Lecture 4: Attention Alternatives and Mixtures of Experts

Language Modeling from Scratch · lecture notes · rebuilt for NVIDIA Blackwell, late 2026

Section 1: Two Scaling Bottlenecks in Dense Transformers and the Ceiling of Basic Attention Optimizations

A standard dense Transformer block consists of two sublayers: multi-head self-attention and a dense feed-forward network (FFN). When language models operate at modest context lengths and modest parameter budgets, both sublayers scale manageably. Pushing either context length or total knowledge capacity by orders of magnitude exposes two distinct structural bottlenecks.

The first bottleneck lives inside self-attention. Context windows have grown by three orders of magnitude in six years: historical baselines such as GPT-3 (2020) operated at a few thousand tokens, whereas late-2026 frontier models—including DeepSeek-V4.1-Flash, Kimi K3, and MiniMax-M3—treat a 1M-token context window as standard. Recall that for a sequence of length n and model hidden dimension d, standard softmax attention computes pairwise query-key dot products across every token pair in the sequence. During prefill, this scales as $O(n^2 d)$ floating-point operations (FLOPs), whereas the dense FFN processes each token independently and scales linearly in sequence length as $O(nd^2)$. During autoregressive decode, generating each new token requires reading the stored keys and values of every preceding token. That stored tensor—the KV cache—grows linearly as $O(n)$ per sequence and turns generation memory-bandwidth-bound, because moving gigabytes of cached keys and values from high-bandwidth memory (HBM) takes far longer than the arithmetic itself.

The second bottleneck lives inside the dense FFN. In a dense Transformer, every single token passes through every weight of the FFN. Increasing the model's total parameter count to store more world knowledge directly increases the per-token FLOPs required for both training and inference. We will break that rigid coupling in Part 2 using Mixtures of Experts (MoE); first, we confront the sequence-length bottleneck in Part 1.

To visualize why attention overwhelms the FFN at long context, picture a stacked latency plot whose horizontal axis measures sequence length n running from 16K to 1M tokens and whose vertical axis measures step latency. The bottom shaded region represents the FFN cost, $O(nd^2)$, rising along a gentle straight line with slope proportional to d^2. Stacked directly on top of that linear wedge sits a second shaded region representing standard attention, $O(n^2 d)$, whose upper boundary curves sharply upward as a parabola. At short lengths near the left edge of the plot, the linear FFN region accounts for most of the step latency; as n crosses beyond d and approaches 1M tokens on the right edge, the parabolic attention region expands to dominate nearly the entire vertical height of the chart.

> **WORKED EXAMPLE**

Consider a Transformer layer with hidden dimension $d = 4{,}096 = 2^{12}$ and compare the dominant scaling terms $n^2 d$ (attention) and nd^2 (FFN) as the sequence length n increases from $16\text{K} = 2^{14}$ to $1\text{M} = 2^{20}$:

1. **At $n = 16{,}384 = 2^{14}$:**
   - Attention term: $n^2 d = (2^{14})^2 \cdot 2^{12} = 2^{40} \approx 1.10 \times 10^{12}$.
   - FFN term: $nd^2 = 2^{14} \cdot (2^{12})^2 = 2^{38} \approx 2.75 \times 10^{11}$.
   - Ratio: $\dfrac{n^2 d}{nd^2} = \dfrac{n}{d} = \dfrac{2^{14}}{2^{12}} = 4$.
2. **At $n = 1{,}048{,}576 = 2^{20}$:**
   - Attention term: $n^2 d = (2^{20})^2 \cdot 2^{12} = 2^{52} \approx 4.50 \times 10^{15}$.
   - FFN term: $nd^2 = 2^{20} \cdot (2^{12})^2 = 2^{44} \approx 1.76 \times 10^{13}$.
   - Ratio: $\dfrac{n^2 d}{nd^2} = \dfrac{n}{d} = \dfrac{2^{20}}{2^{12}} = 256$.

Multiplying the sequence length by $64\times$ multiplies FFN compute by $64\times$, but multiplies attention compute by $64^2 = 4{,}096\times$. At 1M tokens, the quadratic attention term outstrips the FFN term by a factor of 256.

Before redesigning the attention mathematics itself, engineers reached for two foundational tools: **local + global interleaving** and **IO-aware kernels**.

DEFINITION

Sliding-window (local) attention restricts each token at position t to attend only to the w most recent preceding tokens within a fixed band width $w \ll n$, reducing per-layer prefill compute from $O(n^2 d)$ to $O(nwd)$ and bounding that layer's KV cache to $O(w)$ entries.

DEFINITION

Local + global interleaving is a layer-stacking pattern in which most layers perform cheap sliding-window local attention while periodic layers perform full global attention to transmit information across arbitrary distances in the sequence.

Visualize the two architectural components of this basic toolkit side by side. On the left is an $n \times n$ attention matrix for sliding-window attention: the upper-right triangle is masked out for causality, and only a narrow diagonal stripe of width $w \ll n$ immediately below the main diagonal is active; every entry further into the past is zero. On the right is a vertical stack of layers showing an interleaved Local + Global architecture: a Global Attention layer sits at the bottom, followed upward by three consecutive Local Attention layers, capped by another Global Attention layer, with the residual stream flowing straight up from one block to the next. Beneath both diagrams sits the systems engineering layer: **FlashAttention**, an IO-aware kernel that tiles blocks of Q, K, and V inside fast on-chip SRAM so that the full $n \times n$ attention score matrix is never materialized in HBM.

COMMON MISCONCEPTION

You might think FlashAttention solves the long-context scaling problem because it eliminates out-of-memory errors and speeds up attention dramatically, but that breaks because FlashAttention only eliminates $O(n^2)$ HBM memory storage and memory traffic—it still computes every single one of the $O(n^2)$ query-key dot products and value weightings. Similarly, interleaving local and global layers reduces the constant factor by running global attention on fewer layers, yet every remaining global layer still executes $O(n^2)$ prefill FLOPs and caches $O(n)$ KV pairs.

At 1M tokens, reducing a quadratic term by a small constant factor is not enough. To achieve asymptotic savings in both prefill FLOPs and decode KV footprint while preserving long-range recall, we must change the sequence mixer itself—either by reformulating attention into linear/recurrent layers or by introducing dynamic sparse attention.

KEY IDEA

- Dense Transformers face two distinct bottlenecks: attention scales quadratically in prefill ($O(n^2 d)$) and linearly in decode KV cache ($O(n)$), while dense FFNs tie total parameter count directly to per-token FLOPs.
- Basic mitigations—sliding-window interleaving and IO-aware kernels like FlashAttention—improve utilization and eliminate HBM $n \times n$ materialization, but any remaining dense global attention layer still pays $O(n^2)$ FLOPs and $O(n)$ KV memory.

✎ Check Your Understanding

1. **Yes or No with justification:** Does replacing a naive PyTorch attention implementation with FlashAttention change the asymptotic prefill FLOP complexity of a global attention layer from $O(n^2 d)$ to $O(nd^2)$?
   Answer: No. FlashAttention tiles Q, K, V in SRAM to avoid materializing the $n \times n$ score matrix in HBM and prevent out-of-memory failures, but it still computes all $O(n^2)$ pairwise interactions, so its prefill FLOP complexity remains $O(n^2 d)$.
2. **Quick computation:** If a model's context length increases by a factor of 10 (holding d fixed), by what factor does standard attention prefill compute grow compared to dense FFN compute?
   Answer: Standard attention prefill compute scales as $O(n^2 d)$ and grows by $10^2 = 100\times$, whereas dense FFN compute scales as $O(nd^2)$ and grows by $10\times$.

Section 2: Linear Attention and Parallel–Serial Duality

Why does standard attention force us to pay $O(n^2)$ FLOPs in the first place? Write standard attention for query and key matrices $Q, K \in \mathbb{R}^{n \times d_k}$ and value matrix $V \in \mathbb{R}^{n \times d_v}$ as

$$\mathrm{Attn}(Q, K, V) = \rho(QK^\top)V$$

where $\rho(\cdot)$ denotes the row-wise softmax (including the $1/\sqrt{d_k}$ scaling and causal mask). Look closely at the parentheses. Because the nonlinear softmax ρ wraps the product $QK^\top$, we are forced to compute $QK^\top \in \mathbb{R}^{n \times n}$ first, apply ρ elementwise across every row, and only then multiply that $n \times n$ matrix by $V \in \mathbb{R}^{n \times d_v}$. The first matrix multiplication costs $n^2 d_k$ multiply-adds and the second costs $n^2 d_v$, giving a total cost of $n^2 d_k + n^2 d_v$—strictly quadratic in n.

Matrix multiplication itself, however, is associative. If we remove the softmax nonlinearity—setting ρ to the identity—or replace it with a factorized kernel feature map $\phi(Q)\phi(K)^\top$ (introduced by Shen et al., 2018, and Katharopoulos et al., 2020, and closely linked to fast weight programmers), the parentheses can move from the left to the right:

$$(QK^\top)V = Q(K^\top V)$$

> **DEFINITION**
> **Linear attention** removes the row-wise softmax wrapping $QK^\top$ (replacing it with the identity or a factorizable kernel feature map ϕ) so that $(QK^\top)V$ can be re-associated as $Q(K^\top V)$, reducing sequence-mixing complexity from quadratic $O(n^2 d)$ to linear $O(n d_k d_v)$.

Compare the matrix shapes in the two parenthesizations visually. On the left side—computing $(QK^\top)V$—a tall thin matrix Q ($n \times d_k$) multiplies a short wide matrix $K^\top$ ($d_k \times n$) to produce a massive square $n \times n$ intermediate matrix $QK^\top$—the quadratic bottleneck—which then multiplies the tall thin matrix V ($n \times d_v$). On the right side—computing $Q(K^\top V)$—we group $K^\top$ ($d_k \times n$) with V ($n \times d_v$) first. The long sequence dimension n contracts immediately, yielding a compact $d_k \times d_v$ matrix $K^\top V$ whose size does not depend on n at all. Multiplying Q ($n \times d_k$) by that compact $d_k \times d_v$ matrix produces the $n \times d_v$ output. Both products cost $n d_k d_v$ multiply-adds each, for a total cost of $2 n d_k d_v$—linear in n.

In an autoregressive language model, attention must be causal: token t can only attend to positions $1, 2, \ldots, t$, not to future tokens $t+1, \ldots, n$. Writing the t-th row of Q, K, and V as column vectors $q_t, k_t \in \mathbb{R}^{d_k}$ and $v_t \in \mathbb{R}^{d_v}$, the causal output vector $y_t \in \mathbb{R}^{1 \times d_v}$ (or written as a row vector $q_t^\top S_t$) is

$$y_t = \sum_{i=1}^{t} (q_t^\top k_i)\, v_i^\top = q_t^\top \left(\sum_{i=1}^{t} k_i v_i^\top \right)$$

Now define the running prefix sum of key-value outer products as a matrix-valued hidden state $S_t \in \mathbb{R}^{d_k \times d_v}$:

$$S_t = S_{t-1} + k_t v_t^\top \qquad \text{and} \qquad y_t = q_t^\top S_t$$

Let us break this pair of equations apart term by term:

- $k_t v_t^\top \in \mathbb{R}^{d_k \times d_v}$: the outer product of the current token's key vector k_t ($d_k \times 1$) and transposed value vector $v_t^\top$ ($1 \times d_v$). This rank-1 matrix encodes the association between key k_t and value v_t.
- $S_{t-1} \in \mathbb{R}^{d_k \times d_v}$: the accumulated associative memory matrix from all previous tokens $1, \ldots, t-1$ (starting from $S_0 = 0$).
- $S_t = S_{t-1} + k_t v_t^\top$: adds the new token's key-value outer product into the state matrix. Notice that the matrix dimensions remain fixed at $d_k \times d_v$ regardless of how large t becomes.
- $y_t = q_t^\top S_t \in \mathbb{R}^{1 \times d_v}$: reads from the memory matrix using the current query $q_t^\top$ ($1 \times d_k$). Expanding $q_t^\top S_t = \sum_{i=1}^{t} (q_t^\top k_i) v_i^\top$ shows that this single matrix-vector product retrieves a weighted combination of all past values v_i, each weighted by the dot-product similarity $q_t^\top k_i$.

As a conceptual unit, causal linear attention is a **matrix-valued recurrent neural network (RNN)** whose hidden state S_t stores the entire prefix history as a sum of outer products. Picture its dataflow diagram: the state block S_{t-1} enters a summation node from the left; from below, the current token's outer product $k_t v_t^\top$ enters the same summation node; the sum exits to the right as the updated state S_t. From S_t, one arrow goes upward to compute the read output $y_t = q_t^\top S_t$, while a second arrow carries S_t rightward to step $t+1$. Because S_t always has fixed shape $d_k \times d_v$, there is no growing KV cache.

DEFINITION

Parallel–serial duality is the property that causal linear attention can be evaluated either in its parallel matrix form during training—saturating GPU Tensor Cores across the sequence—or in its mathematically equivalent serial recurrent form during autoregressive decode, where each new token costs $O(1)$ memory and $O(1)$ compute with respect to context length.

