiplication), rows (i) represent positions in memory and columns (j) represent elements/variables (both ranging over $1, \ldots, n$). This matrix must satisfy two constraints derived from the definition of a permutation: (1) each position contains exactly one element, $\sum_j M_{ij} = 1$ for all i; and (2) each element exists at exactly one position, $\sum_i M_{ij} = 1$ 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.

In order to efficiently model the marginal state evolution through a series of transformations we propose a new type of linear recurrence with transition matrices on both sides. We refer to it as the **Panini** recurrence since it sandwiches the matrix-valued hidden state between two state-transition matrices. In this case the matrix valued hidden state $\boldsymbol{H}_i$ is the marginal state matrix.

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

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), it's easy to see that this remains an affine update on the vectorized state: $\mathrm{vec}(\boldsymbol{H}_t) = (\boldsymbol{A}_r(x_t)^\top \otimes \boldsymbol{A}_l(x_t))\mathrm{vec}(\boldsymbol{H}_{t-1}) + \mathrm{vec}(\boldsymbol{B}(\boldsymbol{x}_t))$. For our application we parameterize the recurrence as

$$\boldsymbol{A}_l(\boldsymbol{x}_t) = \boldsymbol{P}_s(\boldsymbol{x}_t)\boldsymbol{D}_l(\boldsymbol{x}_t), \quad \boldsymbol{A}_r(\boldsymbol{x}_t) = \boldsymbol{D}_r(\boldsymbol{x}_t),$$

Were $\boldsymbol{D}_l(\boldsymbol{x}_t), \boldsymbol{D}_r(\boldsymbol{x}_t)$ are diagonal matrices. We note that a similar recurrence structure (where $\boldsymbol{P}_s(\boldsymbol{x}_t)$ is a generalized householder matrix) is present in the open-source implementation of Kimi Delta Attention (KDA) (Team et al., 2025) (code) (referred to as key and value gates), although the recurrence was not explicitly described in the paper.

We will explain the parameterization of the recurrence in detail through a description of how probabilistic transitions and state reveals which correspond to variable transformations and print statements in code respectively are implemented.

1. Probabilistic Transitions (Mixing). When the input x_t encodes a shuffle (e.g., "swap contents of Position a and Position b"), we permute the rows of $\boldsymbol{H}_i$. If the shuffle is probabilistic, we apply a linear mixture of permutation matrices to the left:

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

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

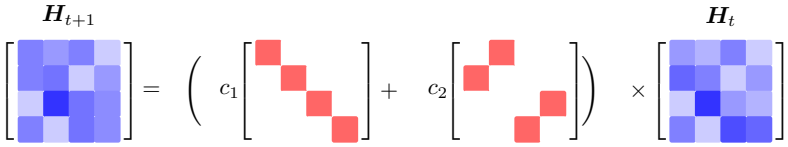

Figure 10: Visualization of Probabilistic Mixing: The state is multiplied by a convex combination of permutation matrices. In this example, the identity and the matrix swapping the first two and the last two positions.

2. State Reveals. If $\boldsymbol{x}_t$ encodes the constraint *"Position i contains Element j."*, this means enforcing the following conditions. (1) The entry (i, j) is confirmed (set to 1). (2) Position i cannot contain any other element (zero out all elements in row i, except the j-th). (3) Element j cannot be in any other position. (zero out all elements in column j, except the i-th). This translates into the following parameterization of the recurrence:

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

Where $\boldsymbol{e}_i$ is a standard basis vector of $\mathcal{R}^n$. The term $\boldsymbol{D}_l\boldsymbol{H}_t\boldsymbol{D}_r$ preserves the probability mass of all configurations *compatible* with the observation (outside the cross), while the input $\boldsymbol{B}(\boldsymbol{x}_t)$ injects the certainty of the observed fact.

In summary, we updated the state matrix M by setting $M_{ij} = 1$ and by setting to zero all other elements in row i and column j. However, note after this update M might not be doubly stochastic anymore and needs to be normalized to be a valid marginal distribution. However, since this normalization is not linear we do not include it as part of the recurrence.

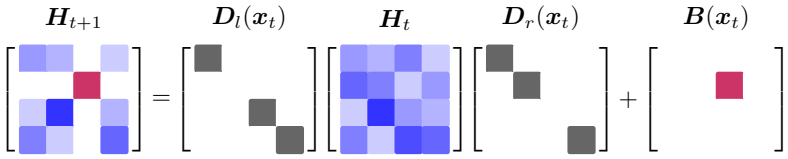

Figure 11: Visualization of State Reveal: Diagonal masks zero out the conflicting row and column, while the input $\boldsymbol{B}_t$ injects the observed certainty. This example shows the reveal *"Position 2 contains element 3"*

3. Projection (Sinkhorn-Knopp algorithm (Sinkhorn & Knopp, 1967)). While the linear recurrence updates efficiently track the mixing and masking of state information, the resulting matrix $\boldsymbol{H}_t$ may drift from the manifold of doubly stochastic matrices. For instance, the "State Reveal" update operation removes probability mass from conflicting entries, possibly leaving some row and column sums strictly less than 1.

To recover a valid marginal distribution, we define our decoder $\mathrm{dec}(\boldsymbol{H}_t, \boldsymbol{x}_t)$ using the Sinkhorn-Knopp algorithm. This algorithm iteratively normalizes rows and columns to project the matrix back onto the Birkhoff polytope:

$$\boldsymbol{S}^{(0)} = \boldsymbol{H}_t, \qquad \boldsymbol{S}^{(k+1)} = \mathcal{T}_c\left(\mathcal{T}_r\left(\boldsymbol{S}^{(k)}\right)\right)$$

where $\mathcal{T}_r(\boldsymbol{X}) = \boldsymbol{X} \oslash (\boldsymbol{X}\boldsymbol{1}\boldsymbol{1}^\top)$ renormalizes rows to sum to 1, and $\mathcal{T}_c(\boldsymbol{X}) = \boldsymbol{X} \oslash (\boldsymbol{1}\boldsymbol{1}^\top\boldsymbol{X})$ renormalizes columns. This projection ensures that our decoded output $\hat{\boldsymbol{y}}_t$ represents a valid probabilistic matching. Recent work in manifold hyper-connections (mHC) (Xie et al., 2025) uses Sinkhorn-Knopp for stable matrix-valued gating on the residual stream of a neural network. We propose this scalable approach over direct Birkhoff polytope parameterization (Yang & Gao, 2026) as a convex combination of all permutation matrices of size n, which becomes intractable ($O(n!)$) for even small n.

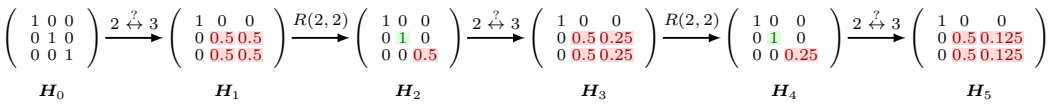

Figure 12: Visualizing the exponential decay of unnormalized probability mass in a Linear RNN for the marginal state. We start with the identity state ($\boldsymbol{H}_0$). A probabilistic mixing of elements 2 and 3 ($\boldsymbol{H}_1$) followed by a deterministic reveal of element 2 at position 2 ($\boldsymbol{H}_2$) 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.