WEIGHTED STATE SOUPING FOR EFFICIENT LONG-CONTEXT RETRIEVAL

Anonymous authors
Paper under double-blind review

ABSTRACT

State space models (SSMs) combine fast inference and fixed state size with competitive language-modeling performance. Recently, *state souping* has been proposed as a procedure whereby an SSM independently encodes chunks of tokens like documents, tool definitions, or memory records. The resulting state encodings may be cached and recombined to cheaply construct a context for downstream queries. However, both state souping and vanilla SSMs deteriorate on long, crowded contexts, limiting their utility for long-horizon domains. To address this issue, we propose *weighted state souping*, a simple method that learns query-dependent weights for composing cached states. The resulting models improve long-context performance substantially across both single- and multi-hop document question answering, needle-in-a-haystack retrieval, variable tracking, and tool use. Our method can be further adapted into an efficient sparse variant, yielding near-constant encoding cost with respect to context length while preserving (and sometimes *improving*) performance. We also present simple theoretical arguments that illustrate why souping works and when it may fail. Weighted state souping thus combines reusable context representations with selective retrieval, offering an efficient approach to language modeling over large collections of documents, tools, and other forms of structured context that increasingly dominate modern agentic workloads.

1 INTRODUCTION

Many applications of large language models (LLMs) require long-lived context organized into discrete chunks of tokens: tool definitions for an agent (Schick et al., 2023; Patil et al., 2025), documents for question answering or retrieval-augmented generation (Rajpurkar et al., 2016; Yang et al., 2018; Trivedi et al., 2022; Lewis et al., 2020; Guu et al., 2020), and memory records or chat histories that persist across interactions (Packer et al., 2023; Wu et al., 2024; Salemi et al., 2024). A useful memory architecture must retrieve relevant chunks from an increasingly crowded store without repeatedly processing the entire history.

In these settings, common practice concatenates the chunks of tokens and processes them as a single flat sequence. With the rising popularity of state space models (SSMs) and related fixed-state architectures (Gu & Dao, 2024; Dao & Gu, 2024; Katharopoulos et al., 2020; Schlag et al., 2021; Sun et al., 2024), an alternative approach becomes possible. Rather than perform a single autoregressive pass on a concatenated sequence, each chunk can be independently encoded into a state. The resulting collection of states are then point-wise summed to initialize the model before it processes a query (Figure 1). By analogy to model souping (Wortsman et al., 2022), this procedure is called *state souping* (PiĂłro et al., 2024; Jafari et al., 2025; Liu et al., 2025): the quantities combined are input-dependent recurrent states, rather than model parameters.

State souping has an obvious efficiency advantage. State encodings can be computed in parallel and, for slowly changing chunks like memory records or tool definitions, cached and reused. This ability to cache representations makes souping naturally suited for handling the structured context that increasingly dominates modern agentic workloads: inference can depend on selected state encodings without repopulating the context with large memory artifacts, tool definitions, and documents. However, finite-state SSMs struggle to copy or retrieve from long contexts (Jelassi et al., 2024; Waleffe et al., 2024; Arora et al., 2024), and state souping performance similarly decays as more

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

documents are added (Jafari et al., 2025). This loss of selectivity limits souping precisely where persistent memory is most useful.

In this work, we introduce *weighted state souping*, which learns attention-like weights over state encodings. Across document QA, needle-in-a-haystack retrieval, variable tracking, and tool use, weighting substantially improves robustness as contexts grow beyond training lengths. Our method can be further adapted into a sparse variant that permits near-constant encoding cost while preserving — and sometimes *improving* — performance. Our main contributions are:

- a novel weighted state souping procedure that permits robust selectivity, strong generalization on long contexts, and substantial efficiency gains over both Transformers and traditional SSMs,
- a comprehensive evaluation over diverse benchmarks including multi-document question answering, needle-in-a-haystack retrieval, variable tracking, and agentic tool use,
- a simple theoretical account of how souping successfully retrieves and composes across chunks, and its potential weaknesses compared against sequential models.

2 Related work

State souping is a relatively recent and sparsely studied method. PiĂłro et al. (2024) introduced the term, and explored linear combinations of recurrent states for in-context skill transfer. Jafari et al. (2025) study independently encoded document states, showing that finetuning enables simple pooling operators to support downstream QA and retrieval. PICASO (Liu et al., 2025) derives state-composition rules similar to souping for permutation-invariant inputs. We build on these works by showing how query-dependent weighting permits both significantly stronger length generalization and substantial efficiency wins.

State souping can be seen as an approximate form of state caching, a procedure whereby states generated from common prefixes can be cached and reused exactly for downstream requests. In Transformers, this procedure typically goes by the name KV caching (Shazeer, 2019; Kwon et al., 2023). In SSMs, state caching has been explored particularly in the context of hybrid models, where common KV caching methods for attention blocks may not transfer directly to SSM layers. In particular, Pan et al. (2025) explore SSM-specific cache eviction policies to maintain an efficient prefix cache for SSMs. Shirokikh & Nikolenko (2026) explore a complementary idea in which cache checkpoints are saved along a sequence; future generation resumes from the deepest checkpoint that matches the current prefix.

Souping is also related to state tuning, the process of learning an initial state for a recurrent model. This process has been studied in the context of parameter-efficient fine tuning (PEFT) for state space models (Galim et al., 2024; Kang et al., 2025; Yoshimura et al., 2025). Kang et al. (2025) in particular find that *state offset tuning*, in which a learned offset is added to every recurrent state, outperforms initial state tuning and other PEFT methods like LoRA. State tuning has also been studied in the RWKV community (Xiao et al., 2025), and historically in early RNNs (Forcada & Carrasco, 1995).

Our weighted approach to state souping is similar to methods that attempt to introduce attention-like features to state space models, interpolating between Transformers and SSMs. In particular, Akhauri et al. (2024) introduce a method in which SSMs produce compressed key-value representations before applying attention. Similarly, Infini-attention (Munkhdalai et al., 2024) combines local attention with a compressive recurrent memory. Additionally, Memory Caching (Behrouz et al., 2026) explores storing and combining recurrent memory checkpoints through attention-like weights. Our approach focuses on structured contexts that permit state souping, and our weighting does not retain token-by-token attention to cached states during generation.

3 Weighted State Souping

State souping operates in the following way. Let an input contain k token chunks $\mathbf{c}_1, \ldots, \mathbf{c}_k$ and query $\mathbf{q}$. The SSM independently encodes each chunk into its final state $\mathbf{s}_i$, where the state encompasses the collection of recurrent states across all layers. We then perform a point-wise average over the k encodings to make the "soup" $\mathbf{s}_0 = \frac{1}{k}\sum_{i=1}^{k} \mathbf{s}_i$. $\mathbf{s}_0$ becomes the initialization for the SSM as it

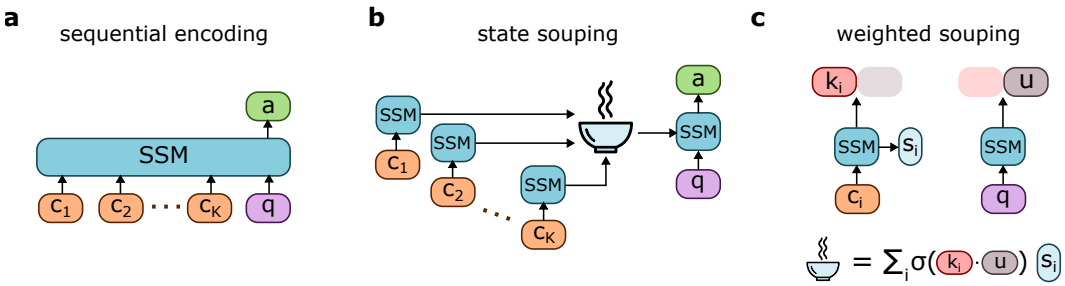


Figure 1: **Sequential encoding versus state souping. (a)** A typical SSM operates by concatenating input chunks together with the user query, and performs a single, long autoregressive pass to produce the answer. **(b)** A state souping SSM encodes each input chunk independently, then soups the resulting state encodings into a single encoding used to initialize the model as it processes the user query. **(c)** In our weighted state souping method, we augment the SSM with an additional linear readout that produces key vectors from the input chunks and a query vector from the user query. The soup is constructed as a weighted sum with weighting function σ.

processes the user query (Figure 1). We use the same SSM for encoding and generation, but these models can be different. Souping is not an inference-time only intervention, and requires either learning from scratch or additional finetuning on a pretrained SSM to teach the model to used this composed initialization (Jafari et al., 2025).

Crucially, the only channel through which the input chunks influence the query is the souped initialization. Since SSMs perform simple linear state updates, a soup constructed through an average of states can be viewed as a naive approximation of the state construct from a concatenated, sequential input. Such a *mean soup* is the approach used in prior work (PiĂłro et al., 2024; Jafari et al., 2025), and performs comparably with a sequential SSM. However, both mean souping and sequential SSMs struggle when operating over long contexts containing many input chunks (Jafari et al., 2025; Chen et al., 2024; Arora et al., 2024). As the number of chunks increases, a mean soup or finite SSM state becomes increasingly noisy, degrading performance.

Query-dependent weighting may instead concentrate the souped state on relevant chunks, the inspiration for *weighted state souping*. Our method is related to other schemes that weight input tokens in sequential SSMs (Akhauri et al., 2024; Behrouz et al., 2026), though our focus is on a souping architecture for chunked inputs. To compute these weightings, we add a learnable linear readout that maps the final activation to a key and query vector. Each chunk produces key $\mathbf{k}_i$. Starting from a zero-initialized state, a pass over the user query produces $\mathbf{u}$. We then compute the souped initialization as $\mathbf{s}_0 = \sum_{i=1}^{k} \sigma(\mathbf{k}_i \cdot \mathbf{u})\, \mathbf{s}_i$, with weighting function $\sigma(\cdot)$. The linear readout and base model are jointly finetuned. See Figure 1c for an illustration of this process [1].