WORKED EXAMPLE

Verify parallel–serial duality on a 3-token sequence ($n = 3$) with $d_k = 2$ and $d_v = 2$. Let:

$$Q = \begin{pmatrix} 1 & 1 \\ 2 & 1 \\ 1 & -1 \end{pmatrix}, \qquad K = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{pmatrix}, \qquad V = \begin{pmatrix} 2 & 1 \\ 3 & -1 \\ 0 & 4 \end{pmatrix}$$

1. **Parallel causal matrix form** $Y = \big(M \odot (QK^\top)\big)V$, where M is the lower-triangular causal mask:

$$QK^\top = \begin{pmatrix} 1 & 1 & 2 \\ 2 & 1 & 3 \\ 1 & -1 & 0 \end{pmatrix} \implies M \odot (QK^\top) = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 1 & -1 & 0 \end{pmatrix}$$

Multiplying by V:

$$Y = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 1 & -1 & 0 \end{pmatrix} \begin{pmatrix} 2 & 1 \\ 3 & -1 \\ 0 & 4 \end{pmatrix} = \begin{pmatrix} 2 & 1 \\ 7 & 1 \\ -1 & 2 \end{pmatrix}$$

2. **Serial recurrent form** starting from $S_0 = \mathbf{0}_{2\times 2}$:
   - **Step $t = 1$:** $k_1 v_1^\top = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \begin{pmatrix} 2 & 1 \end{pmatrix} = \begin{pmatrix} 2 & 1 \\ 0 & 0 \end{pmatrix}$, so $S_1 = \begin{pmatrix} 2 & 1 \\ 0 & 0 \end{pmatrix}$. Output: $y_1 = \begin{pmatrix} 1 & 1 \end{pmatrix} S_1 = \begin{pmatrix} 2 & 1 \end{pmatrix}$.
   - **Step $t = 2$:** $k_2 v_2^\top = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \begin{pmatrix} 3 & -1 \end{pmatrix} = \begin{pmatrix} 0 & 0 \\ 3 & -1 \end{pmatrix}$, so $S_2 = S_1 + k_2 v_2^\top = \begin{pmatrix} 2 & 1 \\ 3 & -1 \end{pmatrix}$. Output: $y_2 = \begin{pmatrix} 2 & 1 \end{pmatrix} S_2 = \begin{pmatrix} 7 & 1 \end{pmatrix}$.
   - **Step $t = 3$:** $k_3 v_3^\top = \begin{pmatrix} 1 \\ 1 \end{pmatrix} \begin{pmatrix} 0 & 4 \end{pmatrix} = \begin{pmatrix} 0 & 4 \\ 0 & 4 \end{pmatrix}$, so $S_3 = S_2 + k_3 v_3^\top = \begin{pmatrix} 2 & 5 \\ 3 & 3 \end{pmatrix}$. Output: $y_3 = \begin{pmatrix} 1 & -1 \end{pmatrix} S_3 = \begin{pmatrix} -1 & 2 \end{pmatrix}$.

Every row matches identically: $(2, 1)$, $(7, 1)$, and $(-1, 2)$. In the recurrent form, generating token $t = 1{,}000{,}000$ would still maintain only a single 2×2 matrix S_t, whereas standard attention would store all $999{,}999$ prior keys and values.

COMMON MISCONCEPTION

You might think that for causal language modeling, you can simply compute one global $K^\top V$ matrix in parallel and multiply $Q(K^\top V)$, but that breaks because a single global $K^\top V$ sums $k_i v_i^\top$ over all $i = 1, \ldots, n$, leaking future tokens $i > t$ into y_t. Causal linear attention requires prefix states $S_t = \sum_{i=1}^{t} k_i v_i^\top$, which in practice is parallelized across sequence chunks (combining intra-chunk parallel matrix products with inter-chunk state propagation) so training still achieves linear $O(n)$ scaling and high GPU Tensor Core utilization.

Plain linear attention still has a severe flaw: $S_t = S_{t-1} + k_t v_t^\top$ adds every token's outer product forever with weight 1, never forgetting stale information as the context grows. A first step toward forgetting is **RetNet**, which multiplies the prior state by a fixed scalar decay $\gamma \in (0, 1)$:

$$S_t = \gamma S_{t-1} + k_t v_t^\top$$

With static exponential decay, tokens from Δ steps ago are weighted by γ^Δ. However, a fixed scalar γ forgets indiscriminately: it decays a crucial variable definition from 500 tokens ago at the exact same rate as filler punctuation. To keep what matters and discard what does not, the recurrence must become input-dependent.

KEY IDEA

- Removing softmax allows $(QK^\top)V = Q(K^\top V)$, cutting attention compute from $O(n^2 d)$ to $O(n d_k d_v)$ and turning causal linear attention into a matrix-valued RNN with fixed state size $d_k \times d_v$.
- Parallel–serial duality gives fast parallel matrix training alongside $O(1)$-per-step decode, but pure accumulation never forgets and static decay (RetNet) forgets indiscriminately.

✎ Check Your Understanding

1. **Fill-in-the-blank:** In causal linear attention $S_t = S_{t-1} + k_t v_t^\top$, the hidden state S_t has fixed dimensions _____ regardless of the sequence length t, which makes decode memory and compute _____ per generated token.
   Answer: $d_k \times d_v$; $O(1)$ (constant).
2. **What would happen if** you set $\gamma = 0.5$ in RetNet ($S_t = \gamma S_{t-1} + k_t v_t^\top$) and needed to recall a fact stated 100 tokens earlier?
   Answer: The contribution of that token's outer product $k_{t-100} v_{t-100}^\top$ would be attenuated by $0.5^{100} \approx 7.9 \times 10^{-31}$, effectively erasing it because static decay forgets every past token uniformly regardless of relevance.

Section 3: Data-Dependent Gating and Selective Erasure: Mamba-2 and Gated DeltaNet

Static decay fails because the model cannot look at the current token x_t to decide whether to retain its memory or reset it. **Mamba-2** generalizes linear attention and RetNet by making the decay gate a learned function of the current input, $\gamma_t = f(x_t)$, and adding a direct skip projection D:

$$S_t = \gamma_t S_{t-1} + k_t v_t^\top, \qquad y_t = q_t^\top S_t + v_t^\top D, \qquad \text{with } \gamma_t = f(x_t)$$

Break down the new components relative to plain linear attention:

- $\gamma_t = f(x_t)$: a data-dependent scalar decay gate computed from the current token representation x_t. When x_t signals a topic shift or document boundary, the model can drive γ_t close to 0 to clear old state; when x_t is part of an ongoing dependency, the model keeps γ_t close to 1 to preserve prior associations.
- $v_t^\top D$: a skip connection projecting the current value directly to the output y_t, bypassing the recurrent state matrix S_t.

DEFINITION

Structured chunkwise duality in Mamba-2 computes the data-dependent gates $\gamma_t = f(x_t)$ in parallel across the full sequence and then splits the sequence into fixed-size chunks, combining parallel intra-chunk matrix multiplications with recurrent inter-chunk state updates to retain both fast training and $O(1)$ decode.

Because every γ_t depends only on x_t (not on S_{t-1}), all gates across the sequence can be computed in parallel before the recurrence runs, so parallel–serial duality still holds. In production systems, Mamba-2 is commonly interleaved with self-attention and MoE layers: the early-2026 baseline Nemotron 3 interleaved Mamba-2, MoE, and a few self-attention layers at roughly a 3:1 ratio, and the late-2026 frontier model Nemotron 3.5 Super / Lightning (Oct 2026) continues this hybrid Mamba-Transformer MoE architecture.

Yet Mamba-2 still suffers from a structural precision limitation: γ_t is a single scalar multiplying the *entire* state matrix S_{t-1}. If a model needs to update the value stored under one specific key—say, overwriting an outdated variable assignment—Mamba-2 has only two blunt options: keep $\gamma_t \approx 1$ and add $k_t v_t^\top$ on top of the stale value (corrupting the slot with a sum of old and new values), or shrink γ_t toward 0 and wipe out every other unrelated key stored in S_{t-1}.

To overwrite one key without harming others, we need key-targeted erasure. This is the core mechanism of **DeltaNet** and its gated extension, **Gated DeltaNet**:

$$S_t = \gamma_t\big(I - \beta_t k_t k_t^\top\big) S_{t-1} + \beta_t k_t v_t^\top, \qquad y_t = q_t^\top S_t$$

with both a data-dependent global decay gate $\gamma_t = f(x_t)$ and a data-dependent write gate $\beta_t = f(x_t)$.

Let us decompose the update hierarchically, starting from the innermost subexpressions:

1. **Retrieving the stale value along key k_t:** Expand the matrix product $(I - \beta_t k_t k_t^\top) S_{t-1} = S_{t-1} - \beta_t k_t (k_t^\top S_{t-1})$. Look at the innermost term $k_t^\top S_{t-1} \in \mathbb{R}^{1\times d_v}$. This is the exact value vector currently stored in S_{t-1} at key direction k_t—call it $v_{\text{old}}^\top = k_t^\top S_{t-1}$.
2. **Targeted erasure and delta-rule overwrite:** Rewriting the entire update (setting $\gamma_t = 1$ momentarily) exposes its geometric meaning:

   $$S_t = S_{t-1} + \beta_t k_t\big(v_t^\top - k_t^\top S_{t-1}\big) = S_{t-1} + \beta_t k_t\big(v_t^\top - v_{\text{old}}^\top\big)$$

   Instead of blindly adding $k_t v_t^\top$, the update computes the error (or "delta") between the desired value $v_t^\top$ and the currently stored value $v_{\text{old}}^\top$, and adds $\beta_t k_t(v_t^\top - v_{\text{old}}^\top)$. This is a single step of online gradient descent on the squared retrieval loss $\frac{1}{2}\|k_t^\top S - v_t^\top\|_2^2$—connecting directly to fast-weight programmers and test-time training (the delta rule).
3. **Orthogonal preservation:** For any other key k_{other} orthogonal to k_t (so $k_{\text{other}}^\top k_t = 0$), left-multiplying $(I - \beta_t k_t k_t^\top) S_{t-1}$ by $k_{\text{other}}^\top$ yields $k_{\text{other}}^\top S_{t-1}$ unchanged. Only the component along k_t is erased.

4. **No-op write gate (β_t):** When token t carries uninformative content, setting $\beta_t = 0$ collapses $(I - \beta_t k_t k_t^\top)$ to I and $\beta_t k_t v_t^\top$ to 0, leaving the state completely untouched (up to γ_t).
5. **Global decay (γ_t):** Multiplying the selectively erased state by γ_t retains Mamba-2's ability to flush or gradually decay the entire memory when context shifts.

As a single conceptual unit, Gated DeltaNet combines global context-level forgetting (γ_t), per-token write skipping ($\beta_t = 0$), and key-targeted memory replacement ($(I - \beta_t k_t k_t^\top)$ followed by $+\beta_t k_t v_t^\top$).

DEFINITION

Gated DeltaNet is a linear-complexity recurrent sequence mixer whose state update $S_t = \gamma_t(I - \beta_t k_t k_t^\top)S_{t-1} + \beta_t k_t v_t^\top$ combines a data-dependent global decay gate γ_t with a key-targeted delta-rule erasure and write gate β_t.

The progression across the four recurrent linear-attention families is summarized below:

Family	State Update Rule (S_t)	What It Controls
RetNet / Lightning	$S_t = \gamma S_{t-1} + k_t v_t^\top$	Static exponential decay
Mamba-2	$S_t = \gamma_t S_{t-1} + k_t v_t^\top$	Input-dependent global decay
DeltaNet	$S_t = S_{t-1}(I - \beta_t k_t k_t^\top) + \beta_t k_t v_t^\top$ *(or left-multiplied form matching $q_t^\top S_t$)*	Targeted key overwrite
Gated DeltaNet	$S_t = \gamma_t(I - \beta_t k_t k_t^\top)S_{t-1} + \beta_t k_t v_t^\top$	Both global decay and targeted erasure

(Note: whether $(I - \beta_t k_t k_t^\top)$ sits on the left or right of S_{t-1} depends solely on whether S_t is stored as $d_k \times d_v$ with read $q_t^\top S_t$, or transposed as $d_v \times d_k$ with read $S_t q_t$; both represent the exact same projection onto the orthogonal complement of k_t.)

WORKED EXAMPLE

Suppose a 2×2 state matrix currently stores two orthogonal key-value associations: key $k_a = \begin{pmatrix}1\\0\end{pmatrix}$ mapped to $v_a = \begin{pmatrix}2\\1\end{pmatrix}$, and key $k_b = \begin{pmatrix}0\\1\end{pmatrix}$ mapped to $v_b = \begin{pmatrix}3\\-1\end{pmatrix}$:

$$S_{t-1} = k_a v_a^\top + k_b v_b^\top = \begin{pmatrix}2 & 1\\3 & -1\end{pmatrix}$$

Verify the initial reads: $k_a^\top S_{t-1} = (2, 1)$ and $k_b^\top S_{t-1} = (3, -1)$. Now token t arrives with key $k_t = k_a = \begin{pmatrix}1\\0\end{pmatrix}$ and a new value $v_t = \begin{pmatrix}10\\0\end{pmatrix}$ meant to replace v_a while leaving v_b intact.

1. **Mamba-2 with $\gamma_t = 0.5$ (and $D = 0$):**

$$S_t = 0.5\begin{pmatrix}2 & 1\\ 3 & -1\end{pmatrix} + \begin{pmatrix}1\\ 0\end{pmatrix}\begin{pmatrix}10 & 0\end{pmatrix} = \begin{pmatrix}1 & 0.5\\ 1.5 & -0.5\end{pmatrix} + \begin{pmatrix}10 & 0\\ 0 & 0\end{pmatrix} = \begin{pmatrix}11 & 0.5\\ 1.5 & -0.5\end{pmatrix}$$

   Reading $k_a^\top S_t = (11, 0.5)$ returns a corrupted mix of the new value and half the stale value, and reading $k_b^\top S_t = (1.5, -0.5)$ reveals that the unrelated association v_b was damaged by half.

2. **Gated DeltaNet with $\gamma_t = 1$ and $\beta_t = 1$:** First form the erasure operator:

$$I - \beta_t k_t k_t^\top = \begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix} - 1 \cdot \begin{pmatrix}1 & 0\\ 0 & 0\end{pmatrix} = \begin{pmatrix}0 & 0\\ 0 & 1\end{pmatrix}$$

   Apply it to S_{t-1} and add $\beta_t k_t v_t^\top$:

$$S_t = 1 \cdot \begin{pmatrix}0 & 0\\ 0 & 1\end{pmatrix}\begin{pmatrix}2 & 1\\ 3 & -1\end{pmatrix} + \begin{pmatrix}10 & 0\\ 0 & 0\end{pmatrix} = \begin{pmatrix}0 & 0\\ 3 & -1\end{pmatrix} + \begin{pmatrix}10 & 0\\ 0 & 0\end{pmatrix} = \begin{pmatrix}10 & 0\\ 3 & -1\end{pmatrix}$$

   Reading $k_a^\top S_t = (10, 0)$ recovers the exact new value, and reading $k_b^\top S_t = (3, -1)$ leaves the unrelated key completely untouched.

3. **Uninformative token with $\gamma_t = 1$ and $\beta_t = 0$:**

$$S_t = 1 \cdot (I - 0)S_{t-1} + 0 = S_{t-1}$$

   The state passes through completely unchanged.

COMMON MISCONCEPTION

You might think that because Gated DeltaNet performs exact key-targeted erasure and overwriting, a pure Gated DeltaNet model can perform exact retrieval across an arbitrarily long 1M-token context, but that breaks because the state $S_t \in \mathbb{R}^{d_k \times d_v}$ has fixed finite capacity. At most d_k keys can be mutually orthogonal in $\mathbb{R}^{d_k}$; once the number of distinct facts to remember exceeds d_k, new keys inevitably have non-zero inner products with old keys, so erasing along k_t partially corrupts previously stored associations.

This finite-rank ceiling of any fixed-size recurrent state creates an unavoidable architectural requirement: to preserve exact multi-key recall over hundreds of thousands of tokens without paying $O(n^2)$ cost at every layer, we must combine recurrent layers with periodic global attention layers—and compress those global layers aggressively.

KEY IDEA

- Mamba-2 adds a data-dependent scalar decay gate $\gamma_t = f(x_t)$ while preserving chunkwise parallel–serial duality, whereas Gated DeltaNet adds a key-targeted erasure factor $(I - \beta_t k_t k_t^\top)$ and write gate $\beta_t = f(x_t)$ to overwrite specific keys cleanly or skip uninformative tokens.
- Every fixed-size $d_k \times d_v$ state matrix hits a hard capacity ceiling when storing more distinct associations than its dimension allows, limiting exact multi-key recall in pure linear models.

✎ Check Your Understanding

1. **What would happen if** Gated DeltaNet sets $\gamma_t = 1$ and $\beta_t = 0$ on a punctuation token x_t?
   Answer: The erasure operator $(I - \beta_t k_t k_t^\top)$ reduces to I and the write term $\beta_t k_t v_t^\top$ becomes 0, so $S_t = S_{t-1}$ and the state is left completely untouched.
2. **Yes or No with justification:** Can Mamba-2's state update $S_t = \gamma_t S_{t-1} + k_t v_t^\top$ erase one stored key-value association without affecting an orthogonal key-value association stored in S_{t-1} ?
   Answer: No. Because γ_t is a single scalar multiplying the entire matrix S_{t-1}, lowering γ_t attenuates every association stored in S_{t-1} equally rather than targeting the direction of k_t.

Section 4: Hybrid Architectures, Gated Attention, and Multi-Head Latent Attention (MLA)

Controlled ablations (ByteDance Seed / UCSC) directly expose the finite-state recall ceiling we just derived: pure linear-attention models drop sharply on exact multi-key recall benchmarks as context grows, whereas **3:1 hybrid architectures**—three cheap linear or Gated DeltaNet layers for every one global attention layer—match full Transformer recall.

Picture a **Repeated 3:1 Hybrid Block** stacked vertically inside a model backbone. Entering from the bottom, the token representations pass upward through Linear / GDN Layer 1 + MoE, then Linear / GDN Layer 2 + MoE, then Linear / GDN Layer 3 + MoE, and finally enter 1× Global Attention (Gated) + MoE FFN at the top of the four-layer macro-block before repeating. Three-quarters of the layers run with $O(1)$ decode state and linear prefill cost, while the periodic global layer provides an exact random-access lookup across the entire context. Earlier designs experimented with even sparser global ratios—for example, the historical 2025 model MiniMax M1 used a 7:1 lightning-linear to full-attention hybrid for linear scaling in context length—before 3:1 emerged as the sweet spot for recall in late-2026 models such as Kimi K3 (3 Kimi Delta Attention layers to 1 global layer) and Qwen3.8-Flash-Next (3 Gated DeltaNet layers to 1 global layer).

Inside those remaining global attention layers, two architectural refinements dramatically improve stability and memory efficiency: **Gated Attention** and **Multi-Head Latent Attention (MLA)**.

DEFINITION

Gated Attention (Qwen, 2025) multiplies the output of the attention sublayer by a learned input-dependent sigmoid gate, improving training stability and allowing larger learning rates without divergence.

Gated Attention is used in the full-attention layers of the early-2026 Qwen3-Next / Qwen3.5 models and in the late-2026 frontier model Kimi K3's gated MLA layers.

Even when only one out of every four layers is global (or when all layers are global, as in GLM-5.3), standard multi-head attention still caches separate full-dimension keys and values for every head at every token. **Multi-Head Latent Attention (MLA)**—pioneered in the historical DeepSeek-V2 and DeepSeek-V3 (Dec 2024) models and adopted in late-2026 frontier models including GLM-5.3 and Kimi K3—compresses the per-token KV cache into a single low-dimensional shared latent vector.

DEFINITION

Multi-Head Latent Attention (MLA) down-projects each token's hidden state $\mathbf{h}_t$ into a compact shared latent vector $\mathbf{c}_t^{KV} = W^{DKV}\mathbf{h}_t$ that is cached at inference, reconstructing per-head content keys $\mathbf{k}_t^C = W^{UK}\mathbf{c}_t^{KV}$ and values $\mathbf{v}_t^C = W^{UV}\mathbf{c}_t^{KV}$ via up-projection matrices that can be algebraically absorbed into the query and output projections.

Trace the MLA dataflow diagram from bottom to top. At the bottom sits the input hidden vector $\mathbf{h}_t$, which splits along three paths:

1. **Query path (left):** $\mathbf{h}_t$ projects down to a compact query latent $\mathbf{c}_t^Q$ (used to save activation memory during training), which then projects up to concatenated content and positional queries $[\mathbf{q}_t^C; \mathbf{q}_t^R]$.
2. **Compressed KV path (right):** $\mathbf{h}_t$ projects down via W^{DKV} into the low-rank latent $\mathbf{c}_t^{KV} = W^{DKV}\mathbf{h}_t$. **This latent vector is what gets cached during inference.** From $\mathbf{c}_t^{KV}$, two up-projections produce the content keys $\mathbf{k}_t^C = W^{UK}\mathbf{c}_t^{KV}$ and content values $\mathbf{v}_t^C = W^{UV}\mathbf{c}_t^{KV}$.
3. **Decoupled RoPE key path (center):** A dashed path projects $\mathbf{h}_t$ directly into a small separate positional key vector $\mathbf{k}_t^R$ that receives Rotary Position Embeddings (RoPE) and is also cached.

All four components—$[\mathbf{q}_t^C; \mathbf{q}_t^R]$, $\mathbf{k}_t^R$, $\mathbf{k}_t^C$, and $\mathbf{v}_t^C$—enter Multi-Head Attention at the top to produce output $\mathbf{u}_t$.

Why does MLA need that third decoupled path $\mathbf{k}_t^R$ instead of simply applying RoPE to $\mathbf{k}_t^C$? The answer lies in **matrix absorption**, the linear-algebraic trick that makes MLA fast during decode. Without RoPE, the inner product between a query $\mathbf{h}W^Q$ and an up-projected key $W^{UK}\mathbf{c}_t^{KV}$ can be rewritten using the adjoint (transpose) of W^{UK}:

$$\left\langle \mathbf{h}W^Q,\ W^{UK}\mathbf{c}_t^{KV} \right\rangle = \left\langle \mathbf{h}(W^Q W^{UK}),\ \mathbf{c}_t^{KV} \right\rangle$$

Let us inspect what this algebraic rearrangement accomplishes. Instead of reading $\mathbf{c}_t^{KV}$ from the cache and multiplying it by W^{UK} to materialize huge per-head keys for every past token, we fold W^{UK} once into the query projection W^Q (forming an effective query matrix $W^Q W^{UK}$) and dot the absorbed

query directly against the tiny cached latent $\mathbf{c}_t^{KV}$! (Similarly, W^{UV} absorbs into the attention output projection W^O.)

Now insert Rotary Position Embeddings (recall that RoPE encodes relative position by multiplying query and key vectors by position-dependent rotation matrices R_q and R_k). With RoPE applied directly to the up-projected key, the inner product becomes:

$$\left\langle R_q(\mathbf{h}W^Q),\ R_k(W^{UK}\mathbf{c}_t^{KV})\right\rangle = (\mathbf{h}W^Q)^\top (R_q^\top R_k) W^{UK}\mathbf{c}_t^{KV}$$

Because the relative rotation matrix $R_q^\top R_k$ sits *between* W^Q and W^{UK}—and varies for every past token position k—W^{UK} can no longer be folded statically into W^Q. Absorption is blocked, which would force us to recompute full keys for every cached token at every decode step.

Decoupled RoPE resolves this conflict cleanly: split each query and key into a RoPE-free content slice ($\mathbf{q}_t^C, \mathbf{k}_t^C$) where matrix absorption works seamlessly, and a tiny separate RoPE slice ($\mathbf{q}_t^R, \mathbf{k}_t^R$) dedicated purely to positional alignment. Because inner products of concatenated vectors simply add, $\langle[\mathbf{q}^C;\mathbf{q}^R],[\mathbf{k}^C;\mathbf{k}^R]\rangle = \langle\mathbf{q}^C,\mathbf{k}^C\rangle + \langle\mathbf{q}^R,\mathbf{k}^R\rangle$, the content term keeps full matrix absorption while the small RoPE slice carries position information. For example, GLM-5.3 uses `kv_lora_rank` $= 512$ for $\mathbf{c}_t^{KV}$ plus a decoupled RoPE dimension of 64 for $\mathbf{k}_t^R$.

WORKED EXAMPLE

Compare the BF16 (2 bytes per parameter) KV cache footprint per token of standard Multi-Head Attention (MHA) against MLA using GLM-5.3's layer count and latent dimensions (78 layers, 64 attention heads, `kv_lora_rank` $= 512$, RoPE dimension $= 64$). Assume a standard MHA head dimension of $d_{\text{head}} = 128$:

1. **Standard MHA (BF16):**
   - Each layer caches both K and V for all 64 heads:

     $$2 \times 64 \text{ heads} \times 128 \text{ dims/head} \times 2 \text{ bytes} = 32{,}768 \text{ bytes/token/layer} = 32 \text{ KiB}$$

   - Across 78 layers:

     $$78 \times 32{,}768 = 2{,}555{,}904 \text{ bytes/token} \approx 2.44 \text{ MiB/token}$$

2. **MLA with Decoupled RoPE (GLM-5.3, BF16):**
   - Each layer caches only the shared latent $\mathbf{c}_t^{KV}$ (512 dims) and the shared RoPE key $\mathbf{k}_t^R$ (64 dims):

     $$(512 + 64) \text{ dims} \times 2 \text{ bytes} = 1{,}152 \text{ bytes/token/layer}$$

   - Across 78 layers:

     $$78 \times 1{,}152 = 89{,}856 \text{ bytes/token} \approx 87.75 \text{ KiB/token} \approx 88 \text{ KB/token}$$

3. **Compression ratio:**

$$\frac{32{,}768}{1{,}152} \approx 28.4\times \text{ smaller KV cache per token}$$

Even with this $28\times$ compression, however, caching $88\ \mathrm{KB/token}$ across a 1M-token sequence still consumes roughly $88\ \mathrm{GB}$ of KV memory for a single sequence—and dense attention over all n latents still executes $O(n^2)$ prefill interactions.