For σ, we use the softmax function with cosine-normalized scores (see Appendix C for details). We also explore *sparse souping*, an efficient variant in which we mask at inference time all but the top-N_{top} logits before softmax, retaining normalized weights while avoiding retrieving zero-weight states. Masked states do not need to be retrieved, providing significant efficiency advantages for sparse souping (Section 4.2). Other reasonable choices of weighting functions include sigmoidal and linear weights, though we find that softmax performs best in our experiments.

4 Experiments

We use a 1.5B Mamba-2 SSM (Dao & Gu, 2024) and compare softmax-weighted, sparse, and mean souping against sequential Mamba-2 and a matched Transformer with rotary positional encodings (Su et al., 2021).

We evaluate our models across a diverse range of tasks. For document QA, we use SQuAD (Rajpurkar et al., 2016) for single-hop; HotpotQA (Yang et al., 2018) and MuSiQue (Trivedi et al., 2022) for multi-hop. For NIAH and variable tracking, we use RULER (Hsieh et al., 2024). For tool use, we

[1]The weighting function $\sigma(\cdot)$ can include a normalization term, making it effectively a function of all keys. For simplicity, we omit the addition of arguments that occur only in the normalization.

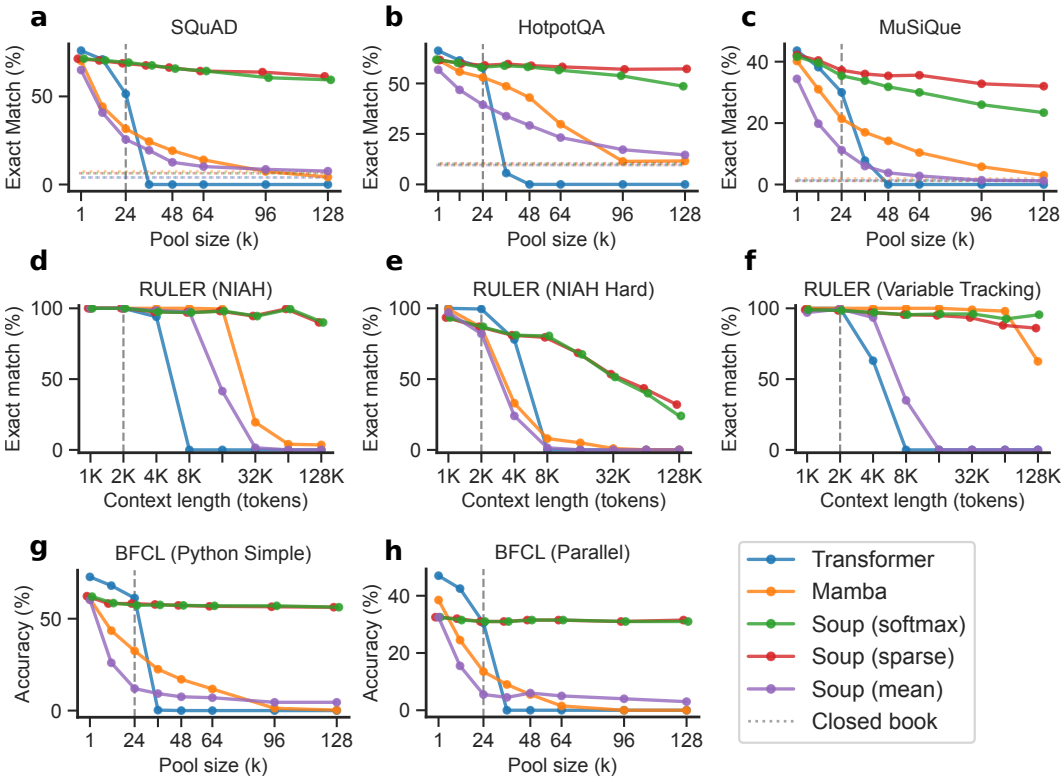


Figure 2: **Weighted state souping benchmark results.** Vertical gray dashed lines indicate the max pool size or context length observed during training. **(a - c)** Single-hop (SQuAD) and multi-hop (HotpotQA, MuSiQue) benchmarks. We plot accuracy as an exact match between the gold answer span and the model prediction. Dotted lines represent closed book accuracy, where no documents are prompted in-context. Sparse souping keeps top $N_{top} = 4$ chunks. **(d,e)** NIAH subsets of RULER. In the easy version, a single needle consisting of a random number is hidden among a natural language haystack of variable size. In the hard version, the haystack consists entirely of needle-like distractors, requiring precise retrieval. Sparse souping keeps top $N_{top} = 20$ chunks. **(e)** Variable tracking subset of RULER. The input consists of random variable assignments embedded in random noise sentences, and the model must infer the correct value of a randomly chosen variable. Sparse souping keeps top $N_{top} = 20$ chunks. **(f)** Simple Python subset of BFCL. The model is prompted with a list of Python functions and descriptions, and must execute a correct function call given a user query. Sparse souping keeps top $N_{top} = 4$ chunks. **(g)** Parallel subset of BFCL. The setup is the same as in *(f)*, but the model needs to execute the function multiple times.

use the Berkeley Function Calling Leaderboard (BFCL) (Patil et al., 2025). To probe long-context performance on document QA and BFCL, we augment the input with additional distractors and plot performance for varying pool size k. If k is smaller than the number of gold documents or gold tool definitions (relevant only for the $k = 1$ datapoints), then the input consists only of gold samples. This suite tests selective retrieval and reasoning as the available context grows, crowding the relevant chunks with increasingly more distractors. See Appendix C for additional details on training, evaluation, and our benchmarks.

4.1 Benchmark results

Figure 2 summarizes the main results; Appendix A reports on additional metrics and task subsets.

In general, weighted state souping performs strongly even at large pool size k. While mean souping and sequential Mamba tend to deteriorate as the number of chunks increases, weighted souping matches their performance at small k, and persists in near-constant performance as k increases to several times beyond the max training size. This phenomenon persists across a diversity of tasks and

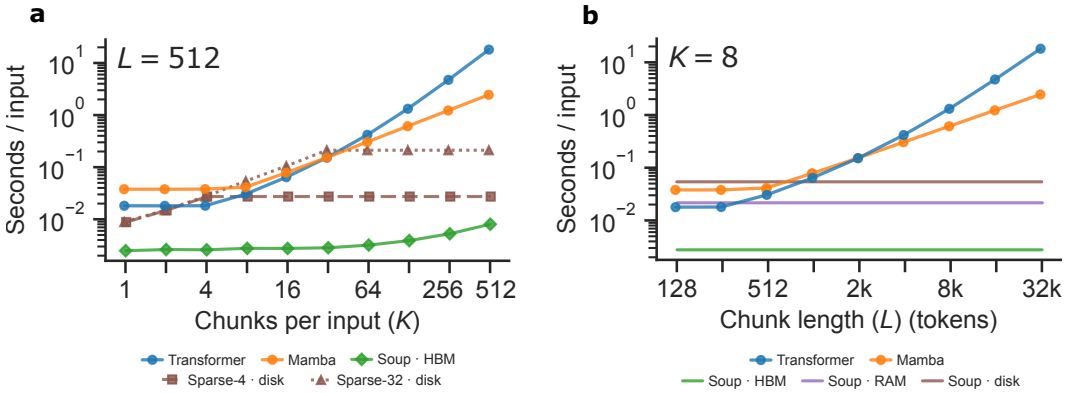


Figure 3: **Computational cost of souping.** We measure the time it takes to encode a set of input chunks on an AWS H100 node. For the souping models, we include the time it takes to encode a 128 token query and compute the softmax weights. The state encodings are retrieved from either GPU high bandwidth memory (HBM), system RAM, or system disk. We plot a subset of timings here, with the full results in Figure 7. We include two sparse souping variants, one that keeps the top $N_{top} = 4$ documents and another that keeps the top $N_{top} = 32$. (**a**) We hold the chunk length fixed at $L = 512$ tokens and increase the total number of chunks. Encoding time therefore grows linearly with increasing chunks for all models. For the HBM model, at low K encoding the query dominates the encoding time, resulting in a flat trend. (**b**) We hold the number of chunks fixed at $K = 8$ and increase the length of each chunk. Since the number of chunks is constant, souping retrieves the same number of chunks for each query, resulting in a flat trend across all L where the height is controlled by retrieval speed.

benchmarks, including in multi-hop settings like HotpotQA, MuSiQue, and RULER variable tracking where the model must link information across multiple chunks. Similar to other works (Malach et al., 2026; Zhou et al., 2024), we observe that Transformers do not extrapolate well to sequences longer than the training data, though their in-distribution performance tends to be strong.

Interestingly, sparse souping matches or improves on full softmax souping, particularly on multi-hop QA (HotpotQA, MuSiQue). Softmax mass may diffuse across many documents as k increases. Sparse souping instead may beneficially concentrate attention in the most relevant indices. An analogous benefit has been observed in sparse attention for Transformers (Veličković et al., 2024; Vasylenko et al., 2025; Zhang et al., 2024). One weakness of sparse souping is the need to specify the number of chunks N_{top} to keep. If N_{top} is insufficiently large, critical chunks may be dropped and the model's performance suffers (Figure 5f). Designing a weighting function with adaptive sparsity represents an interesting avenue for further exploration.

While the performance of weighted souping is generally strong, it underperforms at small k in the *parallel multiple* subset of BFCL, despite strong performance on *python simple* and *parallel* (Figure 6). Whether this reflects an optimization issue or a fundamental limitation remains open.

4.2 Computational Efficiency

The primary computational advantage of souping is in its ability to cheaply retrieve a state encoding from memory, rather than autoregressively re-encoding every input. Since inputs like memory artifacts and tool definitions change slowly, their state encodings may be routinely cached for future retrieval. Hence, a souped model may be considerably faster than non-souped alternatives, particularly as the context length grows. Figure 3 compares the measured time to encode an input between souping, Mamba, and Transformer models. See Figure 7 for additional timings.

There are a number of further considerations for souping. 1) Weighted souping must also factor in the time to perform an initial pass over the user query to generate a query vector, and the time it takes to compute the weightings. In general, user queries average under 100 tokens (Zheng et al., 2024) and computing softmax weights is extremely fast, so the total added cost of computing the weights is typically negligible and amortized over the number of input chunks. 2) The retrieval time

is heavily influenced by the specific memory holding the cached state encodings. GPU HBM is significantly faster than system RAM, which in turn is faster than system disk. Depending on where the encodings live, retrieval may in the worst case be slower than encoding from scratch. 3) Sparse souping improves efficiency further, since chunks assigned zero weight do not need to be retrieved. Hence, if the number of nonzero weights is constant with respect to the context length (which is true in our setting), then the encoding performance of sparse souping is effectively constant as context length grows[2].