COMMON MISCONCEPTION

You might think MLA solves the $O(n^2)$ prefill FLOP bottleneck because it projects keys and values into a small latent vector, but that breaks because MLA compresses along the *feature* dimension (d), not the *sequence* dimension (n). Every query at position t still attends densely to all t prior latent vectors $\mathbf{c}_1^{KV}, \ldots, \mathbf{c}_t^{KV}$, so without sparsity, MLA remains quadratic in sequence length n.

We have compressed the per-token KV vector with MLA and reduced the number of global layers with 3:1 hybrids. To break the $O(n^2)$ barrier *inside* the global layers themselves while keeping exact token-level retrieval, we turn to dynamic sparse attention and cross-layer compression.

KEY IDEA

- Interleaving 3 cheap recurrent layers per 1 global attention layer matches full Transformer multi-key recall while eliminating 75% of global attention cost; adding a post-attention sigmoid gate (Gated Attention) stabilizes training and allows higher learning rates.
- Multi-Head Latent Attention (MLA) caches a single compressed latent $\mathbf{c}_t^{KV}$ per token and absorbs W^{UK} into W^Q during decode, using a small decoupled RoPE slice $\mathbf{k}_t^R$ so positional rotations do not block matrix absorption.

✎ Check Your Understanding

1. **Yes or No with justification:** In MLA without decoupled RoPE, can we still fold W^{UK} statically into W^Q so that keys never need to be materialized from $\mathbf{c}_t^{KV}$ during decode?
   Answer: No. Applying RoPE inserts position-dependent rotation matrices $R_q^\top R_k$ directly between W^Q and W^{UK}; because that rotation depends on the relative token distance, W^{UK} cannot be absorbed statically into W^Q.

2. **Fill-in-the-blank:** Controlled ablations show that a _____ ratio of cheap linear/GDN layers to global attention layers matches full Transformer multi-key recall, whereas pure linear models drop sharply.
   Answer: 3:1.

Section 5: Dynamic Sparse Attention and the 2026 Three-Axis Attention Landscape

Instead of compressing history into a fixed-size recurrent state—which risks overwriting distant details—consider an alternative strategy for global attention: keep exact token representations available, score all past tokens with an ultra-cheap filter, and run exact attention only on the **top-k** highest-scoring positions. This is **Dynamic Sparse Attention (DSA)**, introduced in DeepSeek-V3.2 (Dec 2025) and adopted in GLM-5.3 (2026) with top-$k = 2{,}048$ over MLA.

> **DEFINITION**
> **Dynamic Sparse Attention (DSA)** is a two-stage global attention mechanism that first scores all past positions using a lightweight, low-precision **lightning indexer** and then runs exact attention solely over the top-k selected past tokens (or latent KV entries).

DSA consists of two mathematical stages:

Stage 1 — Lightning indexer (few heads H^I, ReLU activation, runs in FP8):

$$I_{t,s} = \sum_{j=1}^{H^I} w_{t,j}^I \cdot \mathrm{ReLU}\big(\mathbf{q}_{t,j}^I \cdot \mathbf{k}_s^I\big)$$

Stage 2 — Fine-grained token selection:

$$\mathbf{u}_t = \mathrm{Attn}\big(\mathbf{h}_t,\ \{\mathbf{c}_s \mid I_{t,s} \in \text{Top-}k(I_{t,:})\}\big)$$

Break both equations down term by term:

- $\mathbf{q}_{t,j}^I$ **and** $\mathbf{k}_s^I$: compact indexer query vector (for current token t and indexer head j) and indexer key vector (for past token s). Using very few heads H^I, small dimension, and FP8 arithmetic makes computing $\mathbf{q}_{t,j}^I \cdot \mathbf{k}_s^I$ orders of magnitude cheaper than full multi-head attention.
- $\mathrm{ReLU}(\cdot)$: clamps negative dot products to zero so that a mismatch on one indexer head does not cancel out a strong positive match on another head, and avoids the costly exponentials of softmax.
- $w_{t,j}^I$: a token-dependent scalar weight for indexer head j, combining the H^I head scores into a single scalar relevance index $I_{t,s}$ between current token t and past token s.
- $\{\mathbf{c}_s \mid I_{t,s} \in \text{Top-}k(I_{t,:})\}$: selects only the k past latent KV vectors $\mathbf{c}_s$ whose index scores rank highest among all $s \le t$ (for instance, $k = 2{,}048$ out of $1{,}000{,}000$ tokens).
- $\mathbf{u}_t = \mathrm{Attn}(\mathbf{h}_t, \dots)$: executes exact attention exclusively over those k retrieved entries.

Because a randomly initialized indexer would select garbage tokens from the start, DSA uses **post-hoc adaptation**: first train the base model with dense attention at short context lengths, next freeze the base model and warm up the lightning indexer to imitate the base model's attention distribution, and finally jointly adapt both base model and indexer for long-context training.

> **WORKED EXAMPLE**

> Trace DSA's lightning indexer for $H^I = 2$ heads with head weights $w^I_{t,1} = 0.6$, $w^I_{t,2} = 0.4$, indexer queries $\mathbf{q}^I_{t,1} = (1, 0)$, $\mathbf{q}^I_{t,2} = (0, 1)$, and four past tokens with indexer keys $\mathbf{k}^I_1 = (2, -3)$, $\mathbf{k}^I_2 = (-1, 4)$, $\mathbf{k}^I_3 = (1, 2)$, and $\mathbf{k}^I_4 = (-2, -1)$. Select the top $k = 2$ tokens:
>
> 1. **Token $s = 1$:** $I_{t,1} = 0.6 \cdot \mathrm{ReLU}(2) + 0.4 \cdot \mathrm{ReLU}(-3) = 0.6(2) + 0.4(0) = 1.2$.
> 2. **Token $s = 2$:** $I_{t,2} = 0.6 \cdot \mathrm{ReLU}(-1) + 0.4 \cdot \mathrm{ReLU}(4) = 0.6(0) + 0.4(4) = 1.6$.
> 3. **Token $s = 3$:** $I_{t,3} = 0.6 \cdot \mathrm{ReLU}(1) + 0.4 \cdot \mathrm{ReLU}(2) = 0.6(1) + 0.4(2) = 1.4$.
> 4. **Token $s = 4$:** $I_{t,4} = 0.6 \cdot \mathrm{ReLU}(-2) + 0.4 \cdot \mathrm{ReLU}(-1) = 0.6(0) + 0.4(0) = 0.0$.
>
> Selecting $\text{Top-2}(I_{t,:})$ retrieves $\{\mathbf{c}_2, \mathbf{c}_3\}$ (scores 1.6 and 1.4), ignoring tokens 1 and 4 during exact attention. Notice the protective role of ReLU: had we omitted ReLU, token 2's negative match on head 1 would have pulled its score down to $0.6(-1) + 0.4(4) = 1.0$, causing it to lose to token 1 despite having the strongest single-head match in the sequence.

Late-2026 frontier labs pushed dynamic sparse exact attention further in three complementary directions:

- **MiniMax Sparse Attention (MSA)** (MiniMax-M3, June 2026): replaces MiniMax M1's historical 2025 7:1 linear hybrid with **blockwise sparse attention built on Grouped-Query Attention (GQA)**. A lightweight index branch selects top-k KV *blocks* independently per GQA group, and the main branch executes exact block-sparse attention. Selecting contiguous blocks rather than scattered individual tokens aligns directly with fast matrix-block memory reads; MiniMax reports matching GQA quality on a 109B model while cutting per-token attention compute by **28.4× at 1M context**.
- **Qwen Sparse Attention (QSA)** (Qwen3.8-Flash-Next, Aug 2026): in a 125B total / 6B active model, retains the 3:1 Gated DeltaNet to global-attention backbone, and during continued pretraining replaces the dense full-attention layers with QSA. QSA scores micro-blocks using a compressed lightweight indexer with an indexer budget of **2,048**.
- **DeepSeek-V4 (2026) CSA + HCA Hybrid**: combines three parallel attention branches over a full sequence of up to 1M tokens. Picture the three-branch diagram from bottom to top: the full token sequence feeds simultaneously into three boxes in parallel, whose outputs merge into a single "Local + Sparse Global + Summary" representation:
  1. **Local Window** (left branch): sliding-window attention (window $= 128$ in DeepSeek-V4.1-Flash) handles fine-grained local context cheaply.
  2. **CSA — Compressed Sparse Attention** (middle branch): compresses the KV cache along the *sequence* dimension first, then runs DSA-style top-k sparse selection for targeted needle lookup.
  3. **HCA — Heavily Compressed Attention** (right branch): compresses the sequence KV even more aggressively into a compact global summary, then runs **dense** attention over that compact summary so broad, whole-document context is never missed by top-k pruning.

DEFINITION

CSA (Compressed Sparse Attention) compresses keys and values along the sequence dimension before performing top-k sparse selection, while **HCA (Heavily Compressed Attention)** applies heavier sequence-level compression so dense attention can cheaply attend to a global summary of the entire context.

DeepSeek-V4.1-Flash (Sept 2026) pushes sparse attention and KV compression even further with **CSA2 and static cross-layer KV reuse**. Picture its three-stage horizontal pipeline across consecutive global layers:

1. **Full Layer (left):** Computes fresh global KV representations and runs the indexer to select top-k indices. It passes its KV tensors rightward via a "share KV" link.
2. **Reindex Layer (middle):** Reuses the earlier layer's global KV and indexer keys K^I without recomputing or caching new ones, but runs its own queries against K^I to select *fresh* top-k indices. It passes both KV and indices rightward via a "KV + idx" link.
3. **Reuse Layer (right):** Reuses both the earlier layer's KV tensors *and* the already-selected top-k indices, eliminating both KV caching and indexing work for that layer.

Combined with an ultra-compact attention configuration—1 KV head shared by 64 query heads, `head_dim = 512`, indexer top-$k = 512$, and an **FP4 KV cache**—DeepSeek-V4.1-Flash reduces its total global KV footprint to **890 bytes per token**, roughly one-quarter of DeepSeek-V4-Flash and roughly two orders of magnitude smaller than GLM-5.3's ~88 KB/token BF16 MLA cache across 78 layers.

WORKED EXAMPLE

Compare the total global KV cache memory required for a single 1M-token ($n = 10^6$) sequence in GLM-5.3 versus DeepSeek-V4.1-Flash:

1. **GLM-5.3 (BF16 MLA across 78 layers, ≈ 88 KB/token, exact $89{,}856$ B/token):**

$$89{,}856 \text{ bytes/token} \times 10^6 \text{ tokens} \approx 89.9 \text{ GB per sequence}$$

2. **DeepSeek-V4.1-Flash (CSA2 + cross-layer reuse + FP4 KV, 890 B/token):**

$$890 \text{ bytes/token} \times 10^6 \text{ tokens} = 890{,}000{,}000 \text{ bytes} = 890 \text{ MB per sequence}$$

3. **Reduction factor:**

$$\frac{89{,}856}{890} \approx 101\times$$

Serving a single 1M-token context drops from consuming nearly 90 GB of KV memory to under 1 GB —unlocking high-batch-size long-context serving.

COMMON MISCONCEPTION

You might think Dynamic Sparse Attention completely removes the $O(n^2)$ scaling of prefill because exact attention only touches k tokens per query, but that breaks because the lightning indexer itself must still score every past position (or block) $s \leq t$ to find the top-k set. What DSA achieves is replacing the heavy multi-head, high-dimensional, high-precision $O(n^2)$ attention operation with a vastly cheaper, low-head, FP8 (or sequence-compressed) indexer whose constant factor is tiny, while cutting the heavy exact attention and KV memory reads to $O(k)$ per query.

Stepping back, every late-2026 frontier attention design sits inside a clean **three-axis design space**: (1) a cheap local or recurrent mixer, (2) a minority of global layers (or cross-layer reuse), and (3) sparse or compressed selection inside those global layers:

Model (Release)	Local / Cheap Mixer	Global Layer Ratio	Global Selection / Compression
DeepSeek-V4.1-Flash (Sept 2026)	Sliding window (128)	Cross-layer Full / Reindex / Reuse	CSA2 + FP4 KV (890 B/tok)
Kimi K3 / K3.1 (July 2026)	Kimi Delta Attn (GDN family)	3:1 (3 KDA + 1 global)	Gated MLA
Qwen3.8-Flash-Next (Aug 2026)	Gated DeltaNet	3:1 (3 GDN + 1 global)	QSA micro-block sparse (budget 2048)
GLM-5.3 (2026)	— (all global MLA)	All 78 layers	MLA + DSA top-k (2048)
MiniMax-M3 (June 2026)	— (block-sparse GQA)	All global (sparse)	MSA per-GQA-group top-k blocks
Nemotron 3.5 (Oct 2026)	Mamba-2	Hybrid Mamba-Transformer	Self-attention layers

With the sequence-mixing bottleneck brought under control across all three axes, the other half of the Transformer block now dominates total parameter cost and per-token FLOPs: the dense FFN. Part 2 addresses how to scale model capacity to hundreds of billions or trillions of parameters without paying dense FFN compute on every token.

KEY IDEA

- Dynamic sparse attention (DSA, MSA, QSA, CSA/HCA) preserves exact token-level recall at 1M context by pairing a cheap indexer with top-k exact attention, while cross-layer KV/index reuse and FP4 quantization (CSA2 in DeepSeek-V4.1-Flash) shrink global KV footprint to 890 bytes/token.
- Late-2026 frontier attention combines three axes: cheap local/recurrent mixers, a minority of global layers, and sparse or compressed selection inside those global layers.

✎ Check Your Understanding

1. **Quick distinction:** In DeepSeek-V4.1-Flash's CSA2 cross-layer pipeline, what is the exact difference between a *Reindex Layer* and a *Reuse Layer*?
   Answer: A Reindex Layer reuses the earlier Full Layer's KV cache and indexer keys K^I but computes fresh top-k token indices for its own queries, whereas a Reuse Layer reuses both the earlier KV cache and the earlier top-k indices directly.
2. **Why pair CSA with HCA?** Why does DeepSeek-V4 include a Heavily Compressed Attention (HCA) dense branch alongside Compressed Sparse Attention (CSA)?
   Answer: CSA excels at targeted, needle-in-a-haystack retrieval via top-k selection, while HCA runs cheap dense attention over a heavily compressed sequence summary so broad, global context is not lost when top-k prunes most tokens.

Section 6: Mixtures of Experts — Decoupling Capacity from Compute and Designing the Router

In a dense Transformer block, every token exits self-attention and passes through one large dense FFN. If you double the FFN width to store more factual knowledge, you double the FLOPs spent on every single token during training and decode. A **Mixture of Experts (MoE)** breaks this lockstep coupling by replacing the single large FFN with N parallel expert FFNs and a lightweight learned **router** that selects only $K \ll N$ experts for each token.

DEFINITION

A **Mixture of Experts (MoE)** layer replaces a dense FFN with N parallel expert FFNs and a learned router that dynamically activates only $K \ll N$ experts per token, increasing total parameter count (knowledge capacity) without increasing per-token FLOPs.

Compare the architecture of a Dense Transformer Block and a Sparse MoE Block side by side. On the left (Dense), two tokens—"The" and "Dog"—exit Self-Attention + Residual and both flow straight through the exact same One Large Dense FFN before reaching Add & Norm → Output. On the right (Sparse MoE), both tokens exit Self-Attention + Residual and enter a **Router (Top-K)** block. Above the router sit four expert FFNs (FFN 1 through FFN 4). The router sends "The" to FFN 2 and "Dog" to FFN 1, leaving FFN 3 and FFN 4 completely inactive (greyed out) for those tokens; the active expert outputs then merge in a **Weighted Sum + Residual → Output** stage. Although less common variants apply MoE over attention heads (such as ModuleFormer and JetMoE), standard practice sparsifies the MLP/FFN sublayer.

Empirical results explain why MoEs dominate frontier LLMs: at fixed per-token FLOPs, adding total parameters lowers loss dramatically.

- **Switch Transformer** (Fedus et al., 2022): test loss drops smoothly as expert count increases from 1 to 256 at fixed FLOPs, reaching dense T5-Base perplexity up to **7× faster**.
- **OLMoE** (1.3B active / 6.9B total vs. 1.3B dense): achieves roughly **3× fewer FLOPs/tokens** and **2× faster** wall-clock training on HellaSwag.
- **DeepSeek controlled ablation** (0.2B active / 2.0B total vs. 0.2B dense, matched at 2.9T FLOPs per 2K tokens over 100B tokens): Pile validation loss drops from **2.060 to 1.881**.

Late-2026 frontier models push this total-to-active decoupling to extreme ratios:

2026 Frontier Model	Total / Backbone Params	Active Params / Token	Total : Active
DeepSeek-V4.1-Flash (Sept 2026)	552B backbone	16B decode (8B prefill)	**~34×** (decode)
Kimi K3 / K3.1 (July 2026)	2.8T	104B	**2.8T : 104B**
Qwen3.8-Flash-Next (Aug 2026)	125B	6B	**125B : 6B**

Why did MoEs take years to become the universal default? Historically, two severe hurdles slowed adoption: (1) **multi-node systems complexity**, because hosting hundreds of experts requires many GPUs linked by high-throughput all-to-all token dispatch (Fedus et al., 2022), and (2) **training instability**, where heuristic routing objectives and sensitive router logits triggered loss spikes far worse than dense Transformers (Zoph et al., 2022). Every MoE design navigates three core engineering axes: **the routing function**, **expert sizes and structure**, and **training and systems**. We start with the routing function.

The first routing decision is *who chooses whom*. Consider the token-by-expert affinity matrix (experts as rows, tokens as columns) under three paradigms:

1. **Token Chooses Expert (TC):** Each column (token) independently picks its **Top-K** experts. This is the universal standard in autoregressive LLMs because every token routes on its own without waiting for other tokens; controlled OLMoE ablations confirm that TC outperforms Expert-Choice on both validation loss and HellaSwag.
2. **Expert Chooses Token (EC):** Each row (expert) picks its **Top-C** tokens across the batch. While EC guarantees exact load balance by construction, it can leave some tokens unassigned (dropped) and violates causality across sequence positions.
3. **Global Optimization:** Solves a full-matrix linear assignment problem (e.g., BASE Layers) over the entire batch, which is expensive and requires batch-wide coordination.

Non-learned and RL baselines also exist: **Hash routing** (assigning tokens to experts via a fixed hash function) is a surprisingly strong non-learned baseline, while **RL routing** (Bengio, 2013; Clark et al., 2020) suffers from high gradient variance.

COMMON MISCONCEPTION

You might think Expert-Choice (EC) routing is strictly better than Token-Choice (TC) because every expert automatically receives the exact same number of tokens C, eliminating load imbalance, but that breaks in autoregressive language modeling for two reasons: during causal sequence processing, letting an expert compare tokens across the sequence leaks future token positions into earlier routing decisions, and during single-token autoregressive decode there is no batch of sequence tokens for each expert to choose among.

Under Token-Choice Top-K routing, the MoE layer computes:

$$\mathbf{h}_t^l = \mathbf{u}_t^l + \sum_{i=1}^{N} g_{i,t}\,\mathrm{FFN}_i(\mathbf{u}_t^l), \qquad g'_{i,t} = \begin{cases} s_{i,t} & s_{i,t} \in \mathrm{Top}K\big(\{s_{j,t}\}_{j=1}^{N},\, K\big) \\ 0 & \text{otherwise} \end{cases}$$

Break this down term by term:

- $\mathbf{u}_t^l$: the hidden representation of token t entering the MoE sublayer at layer l (after attention and its residual addition).
- $\mathrm{FFN}_i(\mathbf{u}_t^l)$: the output of the i-th expert FFN evaluated on $\mathbf{u}_t^l$. In practice, we only execute the K experts for which $g_{i,t} \neq 0$.
- $s_{i,t}$: the raw affinity score between token t and expert i, derived from the dot product $\mathbf{u}_t^{l\top}\mathbf{e}_i^l$ between the token vector and expert i's learned router centroid vector $\mathbf{e}_i^l$.
- $g'_{i,t}$: the sparse pre-normalization gate—equal to $s_{i,t}$ if expert i lands in token t's Top-K affinities, and 0 otherwise.
- $g_{i,t}$: the final gating weight multiplying $\mathrm{FFN}_i(\mathbf{u}_t^l)$, which either equals $g'_{i,t}$ directly or is renormalized across the winning K experts.

How should we map raw logits $z_{i,t} = \mathbf{u}_t^{l\top}\mathbf{e}_i^l$ to affinities $s_{i,t}$ and weights $g_{i,t}$? Three variants trace the field's evolution:

- **Softmax before Top-K (historical):** $s_{i,t} = \mathrm{Softmax}_i(\mathbf{u}_t^{l\top}\mathbf{e}_i^l)$ over all N experts, then keep Top-K (used in DeepSeek V1–V2, Grok, early Qwen).
- **Softmax after Top-K (historical):** pick the Top-K raw logits first, then apply softmax solely over those K winners (used in Mixtral, DBRX).
- **Sigmoid + Renormalization (2026 frontier default):** compute independent sigmoid affinities $s_{i,t} = \sigma(\mathbf{u}_t^{l\top}\mathbf{e}_i^l)$, select Top-$K$ (with a load-balancing bias term we will define in Section 8), and renormalize across the selected experts:

$$g_{i,t} = \frac{g'_{i,t}}{\sum_{j=1}^{N} g'_{j,t}}$$

This is standard across DeepSeek-V3 (Dec 2024) through DeepSeek-V4 (2026), Kimi K3, and GLM-5.3.

DEFINITION

Renormalized sigmoid routing scores each expert independently via $s_{i,t} = \sigma(\mathbf{u}_t^{l\top} \mathbf{e}_i^l)$, selects the Top-$K$ experts, and divides their scores by $\sum_{j=1}^{N} g'_{j,t}$ so the active weights sum to 1 without creating zero-sum logit competition across hundreds of non-selected experts.

Why does sigmoid beat softmax when N grows to hundreds of experts? Softmax forces all N experts into a zero-sum competition: raising one non-selected expert's logit shrinks the probabilities of the winning experts. Sigmoid scores every expert independently in $(0, 1)$, eliminating cross-expert interference among the $N - K$ inactive experts, while the final division $\sum_{j=1}^{N} g'_{j,t}$ preserves output activation scale.

WORKED EXAMPLE

Suppose $N = 4$ experts and $K = 2$, and a token produces raw router logits $z = [2.0,\ 1.0,\ -1.0,\ -2.0]$. Compare the three gating rules:

1. **Softmax before Top-K:** Compute $e^{2.0} \approx 7.389$, $e^{1.0} \approx 2.718$, $e^{-1.0} \approx 0.368$, $e^{-2.0} \approx 0.135$, sum $Z = 10.610$. The full softmax distribution is $[0.696,\ 0.256,\ 0.035,\ 0.013]$. Keeping Top-2 yields:

$$g = [0.696,\ 0.256,\ 0,\ 0] \quad (\text{sum} = 0.952 < 1)$$

2. **Softmax after Top-K:** Select Top-2 logits $[2.0, 1.0]$ and apply softmax over just those two:

$$g_1 = \frac{e^{2.0}}{e^{2.0} + e^{1.0}} = \frac{1}{1 + e^{-1}} \approx 0.731, \qquad g_2 = 1 - 0.731 = 0.269$$

$$g = [0.731,\ 0.269,\ 0,\ 0] \quad (\text{sum} = 1.000)$$

3. **Sigmoid + Renormalization:** Compute independent sigmoid scores: $\sigma(2.0) \approx 0.881$, $\sigma(1.0) \approx 0.731$, $\sigma(-1.0) \approx 0.269$, $\sigma(-2.0) \approx 0.119$. Keep Top-2: $g' = [0.881,\ 0.731,\ 0,\ 0]$, whose sum is $0.881 + 0.731 = 1.612$. Renormalizing:

$$g_1 = \frac{0.881}{1.612} \approx 0.547, \qquad g_2 = \frac{0.731}{1.612} \approx 0.453 \implies g = [0.547,\ 0.453,\ 0,\ 0]$$

Now observe what happens if non-selected expert 3's logit rises from -1.0 to $+0.5$ (still third place, still inactive): under Softmax-before-Top-K, Z jumps from 10.610 to 11.891, depressing g_1 from 0.696 down to 0.621 even though expert 3 was never chosen. Under Sigmoid + Renormalization, $g_1 = 0.547$ and $g_2 = 0.453$ do not budge at all.

Now that we know how a token selects K experts out of N, the next design axis determines how to carve up the FFN parameter budget across those experts: should we build a small number of wide experts, or shatter them into hundreds of narrow experts alongside always-on shared experts?

KEY IDEA

- Sparse MoE decouples total knowledge capacity from per-token FLOPs (reaching 2.8T total / 104B active in Kimi K3 and 552B backbone / 16B decode active in DeepSeek-V4.1-Flash), yielding sharply lower loss at fixed compute.
- Token-Choice routing with renormalized sigmoid affinities is the late-2026 standard because it preserves causal independence, avoids zero-sum logit competition across hundreds of experts, and maintains stable output scale.