5 THEORY: RETRIEVAL AND COMPOSITION

The experiments in Section 4 provide the main evidence for the effectiveness of weighted state souping. To complement these empirics, we present a simple theoretical analysis that begins to answer the questions of why souping works, and what potential strengths and weaknesses it may have compared with sequential models. Our arguments are primarily expressivity-based: we provide constructions showing that both souping and sequential models can implement an induction-head retrieval task representative of our empirical benchmarks. We also discuss possible weaknesses that independent chunk encoding introduces, namely a sensitivity to chunk boundaries and the inability to encode cross-chunk representations. We conclude with a discussion of the benefits weighted souping has for retrieval over long, crowded contexts. Our analysis is by no means a complete theoretical account of state souping. Instead, we introduce simple intuitions that build towards a principled understanding of the method.

5.1 TASK

We study a key-value version of the induction-head retrieval task (Olsson et al., 2022) and its multi-hop extension (Sanford et al., 2024).

Let $[N] = \{1, \dots, N\}$, and introduce disjoint key and value vocabularies $\mathcal{K} = \{k_a : a \in [N]\}$ and $\mathcal{V} = \{v_a : a \in [N]\}$. An input context defines a partial function $f : [N] \to [N]$ by listing pairs $(k_a, v_{f(a)})$, with each key appearing at most once. Given a query k_q, the n-hop induction target is

$$y_n(f, q) = v_{f^n(q)},$$

whenever $f(q), \dots, f^n(q)$ are defined. Thus one hop retrieves the value paired with k_q; each subsequent hop uses the index of the retrieved value as the next key. The input context is presented to the model as concatenation of all pairs followed by the query,

$$k_a \, , v_{f(a)} \, , k_b \, , v_{f(b)} \, , \dots \, , k_q \, .$$

A sequential model reads this sequence in one pass. A souping model encodes each pair independently. A chain of length n is a sequence of distinct indices $a_0, \dots, a_n$ with $f(a_i) = a_{i+1}$. Multiple chains and irrelevant pairs may appear in the context. Pair order is arbitrary except in the explicitly ordered setting below.

5.2 MODEL

We consider a stripped-down linear attention SSM with Mamba-like history decay and augmented residuals. Let $z_1, \dots, z_T$ be the tokens and let $\mathbf{e}(z_t) \in \mathbb{R}^d$ be orthonormal vocabulary embeddings, with $d \geq 2N$. Initialize the residual stream as $\mathbf{r}_t^{(0)} = [\mathbf{e}(z_t); \mathbf{0}] \in \mathbb{R}^{2d}$. At layer $\ell \in [D]$, a causal width-two convolution copies the previous and current residual vectors into separate coordinates:

$$\mathbf{x}_t^{(\ell)} = \begin{bmatrix} I_{2d} \\ 0 \end{bmatrix} \mathbf{r}_{t-1}^{(\ell-1)} + \begin{bmatrix} 0 \\ I_{2d} \end{bmatrix} \mathbf{r}_t^{(\ell-1)} = \begin{bmatrix} \mathbf{r}_{t-1}^{(\ell-1)} \\ \mathbf{r}_t^{(\ell-1)} \end{bmatrix} \in \mathbb{R}^{4d}. \tag{1}$$

This is the copying role of the convolution identified by Huang et al. (2025). We then perform the SSM state update as

$$H_t^{(\ell)} = \alpha H_{t-1}^{(\ell)} + (W_k^{(\ell)} \mathbf{x}_t^{(\ell)})(W_v^{(\ell)} \mathbf{x}_t^{(\ell)})^\top, \qquad \mathbf{h}_t^{(\ell)} = (H_t^{(\ell)})^\top W_q^{(\ell)} \mathbf{x}_t^{(\ell)}, \tag{2}$$

[2]Technically, the number of weights increases with context size as well. However, since computing weights is considerably faster than retrieval, the context size needs to be impractically immense before the time it takes to compute the weights becomes noticeable.

where $H_t^{(\ell)} \in \mathbb{R}^{d\times d}$ is the SSM state; $W_k^{(\ell)}, W_v^{(\ell)}, W_q^{(\ell)} \in \mathbb{R}^{d\times 4d}$ are the key, value, and query projections; and $\alpha \in (0, 1]$ is the history decay. We omit bias terms.

The residuals are updated as

$$\mathbf{r}_t^{(\ell)} = \mathbf{r}_t^{(\ell-1)} + W_o \mathbf{h}_t^{(\ell)}, \qquad W_o = \begin{bmatrix} 0 \\ I_d \end{bmatrix}.$$

In this way, $\mathbf{r}_t^{(\ell)} = [\mathbf{e}(z_t); \mathbf{u}_t^{(\ell)}]$ preserves the token embedding in the first block and accumulates updates in the second block.

The final output logits produced by the model are $U\mathbf{u}_T^{(D)}$, where $U \in \mathbb{R}^{N\times d}$ is the readout matrix. To extract a concrete next-token prediction, we take the argmax over these logits.

Sequential encoding starts from zero-initialized recurrent states and a zero convolution buffer. For souping, each chunk c_i is encoded independently from zero at every layer, giving states $H_i^{(\ell)}$. The query then starts from the weighted, souped initialization $H_{\text{init}}^{(\ell)} = \sum_{i=1}^{K} w_i H_i^{(\ell)}$, with its convolution buffer reset to zero. The weights w_i are shared across layers and fixed during query processing. Unless stated otherwise, chunk encoding and weighting have no access to the chunks' original positions: permuting the chunks permutes their weights and leaves the soup unchanged. This assumption is accurate of mean and content-based softmax souping. Our discussion below focuses on expressivity rather than strict learnability, and assumes exact arithmetic and suitably chosen positive weights.

Remark 1. *Linear states permit souping. For a single layer with matching convolution resets, sequential encoding equals a weighted sum of independently encoded chunk states.*

There is a simple algebraic reason that states from this model can be souped. Split a context into chunks $c_1, \ldots, c_K$ of lengths $m_1, \ldots, m_K$, and let $H(c_i)$ be the final state obtained by encoding c_i from zero. Let $H_{\text{seq}}(c_1 \| \cdots \| c_K)$ be the final state obtained by encoding the concatenated sequence of chunks. If we reset the convolution at the same boundaries in both schemes, then unrolling (2) gives

$$H_{\text{seq}}(c_1 \| \cdots \| c_K) = \sum_{i=1}^{K} \alpha^{\sum_{j>i} m_j} H(c_i). \tag{3}$$

Thus a sequential linear state is itself a weighted soup of that layer's chunk states, an identity also observed in prior state-composition work (Liu et al., 2025). With $\alpha = 1$ and a one layer model, sequential and souping are functionally equivalent.

However, there are a few important differences to keep in mind. On one hand, learned state souping replaces the fixed recency weights $\alpha^{\sum_{j>i} m_j}$ in (3) with content-dependent weights, permitting selection based on relevance rather than position. On the other hand, independently encoded chunks cannot incorporate information from other chunks during encoding. Particularly when a model has two or more layers, its sequential upper-layer representations may therefore differ from its independently encoded representations, potentially impacting efficiency when cross-chunk representations would be advantageous (see Remark 5 below).

5.3 One-hop induction

We begin by considering the one-hop induction-head task, reflecting the setting in Olsson et al. (2022). We observe that both a sequential and souping model can solve the task, but chunk boundaries that separate the task's key-value pairs can make souping fail.

Proposition 2. *Suppose that every key-value pair is contained in one chunk, each key occurs at most once, and the queried pair's chunk receives positive soup weight. A single layer model with sufficient width (Section 5.2) solves one-hop induction for both sequential encoding and weighted state souping.*

See Section B.1 for the construction proving this statement. Proposition 2 demonstrates that even a single-layer SSM is sufficiently expressive to perform one-hop induction. Together with our earlier discussion in Remark 1, we observe that both souping and sequential models can readily perform in-context retrieval. However, souping is sensitive to chunk boundaries, as we establish in the next proposition.

Proposition 3. *Suppose that chunk boundaries split every key-value pair (i.e. every token is encoded as a separate chunk). Then no souping model solves all one-hop induction instances.*

Proof. Choose distinct indices a, b, u, v. Consider the two token sequences

$$C = (k_a, v_u, k_b, v_v), \qquad y_1(C, a) = v_u,$$
$$C' = (k_a, v_v, k_b, v_u), \qquad y_1(C', a) = v_v.$$

After splitting into singleton chunks, both inputs have the same multiset $\{k_a, k_b, v_u, v_v\}$. For the same query, souping therefore produces the same state for C and C'. Hence, the model must return the same output on both inputs, though their labels differ, failing the task on at least one input. □

The setting in Proposition 3 is extreme, but underlines the sensitivity of souping to correctly formed chunks. However, this does not mean that souping cannot link information between chunks — our multi-hop document QA and variable tracking experiments demonstrate that souping models can robustly link information across multiple chunks. In the next section, we observe that souping can readily link disparate chunks when performing the multi-hop generalization of our task.

5.4 Multi-hop induction

We consider the multi-hop generalization of our induction-head task. We discuss how souping can continue linking information between chunks to solve the task, though under certain circumstances, it may be less efficient that a sequential input model.

Proposition 4. *Suppose that every key-value pair is contained in one chunk, each key occurs at most once, and every chunk containing an edge of the queried n-hop path receives positive soup weight. Souping and sequential models can solve the n-hop induction task by composing the construction in Proposition 2 across n layers.*

See Section B.2 for the construction proving this statement. In this way, we establish the basic competency of souping models to solve these classic in-context retrieval tasks. However, one advantage sequential models retain over souping is their ability to learn cross-chunk representations in deeper layers. Because souping encodes chunks independently, it remains impossible to directly encode information between chunks in deeper layers. This expressivity difference may lead to capability separations between sequential and souping models.