✎ Check Your Understanding

1. **Quick computation:** In DeepSeek-V4.1-Flash (552B backbone parameters, 16B active during decode and 8B active during prefill), approximately what fraction of backbone parameters is active per token during prefill?
   Answer: $8\text{B}/552\text{B} \approx 1.45\%$ (a total-to-active ratio of $69\times$ during prefill, compared to $\sim 34\times$ during decode).
2. **Yes or No with justification:** Under renormalized sigmoid routing, if an unselected expert's logit increases slightly without entering the Top-K, do the final gating weights $g_{i,t}$ of the selected Top-K experts change?
   Answer: No. Because sigmoid computes each affinity $s_{i,t} = \sigma(\mathbf{u}_t^{l\top}\mathbf{e}_i^l)$ independently and renormalization divides only by the sum of the selected Top-K scores, changes to unselected logits have zero effect on $g_{i,t}$.

Section 7: Expert Granularity, Shared Expert Isolation, and Scaling Trajectories

Early MoE designs such as GShard, Switch Transformer, Mixtral, and Grok kept each expert at the full width of a standard dense FFN—what we call **coarse** experts (fine-grained ratio $= 1$). With coarse experts and $K = 2$, a token can only combine two broad, monolithic experts at a time. Each expert must therefore absorb both specialized domain knowledge *and* general syntactic processing, forcing duplicate copies of common features across nearly every expert.

Two architectural innovations—**fine-grained expert segmentation** and **shared expert isolation**—directly solve this redundancy without adding a single active FLOP.

DEFINITION

Fine-grained expert segmentation (DeepSeekMoE, Dai et al., 2024) splits each coarse expert FFN into m narrower experts (each with $1/m$ the intermediate hidden width) and increases the number of active experts per token from K to mK, preserving identical total parameters and per-token FLOPs while exponentially expanding the number of possible expert combinations from $\binom{N}{K}$ to $\binom{mN}{mK}$.

DEFINITION

Shared expert isolation (DeepSpeed-MoE through DeepSeekMoE) dedicates N_s always-on experts that process every token unconditionally, capturing universal linguistic features so the routed experts can specialize without storing redundant copies of common knowledge.

Visualize the three-panel progression of expert layouts side by side:

- **Panel (a) — Conventional Top-2:** N wide rectangular expert blocks sit above a Router ($K = 2$), which activates two coarse blocks (e.g., experts 1 and N).
- **Panel (b) — Fine-Grained ($2N$, $K = 4$):** Each wide expert is sliced vertically in half, producing $2N$ narrower blocks. The Router ($K = 4$) activates four half-width blocks (e.g., experts 1, 3, 4, and $2N$). Total active width—and therefore active FLOPs—is identical to Panel (a), yet the router can compose much finer combinations of sub-skills.
- **Panel (c) — Fine-Grained + Shared Expert Isolation:** One block on the far left is isolated as an always-on shared expert $S1$ receiving a direct green arrow from every token, bypassing the router completely. The remaining $2N - 1$ narrow routed experts sit under a Router ($K = 3$), keeping total active experts equal to $1 \text{ shared} + 3 \text{ routed} = 4$ half-width experts.

WORKED EXAMPLE

Compare a conventional coarse MoE ($N = 16$, $K = 2$, FFN intermediate width $d_{\text{ff}} = 4{,}096$) against a fine-grained MoE segmented by factor $m = 4$ ($N' = 64$, $K' = 8$, each expert intermediate width $d'_{\text{ff}} = 4{,}096/4 = 1{,}024$):

1. **Total parameter width (proportional to total FFN parameters):**
   - Coarse: $16 \times 4{,}096 = 65{,}536$.
   - Fine-grained: $64 \times 1{,}024 = 65{,}536$ (identical).
2. **Active parameter width per token (proportional to per-token FLOPs):**
   - Coarse: $2 \times 4{,}096 = 8{,}192$.
   - Fine-grained: $8 \times 1{,}024 = 8{,}192$ (identical).
3. **Combinatorial expressiveness (number of distinct active expert subsets):**
   - Coarse: $\binom{16}{2} = \dfrac{16 \times 15}{2} = 120$ combinations.

- Fine-grained: $\binom{64}{8} = \frac{64!}{8!\,56!} = 4{,}426{,}165{,}368 \approx 4.43 \times 10^9$ combinations.

At zero additional FLOPs and zero additional parameters, increasing granularity by $m = 4$ expands the router's compositional repertoire by more than $36\ \text{million}\times$.

COMMON MISCONCEPTION

You might think activating mK fine-grained experts instead of K coarse experts multiplies per-token compute by m, but that breaks because each fine-grained expert's intermediate dimension is divided by m. A linear projection from $d \to d_{\text{ff}}/m \to d$ costs $1/m$ the FLOPs of a full expert, so summing mK narrow experts costs $mK \times (1/m) = K$ coarse experts—exact compute parity.

How do these two ideas hold up under controlled empirical ablations? Both DeepSeekMoE and OLMoE ran head-to-head experiments at fixed total parameters and fixed FLOPs:

- **DeepSeekMoE ablations:** Starting from a GShard-style baseline of 0 shared + top-2 of 16 routed experts, isolating **1 shared expert** (1 shared + top-1 of 15 routed) improves knowledge-heavy benchmarks (TriviaQA and NaturalQuestions). Increasing **granularity** from 1 + top-3 of 31 up to 1 + top-7 of 63 delivers the largest overall gains across HellaSwag, ARC, TriviaQA, and NaturalQuestions.
- **OLMoE controlled ablations:** Fine-grained experts win decisively—64 experts beat 32 experts, which beat 8 experts across C4 validation loss, HellaSwag, and MMLU. Interestingly, OLMoE found **no quality gain from a shared expert**: 32 purely routed experts matched 31 routed + 1 shared across loss and downstream curves.
- **2026 frontier practice:** Despite OLMoE's null result on quality alone, **1 to 2 shared experts are now universal** across DeepSeek, Kimi, GLM, and Qwen3.8. Beyond any knowledge-isolation benefit, shared experts provide a convenient always-on dense compute path that executes locally on every device, overlapping naturally with the all-to-all network communication required for routed experts (which we examine in Section 9).

Tracing MoE routing configurations from 2021 to late 2026 makes both trajectories unmistakable—routed expert counts climbed from 64 (DeepSeek V1) to 896 (Kimi K3.1 Pro), and 1–2 shared experts became universal:

Model	Era	Routed Experts	Active (K)	Shared	Fine-Grained Ratio
GShard / Switch Transformer	2021–22	2048 / 64	2 / 1	0	1
Mixtral / DBRX / Grok	2023–24	8 / 16 / 8	2 / 4 / 2	0	1
DeepSeek V1 / OLMoE	2024	64 / 64	6 / 8	2 / 0	1/4 / 1/8
Llama 4 (Maverick)	Apr 2025	128	1	1	1/2
DeepSeek-V3	Dec 2024	256	8	1	1/14
GLM-5.3	**2026**	**256**	**8**	**1**	—
DeepSeek-V4.1-Flash	**Sept 2026**	**384**	**6**	**1**	—
Qwen3.8-Flash-Next	**Aug 2026**	**512**	**10**	**1**	—
Kimi K3.1 Pro	**July 2026**	**896**	**16**	**2**	—

Designing a fine-grained MoE with hundreds of experts sets up our next obstacle: Top-K routing makes a discrete, non-differentiable choice, and without careful training mechanisms, routers either collapse onto a handful of experts or diverge numerically.

KEY IDEA

- Fine-grained segmentation splits experts into m narrower slices and activates mK of them, delivering exponential growth in combinatorial expressiveness $\binom{mN}{mK}$ at identical FLOPs and parameter count, backed by both DeepSeekMoE and OLMoE ablations.
- Although OLMoE saw no standalone loss gain from shared experts, 1–2 shared experts are standard in 2026 frontier models (256–896 routed experts) both to isolate universal features and to overlap dense compute with all-to-all communication.

✎ Check Your Understanding

1. **Quick computation:** If an MoE layer has $N = 8$ experts with $K = 2$ active, and you apply fine-grained segmentation with $m = 2$, what are the new routed expert count, new active expert count, and new number of combinations $\binom{mN}{mK}$ compared to the original $\binom{8}{2} = 28$?
   Answer: The new configuration has $mN = 16$ routed experts and $mK = 4$ active experts (each at half width), giving $\binom{16}{4} = \frac{16\times15\times14\times13}{4\times3\times2\times1} = 1{,}820$ combinations versus 28 originally.
2. **Fill-in-the-blank:** Beyond capturing universal features, always-on shared experts are retained in late-2026 MoE systems because their dense local computation can overlap naturally with _____ communication.
   Answer: all-to-all.

Section 8: Training MoEs: Load Balancing, Aux-Loss-Free Routing, and Numerical Stability

Training a sparse MoE introduces a nasty positive-feedback loop known as **routing collapse**. Suppose by random initialization Expert 1 happens to receive slightly more tokens than Experts $2, \ldots, N$ in the early steps of training. Because Expert 1 gets more gradient updates, its predictions improve faster, which makes the router send even *more* tokens to Expert 1, starving the remaining experts of gradients altogether. Controlled OLMoE ablations confirm that without load balancing, routing collapses onto roughly 2 experts absorbing 50–100% of all tokens—destroying both hardware utilization and C4/Pile validation loss.

Why can't standard backpropagation fix this automatically? Because Top-K selection is discrete and non-differentiable: an expert that is not selected receives zero gradient from the language-modeling loss. Three families of solutions have been tried:

1. **Reinforcement learning (REINFORCE)** (Clark et al., 2020): treats routing as a stochastic policy. While principled and scaling comparably to S-BASE and Hash routing, its high gradient variance and implementation complexity prevented broad adoption.
2. **Stochastic perturbations:** adds random noise to router logits before Top-K so underused experts get sampled occasionally. Shazeer et al. (2017) added learned Gaussian noise, $H(x)_i = (xW_g)_i + \mathcal{N}(0,1) \cdot \mathrm{Softplus}((xW_{\text{noise}})_i)$, and Switch Transformer (Fedus et al., 2022) applied uniform multiplicative jitter $\mathcal{U}(1-\varepsilon, 1+\varepsilon)$ to router logits. However, Zoph et al. (2022) found that removing jitter improved training stability (from 4/6 stable runs to 3/3) at only a slight quality dip ($-1.755 \to -1.777$).
3. **Heuristic load-balancing objectives and online biases:** what works reliably at scale in practice.

Switch Transformer (Fedus et al., 2022) introduced the classic **auxiliary load-balancing loss** over a batch $\mathcal{B}$ of T tokens and N experts:

$$\mathcal{L}_{\text{aux}} = \alpha \cdot N \sum_{i=1}^{N} f_i \cdot P_i, \qquad f_i = \frac{1}{T} \sum_{x \in \mathcal{B}} 1\{\arg\max p(x) = i\}, \qquad P_i = \frac{1}{T} \sum_{x \in \mathcal{B}} p_i(x)$$

Decompose every piece of this objective:

- f_i: the actual fraction of tokens in batch $\mathcal{B}$ dispatched to expert i (discrete, non-differentiable because of the indicator $1\{\cdot\}$). Note $\sum_{i=1}^{N} f_i = 1$.
- P_i: the average router probability mass allocated to expert i across the batch (smooth and differentiable with respect to router weights). Note $\sum_{i=1}^{N} P_i = 1$.
- $\sum_{i=1}^{N} f_i P_i$: the inner product of the discrete load vector f and the probability vector P. Under uniform routing ($f_i = P_i = 1/N$ for all i), $\sum_i f_i P_i = N \cdot (1/N^2) = 1/N$, so multiplying by N normalizes the minimum value to 1 (giving $\mathcal{L}_{\text{aux}} = \alpha$, where α is a small hyperparameter weight).
- **Why the gradient works despite discrete f_i:** Differentiating $\mathcal{L}_{\text{aux}}$ with respect to an individual token's router probability $p_i(x)$ treats f_i as a constant coefficient:

  $$\frac{\partial \mathcal{L}_{\text{aux}}}{\partial p_i(x)} = \frac{\alpha N}{T} f_i = \frac{\alpha N}{T^2} \sum_{x' \in \mathcal{B}} 1\{\arg\max p(x') = i\}$$

  Heavily loaded experts have large f_i, so gradient descent applies a stronger downward push on $p_i(x)$, shifting probability mass away from overloaded experts toward underused ones.

Historical DeepSeek V1 and V2 generalized this to Top-K' routing with $f_i = \frac{N'}{K'T} \sum_t 1(\text{token } t \text{ selects } i)$ and added a device-level balance loss $\mathcal{L}_{\text{DevBal}}$ aggregating load across experts hosted on the same accelerator.

WORKED EXAMPLE

Evaluate Switch Transformer's $\mathcal{L}_{\text{aux}}$ for $N = 4$ experts, $T = 4$ tokens, and $\alpha = 0.01$ under balanced versus collapsed routing:

1. **Balanced routing:** Each expert gets 1 token ($f = [0.25, 0.25, 0.25, 0.25]$) and average probability $P = [0.25, 0.25, 0.25, 0.25]$:

   $$\mathcal{L}_{\text{aux}} = 0.01 \times 4 \times 4(0.25 \times 0.25) = 0.01 \times 4 \times 0.25 = 0.010$$

   Every expert receives the same tiny gradient $\frac{\partial \mathcal{L}_{\text{aux}}}{\partial p_i(x)} = \frac{0.01 \times 4}{4}(0.25) = 0.0025$.

2. **Collapsed routing onto Expert 1:** All 4 tokens pick Expert 1 ($f = [1, 0, 0, 0]$), with $P = [0.85, 0.05, 0.05, 0.05]$:

   $$\mathcal{L}_{\text{aux}} = 0.01 \times 4 \times (1 \cdot 0.85 + 0 + 0 + 0) = 0.034 \quad (3.4\times \text{ higher})$$

   Now the gradients separate sharply:

$$\frac{\partial \mathcal{L}_{\text{aux}}}{\partial p_1(x)} = \frac{0.01 \times 4}{4}(1) = 0.010, \qquad \frac{\partial \mathcal{L}_{\text{aux}}}{\partial p_2(x)} = 0$$

Gradient descent pushes down Expert 1's probability while leaving Experts 2–4 untouched, breaking the collapse.

Auxiliary losses work, but they create an uncomfortable tug-of-war: a large α forces uniform load at the expense of interfering with language-modeling gradients and hurting model quality, whereas a small α allows load imbalance to creep back in. DeepSeek-V3 (Dec 2024) broke this trade-off with **aux-loss-free balancing via per-expert biases** (`noaux_tc`):

$$g'_{i,t} = \begin{cases} s_{i,t} & s_{i,t} + b_i \in \text{Top}K\left(\{s_{j,t} + b_j\}_{j=1}^{N_r},\, K_r\right) \\ 0 & \text{otherwise} \end{cases}$$

DEFINITION

Aux-loss-free balancing (`noaux_tc`) maintains a non-learned per-expert bias b_i updated online after each training step (decreased when expert i is overloaded, increased when underloaded) that shifts Top-K selection without altering the gating weight $s_{i,t}$ or adding gradient interference to the language-modeling loss.

After each training step, we inspect how many tokens each routed expert received: if expert i was overloaded, we decrease b_i by a small step; if underloaded, we increase b_i. DeepSeek-V3 retains only a tiny complementary **sequence-wise auxiliary loss** ($\mathcal{L}_{\text{Bal}}$) to prevent extreme imbalance within any single sequence. By 2026, sigmoid gates + renormalized Top-K + bias balancing (`noaux_tc`) became the standard across DeepSeek, Kimi, and GLM.

COMMON MISCONCEPTION

You might think adding a load-balancing bias b_i distorts the router's weighted combination of expert outputs just as an auxiliary loss distorts router gradients, but that breaks because of where b_i sits in the piecewise formula: b_i enters *only* inside the $\text{Top}K$ membership test $s_{i,t} + b_i$, while the value actually assigned to $g'_{i,t}$ is the unbiased affinity $s_{i,t}$.

Even with balanced routing, MoEs still suffer catastrophic loss spikes if trained naively in BF16. Zoph et al. (2022) pinpointed the numerical cause: **exponentials in the router amplify tiny BF16 roundoff errors**. Recall that BF16 has only 7 mantissa bits, so as logit magnitudes grow large, the spacing between representable BF16 numbers becomes coarse—large enough to visibly distort e^{z_i}.

WORKED EXAMPLE

Suppose a softmax router has 11 experts whose true logits are ten copies of 128.0 and one winning logit of 128.5:

1. **Exact calculation (subtracting the max 128.5 for stability):**

$$p_{\text{exact}} = \frac{e^0}{e^0 + 10\,e^{-0.5}} = \frac{1}{1 + 10(0.6065)} \approx \frac{1}{7.065} \approx 0.142$$

2. **In BF16 precision:** Between $128 = 2^7$ and $256 = 2^8$, 7 mantissa bits give a representable spacing of $2^{7-7} = 1.0$. Therefore 128.5 rounds to 128.0, making all 11 logits identical:

$$p_{\text{BF16}} = \frac{e^0}{e^0 + 10\,e^0} = \frac{1}{11} \approx 0.091$$

3. **Relative error:**

$$\frac{|0.142 - 0.091|}{0.142} \approx 36\% \text{ error}$$

A single 0.5-bit rounding error in BF16 produces a 36% jump in the router's gating weight—and when multiplied across deep layers and backward passes, this triggers violent loss spikes.

Two lightweight fixes completely eliminate this instability:

1. **Fix 1 — FP32 router:** Upcast the router logits to full 32-bit floating point immediately before gating, at negligible compute overhead because the router projection is tiny compared to the experts:

```
router_logits = (x @ W_gate).to(torch.float32)
router_scores = torch.sigmoid(router_logits)  # or softmax in FP32
```

2. **Fix 2 — Router z-loss** (Zoph et al., 2022): Prevent router logits from drifting to large magnitudes in the first place by adding a small quadratic penalty on the log-sum-exp of the logits across a batch of B tokens:

$$L_z(x) = \frac{1}{B}\sum_{i=1}^{B}\left(\log\sum_{j=1}^{N} e^{x_j^{(i)}}\right)^2$$

Break down this formula from the inside out: for token i, the inner sum $\sum_{j=1}^{N} e^{x_j^{(i)}}$ followed by $\log(\cdot)$ computes $\mathrm{LogSumExp}(x^{(i)})$, which smoothly approximates $\max_j x_j^{(i)}$. Squaring it penalizes logits whose scale drifts far from zero, keeping every exponential safely inside the high-resolution region of floating-point arithmetic. OLMoE ablations comparing a z-loss weight of `0.001` against `0` confirm that removing z-loss triggers repeated loss spikes and drops on HellaSwag and MMLU.

DEFINITION

Router z-loss adds the penalty $L_z(x) = \frac{1}{B}\sum_{i=1}^{B}\left(\log\sum_{j=1}^{N} e^{x_j^{(i)}}\right)^2$ to keep router logit magnitudes near zero, preventing exponential overflow and precision loss.

We now have a mathematically balanced, numerically stable sparse MoE training recipe. Turning that recipe into fast wall-clock training and inference across many accelerators brings us to MoE systems engineering.

KEY IDEA

- Without balancing, MoEs suffer routing collapse onto ~2 experts; late-2026 models replace heavy auxiliary losses with aux-loss-free per-expert bias balancing (`noaux_tc`) plus a tiny sequence-wise balance term so load balancing does not fight language-modeling gradients.
- Upcasting router logits to FP32 and adding router z-loss prevents BF16 roundoff errors from being exponentially amplified into training-ending loss spikes.

✎ Check Your Understanding

1. **What would happen if** an expert i receives far more tokens than average during a training step under aux-loss-free balancing (`noaux_tc`)?
   Answer: Its online bias b_i is decreased before the next step, lowering $s_{i,t} + b_i$ so fewer tokens select expert i in Top-K without modifying the actual gating score $s_{i,t}$.
2. **Quick computation:** Why does the logit 128.5 lose accuracy in BF16, and what two complementary fixes prevent router instabilities?
   Answer: BF16 has only 7 mantissa bits, so at magnitude $128 = 2^7$ the spacing between adjacent representable numbers is 1.0, rounding 128.5 to 128 and causing a 36% softmax error; the two fixes are upcasting router computations to FP32 and applying router z-loss to keep logits small.

Section 9: MoE Systems Engineering — Expert Parallelism, Grouped GEMMs, LatentMoE, and Static-Shape Execution

When a model holds 552B or 2.8T parameters across 256 to 896 experts, the experts cannot fit on a single GPU. Hosting and running them efficiently requires a dedicated parallelism dimension and specialized execution kernels.

DEFINITION

Expert Parallelism (EP) partitions disjoint subsets of experts across distinct GPUs (for example, NVLink-connected NVIDIA Blackwell B200/B300 nodes) and composes orthogonally with Data Parallelism (DP) and Tensor/Model Parallelism (TP).

Trace the dataflow of Expert Parallelism across a cluster from Rank 1 to Rank E. On each rank, tokens first pass through **Attn (data shard r)** and the local **Router**, which decides which global experts each token needs. Because a token sitting on Rank 1 may select an expert hosted on Rank E, every MoE layer

executes **two collective communication barriers**:

1. **All-to-All Dispatch:** Each rank shuffles and sends its routed token activations across the interconnect to the ranks that own the selected destination experts.
2. **Local Experts ($1 \ldots k$ on Rank 1, $\ldots N$ on Rank E):** Each rank runs its locally stored expert FFNs on all incoming tokens dispatched to it from across the cluster.
3. **All-to-All Combine:** Each rank returns the computed expert outputs back to each token's original home rank, where they are weighted, summed, and passed to the **Next Layer (shard r)**.

High interconnect bandwidth is critical here—and recall from Section 7 why 1–2 always-on shared experts are so valuable: while routed activations travel across All-to-All Dispatch and Combine, each rank can overlap that network wait with its local shared-expert and attention compute.

Once tokens arrive at a rank's local experts, a second systems bottleneck strikes: **ragged token counts**. Even with good load balancing, different local experts receive slightly different numbers of tokens in any given microbatch—Expert 0 might receive 95 tokens, Expert 1 might receive 62, and Expert 2 might receive 48. Consider three ways to execute those expert matrix multiplications on Tensor Cores:

- **(A) Batched GEMM:** Requires every expert matrix multiplication X_0E_0, X_1E_1, X_2E_2 to have the exact same fixed batch dimension (fixed capacity). Experts with fewer tokens must pad with wasted zeros; experts with more tokens must drop them.
- **(B) Block-Diagonal:** Arranges equal-size expert blocks along the diagonal of a larger sparse matrix—still constrained to uniform capacity per expert.
- **(C) Block-Sparse / Grouped GEMM:** Packs variable-sized, ragged expert blocks (E_0 tall/busy, E_1 medium, E_2 compact) along a block-sparse layout and executes all local experts in a single fused kernel launch (pioneered in MegaBlocks and implemented via custom kernels in CUDA, CUTLASS, **CuTe DSL**, and Triton).

> **DEFINITION**
> **Block-sparse / Grouped GEMMs** execute a batch of matrix multiplications with variable, ragged token counts per expert inside a single fused kernel launch, enabling **dropless MoE** training and serving without padding waste or sacrificing Tensor Core occupancy.

While grouped GEMMs solve local compute efficiency, All-to-All communication volume remains a major overhead: every token routed to K experts must ship $K \times d_{\text{model}}$ elements across the network during Dispatch and another $K \times d_{\text{model}}$ elements back during Combine. **LatentMoE**—introduced in the early-2026 Nemotron 3 architecture for hardware-aware accuracy per byte and adopted at 2.8T scale as **Stable LatentMoE** in Kimi K3 (July 2026)—attacks communication volume directly by projecting activations into a smaller latent dimension $d_{\text{lat}} < d_{\text{model}}$ *before* dispatch.

DEFINITION

LatentMoE down-projects token activations from d_{model} to a smaller latent dimension $d_{\text{lat}} < d_{\text{model}}$ prior to All-to-All Dispatch, executes the routed experts in latent space, and up-projects back to d_{model} after All-to-All Combine, reducing interconnect traffic proportionally to $d_{\text{lat}}/d_{\text{model}}$.

Compare Standard MoE and LatentMoE vertically side by side. In Standard MoE (left), the hidden state of width d_{model} enters **All-to-All (d_{model})**, runs through Routed Experts, returns via **All-to-All (d_{model})**, and sums with the Shared Expert. In LatentMoE (right), the hidden state d_{model} splits: a dashed green bypass sends the full d_{model} vector directly through the local **Shared Expert**, while the routed path passes through a **Latent down-proj** before entering **Dispatch (d_{lat})**, runs **Latent Experts**, returns via **Combine (d_{lat})**, passes through **Latent up-proj**, and merges with the Shared Expert output.

WORKED EXAMPLE

Compute the All-to-All communication volume per token per MoE layer in BF16 (2 bytes per element) for a model with $K = 16$ active experts and $d_{\text{model}} = 8{,}192$, comparing Standard MoE against LatentMoE with $d_{\text{lat}} = 2{,}048$:

1. **Standard MoE:**
   - Dispatch volume: $16 \times 8{,}192 \times 2 \text{ bytes} = 262{,}144 \text{ bytes} = 256 \text{ KiB/token}$.
   - Combine volume: another 256 KiB/token.
   - Total All-to-All traffic: $512 \text{ KiB/token/layer}$.
2. **LatentMoE ($d_{\text{lat}} = 2{,}048$):**
   - Dispatch volume: $16 \times 2{,}048 \times 2 \text{ bytes} = 65{,}536 \text{ bytes} = 64 \text{ KiB/token}$.
   - Combine volume: another 64 KiB/token.
   - Total All-to-All traffic: $128 \text{ KiB/token/layer}$.