For instance, suppose the inputs to our multi-hop induction task are favorably ordered. Consider C node-disjoint chains presented consecutively in forward order, with each chain taking the form

$$(k_{a_{c,0}}, v_{a_{c,1}}),\ (k_{a_{c,1}}, v_{a_{c,2}}),\ \ldots,\ (k_{a_{c,n-1}}, v_{a_{c,n}}). \tag{4}$$

Suppose the query always names a chain's beginning and asks for its n-hop endpoint. Then we observe the following.

Remark 5. *For the ordered, endpoint-only task with $n = 2^L$ and $\alpha = 1$, an $L+1$-layer sequential model can form a final-layer state containing only the C start-to-end associations, effectively compressing the sequence. Souping models that independently encode single pairs in isolation cannot implement this same compression.*

The construction is given in Section B.3. Each of the first L layers stores reverse associations whose hop length doubles with depth, allowing later tokens to identify earlier nodes in their chain. The final layer then writes

$$H_{\text{end}} = \sum_{c=1}^{C} \mathbf{e}(k_{a_{c,0}})\mathbf{e}(v_{a_{c,n}})^\top,$$

so a query at a chain start directly retrieves its endpoint. Independently encoding a single pair cannot perform this cross-chunk preprocessing, though souping can still compose the original relations at query time as in Proposition 4.

A simple counting argument clarifies why the number of retained associations matters for storage.

Proposition 6. *A fixed B-bit state that must recover m associations in $[N]$ requires $B \geq \lceil m \log_2 N \rceil$, for $N \geq 2$.*

Proof. There are N^m possible assignments. Distinct assignments must yield distinct states, since they disagree on at least one query. Thus $2^B \geq N^m$. Taking the log of both sides results in the desired bound $B \geq \lceil m \log_2 N \rceil$, for $N \geq 2$. □

For the ordered task, the final summary retains C endpoint associations instead of all Cn edges. Hence, Proposition 6 suggests that the SSM state for a sequential model could be smaller than that of a souping model to learn a task with a fixed number of chains C and hops n. Indeed, to learn Cn associations, a souping model would require $B = Cn \log_2 N$ bits, whereas a sequential model needs a factor of n fewer with just $B = C \log_2 N$ bits.

More generally, sequential encoding may permit useful composed representations in deeper layers. The doubling construction is similar in spirit to the pointer-doubling strategy of Sanford et al. (2024), and gives a logarithmic-depth sequential solution in this ordered setting. We conjecture that this may yield a depth separation from souping under suitable width and precision constraints.

5.5 Weighted state souping permits selective retrieval

While souping may be sensitive to chunk boundaries and exhibit less efficiency for favorably ordered inputs, weighted souping nonetheless retains a decisive advantage over sequential models for long-context retrieval. Poor long-context performance can be understood as a consequence of gating and noisy contexts. In a sequential model with Mamba-like history decay, a contribution at distance Δ is exponentially decayed by a factor α^{Δ}. Tong et al. (2026) have formalized the resulting tradeoff: reducing decay preserves distant signals but permits distractor variance to accumulate, whereas stronger decay controls that variance at the cost of attenuating earlier tokens. Hence, no setting permits a sequential model to retrieve from every token in a long context with even reliability.

Weighted souping bypasses this tradeoff by permitting direct attention on the relevant chunks. In this scheme, distractor variance is determined only by the selected set of chunks rather than the full context size. Earlier chunks are not attenuated compared with later chunks, so weighting is computed evenly across the whole context. Provided that the weights remain concentrated as the number of distractors increases, weighted souping may maintain strong long-context performance when sequential and mean-souped variants fail. In some sense, weighted souping interpolates between recurrent accumulation and attention-like selection. It is not equivalent to full attention (souping weights are scalar, chunk-level, and computed once from the original query). Nonetheless, our experiments support the strong retrieval performance of weighted souping models. Together with their efficiency benefits, weighted state souping presents an attractive paradigm for future models.

6 Discussion

Weighted state souping augments reusable context encoding with query-dependent selection. A lightweight routing head combines cached chunk states into a fixed-size recurrent initialization, improving long-context performance across document QA, retrieval, variable tracking, and tool use benchmarks. The results show that a fixed state need not imply uniformly compressing the entire available collection: each query determines which information enters the working state. Inference-time sparsification often improves both accuracy and state-transfer cost, particularly beyond the training pool size. The theory complements these results by demonstrating retrieval and cross-chunk composition, while identifying information lost at chunk boundaries and discussing the potential advantages of sequential processing.

There remain many avenues for further exploration. Our focus in these experiments has been on single-turn interactions, but generalizing souping to multi-turn interactions featuring several rounds of back-and-forth between user and assistant remains an unexplored direction. There are also many opportunities for improving weighted state souping, for instance by replacing fixed-N sparse souping with adaptive sparsity that adjusts to the problem, exploring alternative weighting schemes, or using thinking tokens in conjunction with souping. Our theoretical discussion offers initial intuitions for why state souping works and when it may fail, but there remain many opportunities for investigating potential memory and depth separations between souping and sequential models.

AI USE STATEMENT

In this work, we used generative AI tools for implementing experiments, assisting in formulating mathematical claims and writing proofs, discovering related literature, and polishing writing. We have not used generative AI tools for developing theoretical models or conceptual frameworks, proposing or refining hypotheses, designing or providing feedback on research methodology, or interpreting results. Generating synthetic datasets is not applicable to this work. We have reviewed all AI-assisted work by manually verifying and testing generated code, carefully reviewing, editing, and cherry-picking all generated math, and checking all suggested references identified in a literature search. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REFERENCES

Yash Akhauri, Safeen Huda, and Mohamed S Abdelfattah. Attamba: Attending to multi-token states. *arXiv preprint arXiv:2411.17685*, 2024.

Simran Arora, Sabri Eyuboglu, Aman Timalsina, Isys Johnson, Michael Poli, James Zou, Atri Rudra, and Christopher Ré. Zoology: Measuring and improving recall in efficient language models. In *Proceedings of 12th International Conference on Learning Representations (ICLR)*. ICLR, 2024.

Ali Behrouz, Zeman Li, Yuan Deng, Peilin Zhong, Meisam Razaviyayn, and Vahab Mirrokni. Memory caching: Rnns with growing memory. *arXiv preprint arXiv:2602.24281*, 2026.

Yingfa Chen, Xinrong Zhang, Shengding Hu, Xu Han, Zhiyuan Liu, and Maosong Sun. Stuffed mamba: Oversized states lead to the inability to forget. *arXiv preprint arXiv:2410.07145*, 2024.

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

Mikel L Forcada and Rafael C Carrasco. Learning the initial state of a second-order recurrent neural network during regular-language inference. *Neural computation*, 7(5):923–930, 1995.

Kevin Galim, Wonjun Kang, Yuchen Zeng, Hyung Il Koo, and Kangwook Lee. Parameter-efficient fine-tuning of state space models. *arXiv preprint arXiv:2410.09016*, 2024.

Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces. In *First Conference on Language Modeling*, 2024. URL https://openreview.net/forum?id=tEYskw1VY2.

Kelvin Guu, Kenton Lee, Zora Tung, Panupong Pasupat, and Mingwei Chang. Retrieval augmented language model pre-training. In *International conference on machine learning*, pp. 3929–3938. PMLR, 2020.

Cheng-Ping Hsieh, Simeng Sun, Samuel Kriman, Shantanu Acharya, Dima Rekesh, Fei Jia, Yang Zhang, and Boris Ginsburg. RULER: What's the real context size of your long-context language models? In *First Conference on Language Modeling*, 2024. URL https://openreview.net/forum?id=kIoBbc76Sy.

Ningyuan Huang, Miguel Sarabia, Abhinav Moudgil, Pau Rodriguez, Luca Zappella, and Federico Danieli. Understanding input selectivity in mamba: impact on approximation power, memorization, and associative recall capacity. *arXiv preprint arXiv:2506.11891*, 2025.

Yasaman Jafari, Zixian Wang, Leon Bergen, and Taylor Berg-Kirkpatrick. The surprising soupability of documents in state space models. *arXiv preprint arXiv:2505.24033*, 2025.

Samy Jelassi, David Brandfonbrener, Sham M Kakade, and Eran Malach. Repeat after me: Transformers are better than state space models at copying. *arXiv preprint arXiv:2402.01032*, 2024.

Wonjun Kang, Kevin Galim, Yuchen Zeng, Minjae Lee, Hyung Il Koo, and Nam Ik Cho. State-offset tuning: State-based parameter-efficient fine-tuning for state space models. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pp. 474–487, 2025.

Angelos Katharopoulos, Apoorv Vyas, Nikolaos Pappas, and François Fleuret. Transformers are rnns: Fast autoregressive transformers with linear attention. In *International conference on machine learning*, pp. 5156–5165. PMLR, 2020.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th symposium on operating systems principles*, pp. 611–626, 2023.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems*, 33: 9459–9474, 2020.

Tian Yu Liu, Alessandro Achille, Matthew Trager, Aditya Golatkar, Luca Zancato, and Stefano Soatto. Picaso: Permutation-invariant context composition with state space models. *arXiv preprint arXiv:2502.17605*, 2025.

Eran Malach, Omid Saremi, Sinead Williamson, Arwen Bradley, Aryo Lotfi, Emmanuel Abbe, Joshua Susskind, and Etai Littwin. To infinity and beyond: Tool-use unlocks length generalization in state space models. In *International Conference on Learning Representations*, volume 2026, pp. 44874–44900, 2026.

Tsendsuren Munkhdalai, Manaal Faruqui, and Siddharth Gopal. Leave no context behind: Efficient infinite context transformers with infini-attention. *arXiv preprint arXiv:2404.07143*, 2024.

NVIDIA. NVIDIA nemotron nano 2: An accurate and efficient hybrid Mamba-Transformer reasoning model. *arXiv preprint arXiv:2508.14444*, 2025. doi: 10.48550/arXiv.2508.14444. URL https://arxiv.org/abs/2508.14444.

Catherine Olsson, Nelson Elhage, Neel Nanda, Nicholas Joseph, Nova DasSarma, Tom Henighan, Ben Mann, Amanda Askell, Yuntao Bai, Anna Chen, et al. In-context learning and induction heads. *arXiv preprint arXiv:2209.11895*, 2022.

Charles Packer, Vivian Fang, Shishir_G Patil, Kevin Lin, Sarah Wooders, and Joseph_E Gonzalez. Memgpt: towards llms as operating systems. 2023.

Rui Pan, Zhuang Wang, Zhen Jia, Can Karakus, Luca Zancato, Tri Dao, Yida Wang, and Ravi Netravali. Marconi: Prefix caching for the era of hybrid llms. *Proceedings of Machine Learning and Systems*, 7, 2025.

Shishir G. Patil, Huanzhi Mao, Fanjia Yan, Charlie Cheng-Jie Ji, Vishnu Suresh, Ion Stoica, and Joseph E. Gonzalez. The berkeley function calling leaderboard (BFCL): From tool use to agentic evaluation of large language models. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 48371–48392. PMLR, 2025. URL https://proceedings.mlr.press/v267/patil25a.html.

Maciej PiĂłro, Razvan Pascanu, Johannes von Oswald, JoĂŁo Sacramento, et al. State soup: In-context skill learning, retrieval and mixing. *arXiv preprint arXiv:2406.08423*, 2024.

Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. SQuAD: 100,000+ questions for machine comprehension of text. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pp. 2383–2392, Austin, Texas, 2016. Association for Computational Linguistics. doi: 10.18653/v1/D16-1264. URL https://aclanthology.org/D16-1264/.

Stephen Robertson and Hugo Zaragoza. The probabilistic relevance framework: BM25 and beyond. *Foundations and Trends in Information Retrieval*, 3(4):333–389, 2009. doi: 10.1561/1500000019. URL https://doi.org/10.1561/1500000019.

Alireza Salemi, Sheshera Mysore, Michael Bendersky, and Hamed Zamani. Lamp: When large language models meet personalization. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 7370–7392, 2024.

Clayton Sanford, Daniel Hsu, and Matus Telgarsky. Transformers, parallel computation, and logarithmic depth. *arXiv preprint arXiv:2402.09268*, 2024.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *Advances in neural information processing systems*, 36:68539–68551, 2023.

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

Noam Shazeer. Fast transformer decoding: One write-head is all you need. *arXiv preprint arXiv:1911.02150*, 2019.

Mikhail Shirokikh and Sergey Nikolenko. Sparse prefix caching for hybrid and recurrent llm serving. *arXiv preprint arXiv:2605.05219*, 2026.

Dan Su, Kezhi Kong, Ying Lin, Joseph Jennings, Brandon Norick, Markus Kliegl, Mostofa Patwary, Mohammad Shoeybi, and Bryan Catanzaro. Nemotron-CC: Transforming Common Crawl into a refined long-horizon pretraining dataset. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 2459–2475, Vienna, Austria, 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.acl-long.123. URL https://aclanthology.org/2025.acl-long.123/.

Jianlin Su, Yu Lu, Shengfeng Pan, Ahmed Murtadha, Bo Wen, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding. *arXiv preprint arXiv:2104.09864*, 2021.

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. *arXiv preprint arXiv:2407.04620*, 2024.

William L Tong, Aryo Lotfi, Emmanuel Abbe, Kostas Vaggelakos, Vishnu Banna, Etai Littwin, Josh Susskind, Cengiz Pehlevan, and Eran Malach. On the importance of gating: Memorization vs. in-context learning in state space models. *arXiv preprint arXiv:2609.16540*, 2026.

Harsh Trivedi, Niranjan Balasubramanian, Tushar Khot, and Ashish Sabharwal. MuSiQue: Multihop questions via single-hop question composition. *Transactions of the Association for Computational Linguistics*, 10:539–554, 2022. doi: 10.1162/tacl_a_00475. URL https://aclanthology.org/2022.tacl-1.31/.

Pavlo Vasylenko, Hugo Pitorro, André FT Martins, and Marcos Treviso. Long-context generalization with sparse attention. *arXiv preprint arXiv:2506.16640*, 2025.

Petar Veličković, Christos Perivolaropoulos, Federico Barbero, and Razvan Pascanu. Softmax is not enough (for sharp size generalisation). *arXiv preprint arXiv:2410.01104*, 2024.

Roger Waleffe, Wonmin Byeon, Duncan Riach, Brandon Norick, Vijay Korthikanti, Tri Dao, Albert Gu, Ali Hatamizadeh, Sudhakar Singh, Deepak Narayanan, et al. An empirical study of mamba-based language models. *arXiv preprint arXiv:2406.07887*, 2024.

Mitchell Wortsman, Gabriel Ilharco, Samir Ya Gadre, Rebecca Roelofs, Raphael Gontijo-Lopes, Ari S. Morcos, Hongseok Namkoong, Ali Farhadi, Yair Carmon, Simon Kornblith, and Ludwig Schmidt. Model soups: Averaging weights of multiple fine-tuned models improves accuracy without increasing inference time. In *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pp. 23965–23998. PMLR, 2022. URL https://proceedings.mlr.press/v162/wortsman22a.html.

Di Wu, Hongwei Wang, Wenhao Yu, Yuwei Zhang, Kai-Wei Chang, and Dong Yu. Longmemeval: Benchmarking chat assistants on long-term interactive memory. *arXiv preprint arXiv:2410.10813*, 2024.

Liu Xiao, Li Zhiyuan, and Lin Yueyu. State tuning: State-based test-time scaling on rwkv-7. *arXiv preprint arXiv:2504.05097*, 2025.

Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William W. Cohen, Ruslan Salakhutdinov, and Christopher D. Manning. HotpotQA: A dataset for diverse, explainable multi-hop question answering. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pp. 2369–2380, Brussels, Belgium, 2018. Association for Computational Linguistics. doi: 10.18653/v1/D18-1259. URL https://aclanthology.org/D18-1259/.

Masakazu Yoshimura, Teruaki Hayashi, and Yota Maeda. Mambapeft: Exploring parameter-efficient fine-tuning for mamba. In *International Conference on Learning Representations*, volume 2025, pp. 94093–94117, 2025.

Xuechen Zhang, Xiangyu Chang, Mingchen Li, Amit Roy-Chowdhury, Jiasi Chen, and Samet Oymak. Selective attention: Enhancing transformer through principled context control. *Advances in Neural Information Processing Systems*, 37:11061–11086, 2024.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Tianle Li, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zhuohan Li, Zi Lin, Eric Xing, et al. Lmsys-chat-1m: A large-scale real-world llm conversation dataset. In *International Conference on Learning Representations*, volume 2024, pp. 22225–22257, 2024.

Yongchao Zhou, Uri Alon, Xinyun Chen, Xuezhi Wang, Rishabh Agarwal, and Denny Zhou. Transformers can achieve length generalization but not robustly. *arXiv preprint arXiv:2402.09371*, 2024.

A ADDITIONAL FIGURES

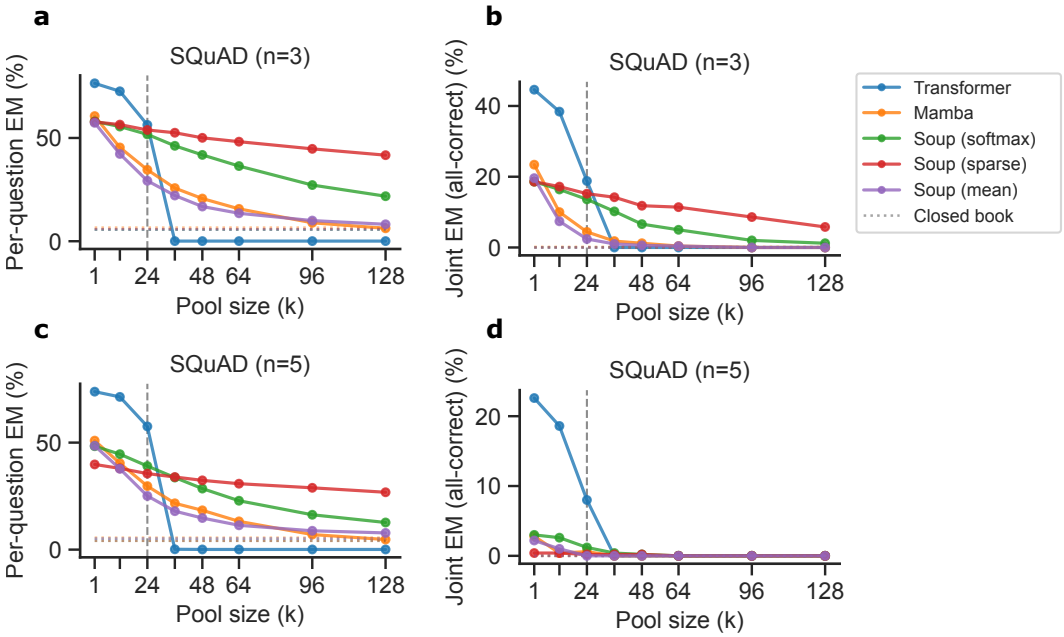


Figure 4: **Squad-n benchmark results.** As a further test of souping when the task requires attending to multiple documents, we construct a task based on SQuAD called *SQuAD-n*. Each example consists of n individual SQuAD document-question-answer sets. The model is prompted with all n documents (mixed among $\max(k-n, 0)$ distractors) followed by the questions, and must produce the answer spans corresponding to each question. We plot both per-question exact match accuracy in (**a, c**) as well as joint accuracy in (**b, d**) for $n = 3$ and $n = 5$. Souping generally performs competitively with vanilla Mamba, and sparse souping in particular maintains long context generalization better than full souping. However, both Mamba and souping underperform Transformers on in-context K in this setting. This may be in part due to the Transformer's sharper ability to track its context, and determine which questions have already been answered and which require an answer still.

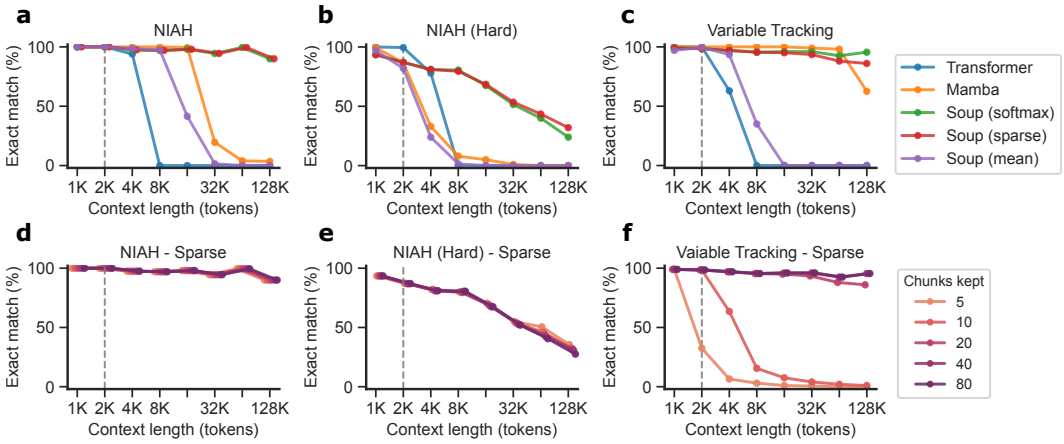


Figure 5: **Ruler benchmark, additional results.** In the main text, we present results on needle-in-a-haystack **(a,b)** and variable tracking **(c)**. We additionally plot a sweep over the number of chunks N_{top} kept by the sparse souping model in **(d - f)**. For the NIAH tasks, where only a single chunk is relevant, the value of N_{top} is irrelevant. However, for the variable tracking task, relevant variable assignments are distributed across many more chunks, so below a certain threshold, the model fails to learn the task well.

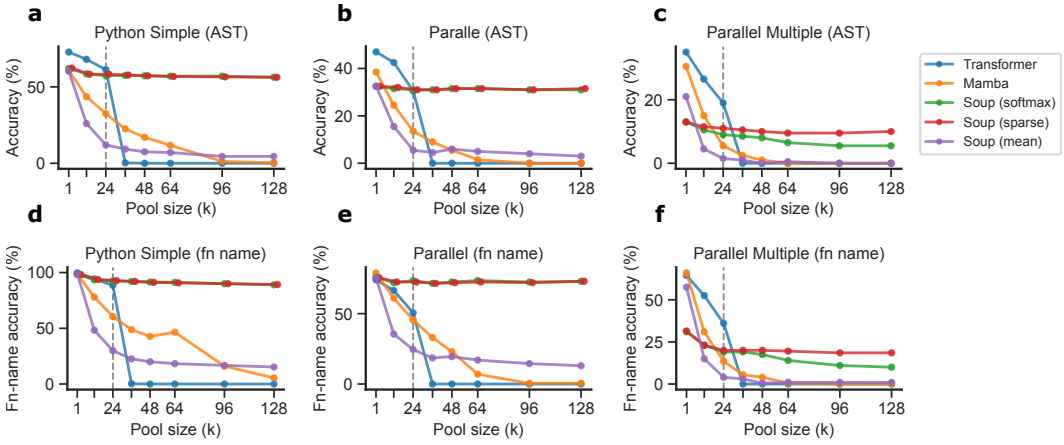


Figure 6: **BFCL benchmark, additional results.** In the main text, we present results on the Python Simple split of BFCL (**a**), in which the model must produce a single tool call given a list of tool definitions and a user query. Here, we also plot performance on the Parallel split (the model must call a single tool multiple times) and the Parallel Multiple split (the model must call multiple tools multiple times). We measure accuracy by BFCL's built-in abstract syntax tree (AST)-based checker (**a - c**) as well as a match based on function name alone (**d - f**). In general, weighted souping performs well for both Python Simple and Parallel, but a substantial gap opens on small, in-distribution K for Parallel Multiple. Mean souping performs better than weighted souping at $K = 1$ (where only gold tool definitions are prompted), indicating that souping may still work well, but the model is struggling to simultaneously learn appropriate weights and use a souped state initialization.

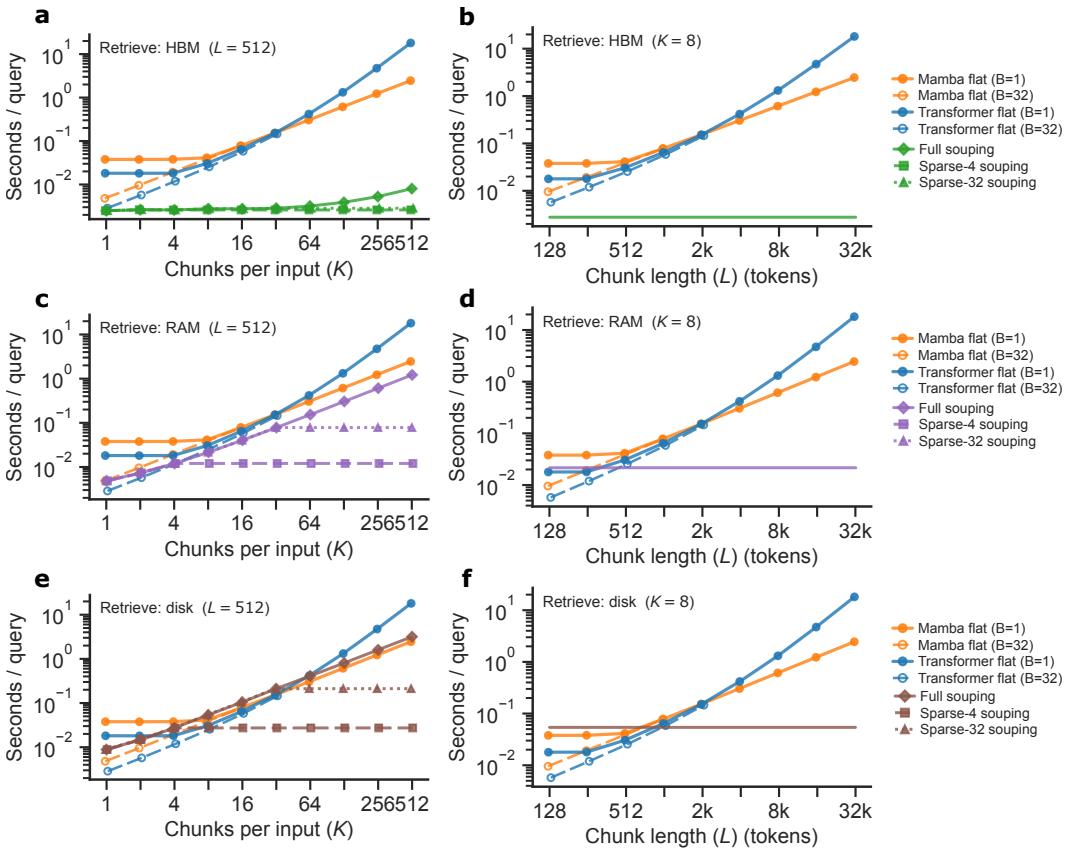


Figure 7: **Additional measurements on encoding time across models.** We plot additional sets of timings beyond those showin in Figure 3. In particular, we show full timings measured for full and sparse souping models retrieving from HBM (**a, b**), system RAM (**c, d**), and system disk (**e, f**). The timings for Transformer and sequential Mamba models in the main text are computed with batch size of 1. To give a sense for what the compute-bound timings look like, we plot the per-example timing for batch size 32 for Transformer and sequential Mamba models (dashed lines).

B CONSTRUCTIONS

We use the simplified SSM model in Section 5 throughout. In the constructions below, each block matrix column has width d, in the order $[\mathbf{e}(z_{t-1}); \mathbf{u}_{t-1}^{(\ell-1)}; \mathbf{e}(z_t); \mathbf{u}_t^{(\ell-1)}]$. At a reset boundary, the previous residual vector is zero. Recurrent states are read after the current token's write, as specified in (2).

B.1 ONE-HOP RETRIEVAL

Proof. Let $E_K = [\mathbf{e}(k_1) \ \cdots \ \mathbf{e}(k_N)]$ and $E_V = [\mathbf{e}(v_1) \ \cdots \ \mathbf{e}(v_N)]$. Orthogonality gives $E_K^\top E_K = E_V^\top E_V = I_N$ and $E_K^\top E_V = 0$. Define $A_K = E_K E_K^\top$ and $A_V = E_V E_V^\top$, and use depth $D = 1$. The four block columns correspond to $[\mathbf{e}(z_{t-1}); \mathbf{u}_{t-1}^{(0)}; \mathbf{e}(z_t); \mathbf{u}_t^{(0)}]$. Choose

$$\begin{aligned} W_k^{(1)} &= [A_K \ \ 0 \ \ 0 \ \ 0], \qquad W_v^{(1)} = [0 \ \ 0 \ \ A_V \ \ 0], \\ W_q^{(1)} &= [0 \ \ 0 \ \ A_K \ \ 0], \qquad U = E_V^\top. \end{aligned} \tag{5}$$

Together with the stated embeddings, convolution, residual map W_o, zero initial state, zero biases, and any fixed $\alpha \in (0, 1]$, these choices specify all model parameters. We suppress the layer superscript below.

At a value token $v_{f(a)}$, the previous token is k_a, so the state increment is $\mathbf{e}(k_a)\mathbf{e}(v_{f(a)})^\top$. At a key token, including the query, $W_v \mathbf{x}_t = \mathbf{0}$, so there is no write. At the query the state is therefore

$$H_q = \sum_{a \in \mathrm{dom}(f)} \gamma_a \mathbf{e}(k_a)\mathbf{e}(v_{f(a)})^\top.$$

For sequential encoding, $\gamma_a = \alpha^{T - t_a}$, where t_a is the position of the value token and T is the query position. For souping, if this pair lies in chunk $i(a)$ and its value has local position $t_{a,i(a)}$, then $\gamma_a = w_{i(a)} \alpha^{m_{i(a)} - t_{a,i(a)} + 1}$. These coefficients include the decay at the query token. Orthogonality now gives $\mathbf{h}_q = \gamma_q \mathbf{e}(v_{f(q)})$. The residual update gives $\mathbf{u}_q^{(1)} = \mathbf{h}_q$ because $\mathbf{u}_q^{(0)} = 0$, so $U\mathbf{u}_q^{(1)}$ has its only nonzero entry, $\gamma_q > 0$, at coordinate $f(q)$. Argmax returns the correct value. In the sequential model, continuous convolution across pair boundaries gives the same writes, since keys have zero value projection. □

B.2 MULTI-HOP RETRIEVAL WITH ADDITIVE RESIDUALS

Proof. Use the embedding matrices and A_K, A_V from the one-hop proof, and let $F = E_K E_V^\top$, so $F\mathbf{e}(v_a) = \mathbf{e}(k_a)$. The four block columns below correspond to $[\mathbf{e}(z_{t-1}); \mathbf{u}_{t-1}^{(\ell-1)}; \mathbf{e}(z_t); \mathbf{u}_t^{(\ell-1)}]$. In the first layer, use

$$\begin{aligned} W_k^{(1)} &= [A_K \ \ 0 \ \ 0 \ \ 0], \qquad W_v^{(1)} = [0 \ \ 0 \ \ A_V \ \ 0], \\ W_q^{(1)} &= [0 \ \ 0 \ \ A_K \ \ 0]. \end{aligned}$$

Since $\mathbf{u}_t^{(0)} = \mathbf{0}$, this is the one-hop construction. Every context output is zero: a context key has not yet been written, and a value has zero query projection. At the final query, $\mathbf{u}_q^{(1)} = \gamma_q \mathbf{e}(v_{f(q)})$.

Subsequent layers must add the next retrieval and subtract the previous working vector. This can be implemented by the attention block itself. Choose a fixed unit vector $\mathbf{b} = \mathbf{e}(v_1)$, orthogonal to all key embeddings, and let $\mathbf{c} = \sum_{a=1}^N \mathbf{e}(k_a)$ and $B = \mathbf{b}\mathbf{c}^\top$. Thus $B\mathbf{e}(k_a) = \mathbf{b}$ and $B\mathbf{e}(v_a) = \mathbf{0}$. For $\ell \geq 2$, set

$$\begin{aligned} W_k^{(\ell)} &= [A_K \ \ 0 \ \ B \ \ 0], \qquad W_v^{(\ell)} = [0 \ \ 0 \ \ A_V \ \ -I_d], \\ W_q^{(\ell)} &= [0 \ \ 0 \ \ B \ \ F]. \end{aligned}$$

At context positions the working vectors remain zero: a value writes the original association and has zero query projection; a key has zero value projection and queries $\mathbf{b}$, orthogonal to all association keys. Consequently, every layer's context state contains precisely the original associations, for both sequential encoding and independently encoded chunks.

At the final query, write $\mathbf{u} = \mathbf{u}_q^{(\ell-1)}$ and let

$$M = \sum_{a \in \mathrm{dom}(f)} \gamma_a \mathbf{e}(k_a)\mathbf{e}(v_{f(a)})^\top$$

be the context contribution after multiplying the initialized context state by α when processing the query token, with the same coefficients as in the one-hop proof. The query's key, value, and query projections are respectively $\mathbf{b}$, $-\mathbf{u}$, and $F\mathbf{u} + \mathbf{b}$. Since the recurrence reads after the current write,

$$\mathbf{h}_q^{(\ell)} = (M - \mathbf{b}\mathbf{u}^\top)^\top (F\mathbf{u} + \mathbf{b}) = M^\top F\mathbf{u} - \mathbf{u}.$$

Here $M^\top \mathbf{b} = \mathbf{0}$, $\mathbf{b}^\top F\mathbf{u} = 0$, and $\|\mathbf{b}\| = 1$. The additive residual therefore gives $\mathbf{u}_q^{(\ell)} = M^\top F\mathbf{u}_q^{(\ell-1)}$. Inductively,

$$\mathbf{u}_q^{(\ell)} = \left(\prod_{j=0}^{\ell-1} \gamma_{f^j(q)} \right) \mathbf{e}(v_{f^\ell(q)}).$$

Use $U = E_V^\top$ and the same α at every layer. Every factor is positive, so the final readout identifies $v_{f^n(q)}$. □

B.3 Ordered-chain construction

We give the explicit construction for Remark 5. Let $n = 2^L$, and let the C chains in (4) have mutually disjoint node sets, so $C(n+1) \le N$. Set $\alpha = 1$ and use the orthogonal embeddings, residual path, and convolution from Section 5. Convolution continues across pair boundaries in the sequential model. All recurrent states start at zero and all biases are zero. Write $A_K = E_K E_K^\top$, $A_V = E_V E_V^\top$, and $F = E_K E_V^\top$, as in the retrieval proofs.

The first L layers store reverse associations. Using the same block columns as in the proof of Proposition 4, set

$$\begin{aligned} W_k^{(1)} &= [0 \;\; 0 \;\; F \;\; 0], \qquad W_v^{(1)} = [A_K \;\; 0 \;\; 0 \;\; 0], \\ W_q^{(1)} &= [0 \;\; 0 \;\; A_K \;\; 0]. \end{aligned}$$

For $2 \le \ell \le L$, let $G = F^\top$ and set

$$\begin{aligned} W_k^{(\ell)} &= [0 \;\; 0 \;\; F + G \;\; 0], \qquad W_v^{(\ell)} = [0 \;\; I_d \;\; 0 \;\; -I_d], \\ W_q^{(\ell)} &= [0 \;\; 0 \;\; G \;\; I_d]. \end{aligned} \tag{6}$$

Working vectors at value tokens remain zero, whereas those at key tokens lie in the key-embedding span. In the first layer, each value token writes its reverse one-hop association using the preceding key's original embedding. For $\ell \ge 2$, a value token $v_{a_{c,j}}$ writes the reverse association with key $\mathbf{e}(k_{a_{c,j}})$ and value equal to the preceding key's working vector. A key token k_a instead writes $-\mathbf{e}(v_a)(\mathbf{u}_t^{(\ell-1)})^\top$ and queries $\mathbf{u}_t^{(\ell-1)} + \mathbf{e}(v_a)$. The value-embedding direction $\mathbf{e}(v_a)$ is orthogonal to all reverse-association keys and to the cancellation keys at earlier context positions, since each context key is unique. Its current write therefore subtracts exactly $\mathbf{u}_t^{(\ell-1)}$ from the attention output. After residual addition, the working vector contains only the newly retrieved reverse jump. These fixed key and value spans suffice at every layer.

Index edges by $j = 1, \ldots, n$, so edge j is $(k_{a_{c,j-1}}, v_{a_{c,j}})$. Let $\mathbf{u}_{c,j}^{(\ell)}$ denote the working vector at the key token of edge j in chain c, for $\ell \ge 1$. We claim

$$\mathbf{u}_{c,j}^{(\ell)} = \begin{cases} \mathbf{e}(k_{a_{c,j-2^\ell}}), & j \ge 2^\ell, \\ \mathbf{0}, & j < 2^\ell. \end{cases} \tag{7}$$

For the base case $\ell = 1$, the key token of edge j is $k_{a_{c,j-1}}$. For $j \ge 2$, the preceding edge has already written the reverse association from $k_{a_{c,j-1}}$ to $k_{a_{c,j-2}}$; for $j = 1$ no such association exists. This proves the stated base case. For $\ell \ge 2$, suppose the claim holds for layer $\ell - 1$. Put $s = 2^{\ell-1}$. At the value token of edge j, layer ℓ writes

$$\begin{cases} \mathbf{e}(k_{a_{c,j}})\mathbf{e}(k_{a_{c,j-s}})^\top, & j \ge s, \\ 0, & j < s. \end{cases}$$

The component of the state addressed by key embeddings therefore stores reverse jumps of length s. At the key token of edge j, the corresponding query component is $\mathbf{e}(k_{a_{c,j-s}})$ when $j \geq s$. If $j \geq 2s$, edge $j - s$ has already been processed, so its reverse association returns $\mathbf{e}(k_{a_{c,j-2s}})$. For $s \leq j < 2s$, no matching association has been written; for $j < s$, this query component is zero. Both cases give zero retrieval. In layers $\ell \geq 2$, the additional query component cancels the incoming working vector as described above. Orthogonality and disjoint chain nodes exclude all other matches, proving (7) after residual addition.

For $L \geq 1$, the final layer writes forward endpoint associations. Set

$$W_k^{(L+1)} = [0 \;\; A_K \;\; 0 \;\; 0], \qquad W_v^{(L+1)} = [0 \;\; 0 \;\; A_V \;\; 0],$$
$$W_q^{(L+1)} = [0 \;\; 0 \;\; A_K \;\; 0], \qquad U = E_V^\top.$$

By (7), the preceding key's working vector is nonzero only for $j \geq 2^L = n$. Thus only the final edge of each chain writes to this layer, producing

$$H_{\text{end}} = \sum_{c=1}^{C} \mathbf{e}(k_{a_{c,0}})\mathbf{e}(v_{a_{c,n}})^\top.$$

At a chain-start query, every reverse layer returns zero: no reverse association points back from a chain start, and the earlier occurrence of this start wrote no cancellation term. Hence the query enters the final layer with zero working vector, writes zero, and uses its original key embedding to retrieve the appropriate endpoint. The readout U identifies that value. For example, four-hop retrieval uses two reverse layers, storing jumps of lengths one and two, followed by a final layer associating each chain start with its four-hop endpoint.

C IMPLEMENTATION DETAILS

C.1 TASKS

Across the document-QA and tool-use benchmarks we probe long-context robustness with a shared *pooling* construction. Each example carries one or more *gold* items — the documents or tool definitions required to answer the query — which we always retain; we then add sampled *distractors* until the pool holds k items in total and shuffle so the gold position is randomized. Thus k counts the *total* number of in-context chunks (not the number of distractors), and $k = 1$ denotes a gold-only pool. Distractors are drawn without replacement from a split-local pool that excludes the example's own gold items, so training and evaluation distractors never cross split boundaries. During finetuning the pool size is sampled per example as $k \sim \mathrm{Uniform}[1, k_{\max}]$; at evaluation we sweep k well beyond $k_{\max}$ to test generalization. All tasks share the Nemotron-Nano tokenizer (NVIDIA, 2025) (vocabulary size 131,072) and are evaluated on the public validation split (the official test splits are hidden). RULER is handled differently, as described below.

Berkeley Function Calling Leaderboard (BFCL). A standard tool-use benchmark (Patil et al., 2025): the model is prompted with a list of tool definitions and must emit a tool call that solves the user query. Before evaluation we finetune each model for tool calling on a filtered subset of NVIDIA's Nemotron agentic SFT data (NVIDIA, 2025), keeping the ≈228,700 examples whose first assistant turn is a tool call. We evaluate on BFCL, reporting the *simple python* (one function, one call; 400 examples), *parallel* (one function, multiple calls; 200 examples), and *parallel multiple* (multiple functions, multiple calls; 200 examples) splits. Tool pools are built with a global BM25 retriever (Robertson & Zaragoza, 2009) over the ≈14,700-tool non-live vocabulary: candidate tools are scored by BM25, truncated to the top k, and the gold tool(s) are then force-included and randomly positioned. Training samples $k \sim \mathrm{Uniform}[1, 24]$; the headline evaluation sweeps $k \in \{1, 12, 24, 36, 48, 64, 96, 128\}$. Correctness is scored by BFCL's abstract-syntax-tree checker, which validates the function name and every argument value against the reference-answer set. We additionally report an order-independent function-name-only match.

Stanford Question Answering Dataset (SQuAD). A single-document extractive QA dataset (Rajpurkar et al., 2016): given a Wikipedia paragraph, the model quotes a text span answering the query. We use SQuAD v1.1 and draw distractor paragraphs from the same split. Training uses $k \sim \mathrm{Uniform}[1, 24]$; evaluation sweeps $k \in \{1, 12, 24, 36, 48, 64, 96, 128\}$ over a fixed subset of 500 validation questions (shared across all arms via a common seed). We report exact match (EM) accuracy under SQuAD normalization (lowercasing, and removal of punctuation, articles, and extra whitespace). Dotted lines in the figures report closed-book accuracy ($k = 0$, no documents in context) as a memorization floor.

HotpotQA. A multi-hop QA dataset (Yang et al., 2018). Each context paragraph becomes one document, and the annotated supporting paragraphs are marked gold (a multi-gold example). Pooling, evaluation k-sweep, example count, and metrics match SQuAD.

MuSiQue. A multi-hop QA dataset engineered to require genuine reasoning over many hops and to resist single-document shortcuts (Trivedi et al., 2022); it is a harder version of HotpotQA. We use the answerable variant (`musique_ans`), treating the annotated supporting paragraphs as gold and scoring against the gold answer and its aliases. Pooling, evaluation k-sweep, example count, and metrics match SQuAD.

SQuAD-n. To further stress multi-document attention, we construct *SQuAD-n*: we merge n SQuAD examples with distinct gold paragraphs into a single example whose prompt contains all n gold paragraphs (mixed among $\max(k-n, 0)$ distractors) followed by the n questions, labeled $(1) \ldots (n)$; the model must return the n answer spans in order. We use $n \in \{3, 5\}$. We report micro-averaged EM over all (example, sub-question) pairs, and a strict *joint* EM that credits an example only when every sub-answer is an exact match.

RULER. A synthetic long-context benchmark (Hsieh et al., 2024). We use an in-repository reimplementation of its tasks and focus on three subtasks: single needle-in-a-haystack (a magic-

number "needle" line hidden among Paul Graham essays), multi-key NIAH (the needle is hidden among confusable distractor needles), and variable tracking (chains of variable assignments embedded in filler text, where the model resolves the value of a queried variable by following a four-hop chain). Unlike the QA and tool-use tasks, RULER has no natural chunks and no distractor pool; following Jafari et al. (2025) we chunk each input into consecutive 256-token blocks, so the "pool size" is $\lceil \text{context length}/256 \rceil$ and grows with the context. A block is treated as gold if its token span overlaps a needle sentence (NIAH) or an assignment in the queried chain (variable tracking). We cap the training context at 2048 tokens (8 chunks) and evaluate at context lengths $\{1, 2, 4, 8, 16, 32, 64, 128\}$K tokens over 200 examples per (subtask, length).

C.2 Models

Architectures. Our SSM is a 1.5B-parameter Mamba-2 model (Dao & Gu, 2024): 32 Mamba-2 mixer layers (with no interleaved attention or MLP blocks), hidden size 2048, RMSNorm, and no positional embedding. Each mixer uses a state-expansion factor of 2 (inner dimension 4096) partitioned into 64 heads of head dimension 64, state dimension 128, 8 groups, and a convolution kernel of width 4. The recurrent state is therefore $4096 \times 128 = 524{,}288$ elements per layer, or $\approx$16.8M elements (33.6 MB in bf16) across all 32 layers — this is the size of a single cached state encoding. Our Transformer is a 1.5B-parameter LLaMA-style model: 20 layers, hidden size 2048, SwiGLU feed-forward width 5504, 16 attention heads with full multi-head attention, rotary position embeddings, and RMSNorm. The two models are matched in parameter count ($\approx$1.5B). Both are pretrained on the Nemotron-CC corpus (Su et al., 2025) (on the order of 300B tokens) with the same optimizer schedule (AdamW, peak learning rate 3×10^{-4}, cosine decay), share the Nemotron-Nano tokenizer, and run in bf16. We deliberately give the Transformer rotary embeddings so that it can process contexts longer than those seen in training, so that the sequential-Transformer baseline is given a fair chance at length generalization.

Weighted-souping read-out. We augment the SSM with a lightweight read-out head that produces the key and query vectors of Section 4. Concretely, we read the last-token *residual* activation at the final decoder block and pass it through two learned linear projections $W_k, W_q \in \mathbb{R}^{d \times d'}$ with $d = 2048$ and $d' = 128$ (equivalently, a single projection to $\mathbb{R}^{2d'}$ split into a key half and a query half). The two projections are separate parameters, initialized to identical values ($W_k = W_q$) so the head begins as the parameter-free cosine ranker over the base model's activations; they are untied thereafter and trained independently. The head is the only component initialized from scratch; it is trained jointly with the base model but placed in its own optimizer group at learning rate base LR $\times$ `lr_scale`, with `lr_scale` $= 10$.

Cosine normalization. The weight assigned to chunk i is $\sigma(\text{score}_i)$, where the score is a dot-product between the chunk key $\mathbf{k}_i$ and the query vector $\mathbf{u}$. Rather than the usual $1/\sqrt{d}$ dot-product scaling, we ℓ_2-normalize the key and query and scale their cosine similarity by a learnable inverse temperature γ: $\text{score}_i = \gamma \frac{\mathbf{k}_i \cdot \mathbf{u}}{\|\mathbf{k}_i\| \, \|\mathbf{u}\|}$. We initialize $\gamma = 14$ and cap it at 100. Because the pre-nonlinearity logits are bounded to $[-\gamma, \gamma]$ regardless of pool size, the weighting stays well-conditioned as k grows, which we found important for generalizing to pool sizes far beyond the training distribution. Our headline weighting function σ is the softmax.

Sparse souping. Sparse souping keeps only the top-N weights and zeroes the rest; with softmax we apply the mask before the nonlinearity so the surviving weights still sum to one. It is applied at inference time and reuses the softmax checkpoints unchanged. We use $N_{top} = 4$ for the document-QA and BFCL figures and $N_{top} = 20$ for RULER; the larger N_{top} is needed for variable tracking, where gold information is spread across many chunks (we sweep N_{top} in Figure 5).

Auxiliary routing loss. We optionally supervise the weights toward the known gold chunks with an auxiliary loss, $\mathcal{L} = \mathcal{L}_{\text{SFT}} + \lambda \mathcal{L}_{\text{aux}}$, where $\mathcal{L}_{\text{aux}}$ is a cross-entropy over the k chunks under softmax weighting, computed from the same scores used to combine the states. We set $\lambda = 1$ for RULER and tool use, where the gold chunks are reliably identifiable, and $\lambda = 0$ for document QA, where the head learns the weighting purely from the downstream SFT signal.

Souping finetuning. Finetuning is necessary to learn the souping format (Jafari et al., 2025). All souping models are finetuned from the pretrained base with the softmax weighting, residual read-out, and cosine normalization above. For document QA we train for 3 epochs (SQuAD, HotpotQA) or 5 epochs (MuSiQue) at base learning rate 5×10^{-6} with no auxiliary loss, from the train splits included with the benchmarks. For RULER we train for 6000 steps at base learning rate 10^{-5} with $\lambda = 1$, chunked at 256 tokens. For tool use we *warm-start* from a tool-calling-SFT checkpoint already finetuned on the Nemotron Agentic SFT split (NVIDIA, 2025), and train for an additional one epoch at base learning rate 10^{-5} with $\lambda = 1$ to learn the souping format. In every case the read-out head uses the $10\times$ learning-rate multiplier described above. The sequential baselines are finetuned on the concatenated input for matched steps at learning rate 10^{-5} (QA/RULER) or 5×10^{-5} (tool use), so that all models see matched data and compute.

Decoding and evaluation. All models decode greedily. For tool use, decoding is constrained by a tokenizer-level logit mask that enforces the tool-call envelope (`<TOOLCALL>[ ... ]</TOOLCALL>`); the constraint is applied identically to the souping arms and the sequential baselines. The QA and RULER tasks decode without constraints. Evaluation pool sizes, context lengths, and example counts are as given in Appendix C.1.

C.3 Computational benchmark

The timings in Figure 3 are measured on a single AWS `p5.48xlarge` node ($8\times$ H100). We compare four inference paths: a sequential Transformer, a sequential Mamba, full weighted souping, and sparse souping. For the souping paths, we time the encoding of a 128-token query and the computation of the softmax weights, while the chunk state encodings themselves are *retrieved* rather than recomputed, from one of three memory tiers: GPU high-bandwidth memory (HBM), system RAM, or system disk. Each cached Mamba-2 encoding is the 33.6 MB bf16 state described above, so retrieval cost is dominated by the bandwidth of the tier in which the encodings live; retrieval from HBM is far faster than from RAM, which in turn is faster than from disk, and in the worst case (short chunks retrieved from disk) retrieval can be slower than re-encoding.