Compressing activations by $4\times$ into latent space slashes All-to-All network traffic by $4\times$ on every MoE layer, while the local shared expert bypass continues operating at full width d_{model}.

What happens if a system does *not* use dropless grouped GEMMs and instead enforces a fixed `capacity_factor` buffer per expert? Trace the four-stage execution diagram with a buffer capacity of 2 tokens per expert:

1. **(1) Routing:** Five tokens route as: `"the"` $\to E_1$, `"quick"` $\to E_0$, `"brown"` $\to E_2$, `"fox"` $\to E_0$, and `"over"` $\to E_0$ (plus `"jumped"` $\to E_1$ elsewhere in the batch).
2. **(2) Permute (cap = 2):** Each expert's fixed 2-slot buffer is filled: E_2 holds `["brown", pad]`, E_1 holds `["the", "jumped"]`, and E_0 fills with `["quick", "fox"]`. Because E_0's buffer is full, **`"over"` is DROPPED**.
3. **(3) Expert Compute:** Experts 2, 1, and 0 run on their fixed 2-slot buffers.
4. **(4) Un-Permute & Scale:** Processed tokens return scaled by their router probabilities ($E_1(\text{"the"}) \cdot p_1$, $E_0(\text{"quick"}) \cdot p_2$, etc.), while **`"over"` receives 0** from the MoE sublayer and passes forward via

the residual connection alone.

COMMON MISCONCEPTION

You might think setting sampling temperature to 0 (greedy decoding) makes an MoE inference server completely deterministic for your prompt, but that breaks whenever fixed `capacity_factor` token dropping is enabled. Because expert buffers are shared across the entire serving batch, another user's prompt co-scheduled in the same batch can fill an expert's buffer first and cause *your* token to be dropped—producing cross-request nondeterminism even at temperature 0.

Even in dropless systems, extreme popularity skew creates a different problem: **stragglers**. If Expert 1 is "hot" and receives $2\times$ the token load of Experts 2 and 3, Rank 0 takes twice as long as Ranks 1 and 2 while everyone else sits idle waiting at the All-to-All Combine barrier. **Online expert replication** solves both skew and dynamic-shape overhead at once.

DEFINITION

Online expert replication dynamically copies popular ("hot") experts onto underloaded ranks and splits incoming tokens across replicas so every rank processes an identical token count—achieving completely static tensor shapes with zero token drops and zero padding.

Picture the before-and-after load chart across three ranks. **Before (Skewed Static Placement):** Rank 0 holds hot expert `E1` at a tall $2\times$ load bar (the straggler), while Rank 1 (`E2`) and Rank 2 (`E3`) have short bars. After triggering **Replicate hot expert**, **After (Online Replication — Static Shapes):** Rank 0 processes half of `E1`'s tokens, Rank 1 processes `E2`, and Rank 2 hosts an `E1 copy` alongside `E3`—bringing all three bars to the exact same horizontal height line. Production frameworks operationalize this directly: **MoonEP** (Moonshot) keeps every rank's received token count identical so tensor shapes stay completely static, and **DeepEP V2.5** adds redundant-expert weight prefetching and gradient reduction across expert replicas during training.

Having examined how MoEs execute across clusters, we turn to two lifecycle challenges—fine-tuning sparse models without overfitting and initializing MoEs from existing dense models—and trace how DeepSeek integrated every routing and systems technique across four generations.

KEY IDEA

- Expert Parallelism introduces two All-to-All collective barriers per layer (Dispatch and Combine); Grouped/Block-Sparse GEMMs eliminate fixed-capacity token dropping (and batch-level nondeterminism), while LatentMoE cuts All-to-All traffic by projecting tokens to $d_{\text{lat}} < d_{\text{model}}$ before dispatch.
- Online expert replication (MoonEP, DeepEP V2.5) eliminates hot-expert stragglers by copying popular experts across ranks so every rank receives an identical token count with static shapes, zero drops, and zero padding.

✎ Check Your Understanding

1. **Yes or No with justification:** With fixed-capacity token dropping enabled, does a dropped token get deleted from the sequence completely?
   Answer: No. A dropped token skips the expert FFN (its MoE contribution is set to 0) and passes forward to the next layer strictly through the residual connection.
2. **What would happen if** an MoE system replaced standard All-to-All communication with LatentMoE where $d_{\text{lat}} = d_{\text{model}}/2$?
   Answer: Per-token All-to-All Dispatch and Combine communication volume would be cut in half (from $2Kd_{\text{model}}$ elements to $2Kd_{\text{lat}}$ elements per token per layer), while the shared expert bypass would continue operating locally at full d_{model}.

Section 10: MoE Fine-Tuning, Sparse Upcycling, and the DeepSeek Architectural Evolution

Sparse MoE models pack enormous total parameter counts into low active-FLOP footprints, which creates a sharp asymmetry during downstream fine-tuning. When supervised fine-tuning (SFT) datasets are small, a sparse model's massive parameter pool memorizes the training examples and **overfits** far more readily than a dense model of equal active FLOPs—an effect clearly documented by Zoph et al. (2022) on the SuperGLUE CB benchmark. Two practical fixes address this depending on data availability:

- **Fix A (small data):** Freeze the MoE expert weights and fine-tune only the non-MoE weights (attention, norms, embeddings, and output projections).
- **Fix B (scale data):** Scale up SFT data volume massively so all experts see diverse supervision—for example, DeepSeek's **1.4M** curated SFT examples.

A second practical question arises before pretraining even begins: if you already have a strong pretrained dense checkpoint, must you train an MoE from scratch? **Sparse upcycling** avoids throwing away that dense pretraining investment.

DEFINITION

Sparse upcycling initializes an MoE model from a pretrained dense Transformer checkpoint by copying the attention and normalization layers verbatim, replicating each dense MLP weight matrix E times to form E initial experts, and inserting a freshly initialized router.

Trace the upcycling diagram from top to bottom. At the top sits a **Dense Block** containing `Attn + Norm` feeding into `Dense MLP`. A dashed green `copy` arrow transfers `Attn + Norm` directly into the **Upcycled MoE Block** below. Meanwhile, a `Replicate E× copies` transformation duplicates the `Dense MLP` into

`MLP copy 1`, `MLP copy 2`, …, `MLP copy E`, and a freshly initialized **New Router** sits between `Attn + Norm` and the E MLP copies. Continued pretraining rapidly breaks symmetry and specializes the experts, yielding strong efficiency gains:

- **MiniCPM-MoE** (13.6B total / ~4B active, top-2 of 8 experts, continued for ~520B tokens): beats the dense MiniCPM-2.4B model.
- **Qwen1.5-MoE-A2.7B** (14.3B total / 2.7B active, upcycled from Qwen 1.8B into 60 routed + 4 shared experts, top-4): matches 7B dense models at a fraction of the active compute.

> **COMMON MISCONCEPTION**
> You might think replicating a dense MLP E times during sparse upcycling leaves all E experts permanently identical because they start from the exact same weights, but that breaks because the newly inserted router is randomly initialized. On the very first training batch, the random router routes different subsets of tokens with different gating weights to each copy, producing distinct gradients across copies 1 through E that immediately diverge and drive expert specialization.

To see how every concept from Parts 1 and 2 converges in real frontier engineering, trace the four-generation **DeepSeek MoE Evolution (V1 → V4.1-Flash)**:

Generation	Params (Total / Active)	Experts (Routed / K / Shared)	Key Architectural Advances
V1 (2024)	16B / 2.8B	64 (1/4) / 6 / 2	Fine-grained + shared experts; softmax top-K; expert + device aux loss
V2 (2024)	236B / 21B	160 (1/10) / 6 / 2	**Top-M device-limited routing**; comm balance loss; **MLA**
V3 (Dec 2024)	**671B / 37B**	**256** (1/14) / **8** / **1**	**Sigmoid + renorm top-K**; **aux-loss-free bias** + seq-wise aux; MLA + 1-step **MTP**
V4 / V4.1-Flash (2026)	**552B backbone / 16B decode** (8B prefill)	**384 / 6 / 1**	V4: **CSA + HCA**, **mHC** residuals; V4.1-Flash: **CSA2** (FP4 KV, 890 B/tok), **3 MTP layers**, Engram

Notice the relentless two-fold trend across generations: every generation increased expert granularity (from 64 up to 384 routed experts) while simultaneously tightening both cross-node communication and KV-cache efficiency.

Zoom in on the systems-aware routing innovation introduced in DeepSeek V2 and refined in DeepSeek-V3 (Dec 2024): **Device-Limited Top-M Routing**. Under unrestricted Top-K routing across a multi-node cluster, a single token choosing $K = 8$ experts can easily land on 8 completely different nodes,

multiplying inter-node All-to-All traffic by K.

DEFINITION

Device-limited Top-M routing restricts each token to at most M destination devices by first selecting the M devices containing the highest expert affinity scores and then choosing Top-K experts solely among the experts hosted on those M devices, strictly bounding communication fan-out.

Empirically, setting $M \geq 3$ matches the quality of unrestricted Top-K routing while guaranteeing that no token ever fans out to more than M nodes. To ensure devices also send and receive balanced traffic, DeepSeek V2 added a **communication balance loss** ($\mathcal{L}_{\text{CommBal}}$) encouraging every device to both send and receive roughly MT hidden states. DeepSeek-V3 then combined device-limited Top-M routing with the renormalized sigmoid gate and bias-shifted selection:

$$s_{i,t} = \text{Sigmoid}(\mathbf{u}_t^\top \mathbf{e}_i), \qquad g_{i,t} = \frac{g'_{i,t}}{\sum_{j=1}^{N_r} g'_{j,t}}$$

where $g'_{i,t} = s_{i,t}$ only if expert i lies inside the Top-K of $\{s_{j,t} + b_j\}$ restricted to the Top-M devices (and 0 otherwise).

WORKED EXAMPLE

Trace Device-Limited Top-M Routing with 4 devices (D_1, D_2, D_3, D_4) hosting 2 experts each ($N_r = 8$), setting $K = 3$ active experts and $M = 2$ max devices. Suppose a token's sigmoid affinities $s_{i,t}$ (with $b_i = 0$ for simplicity) are:

- D_1: $E_1 = 0.80,\ E_2 = 0.72$ (device max $= 0.80$)
- D_2: $E_3 = 0.78,\ E_4 = 0.30$ (device max $= 0.78$)
- D_3: $E_5 = 0.75,\ E_6 = 0.25$ (device max $= 0.75$)
- D_4: $E_7 = 0.40,\ E_8 = 0.20$ (device max $= 0.40$)

1. **Unrestricted Top-3:** Selects $\{E_1(0.80), E_3(0.78), E_5(0.75)\}$, which forces All-to-All Dispatch to ship the token to **3 devices** (D_1, D_2, D_3).
2. **Device-Limited Top-M ($M = 2$):**
   - **Stage 1 (Top-2 devices):** Comparing device maxima $(0.80, 0.78, 0.75, 0.40)$ selects $\{D_1, D_2\}$.
   - **Stage 2 (Top-3 experts within $\{D_1, D_2\}$):** Among $\{E_1, E_2, E_3, E_4\}$, the Top-3 winners are $\{E_1(0.80), E_3(0.78), E_2(0.72)\}$.
   - **Renormalization:** Sum $= 0.80 + 0.78 + 0.72 = 2.30$, yielding final gating weights:

$$g_1 = \frac{0.80}{2.30} \approx 0.348, \qquad g_3 = \frac{0.78}{2.30} \approx 0.339, \qquad g_2 = \frac{0.72}{2.30} \approx 0.313$$

> Fan-out drops strictly from 3 devices to 2 devices while substituting $E_2(0.72)$ for $E_5(0.75)$—a tiny affinity trade-off for a strict network guarantee.

Look back at the DeepSeek V4 / V4.1-Flash row in the evolution table: alongside attention alternatives and MoE, two non-MoE architectural companions appear across late-2026 frontier models—**Manifold-Constrained Hyper-Connections (mHC)** and **Multi-Token Prediction (MTP)**—plus conditional memory tables. We examine them next to complete the full late-2026 architecture picture.

KEY IDEA

- Sparse MoEs overfit small SFT datasets (mitigated by freezing MoE weights or scaling SFT data, such as DeepSeek's 1.4M examples) and can be initialized efficiently from dense checkpoints via sparse upcycling.
- DeepSeek's evolution (V1 → V4.1-Flash) illustrates the co-design of fine-grained experts with systems-aware routing—most notably Device-Limited Top-M routing, which caps inter-node fan-out at M devices while preserving unrestricted Top-K quality for $M \geq 3$.

✎ Check Your Understanding

1. **Fill-in-the-blank:** When fine-tuning a sparse MoE on a small SFT dataset where overfitting occurs, Fix A is to fine-tune only the _____ weights, while Fix B is to scale up the SFT dataset size.
   Answer: non-MoE (leaving the sparse expert weights frozen).
2. **What would happen if** you lowered M in Device-Limited Top-M routing from $M = 4$ to $M = 2$ on a model with $K = 8$ active experts?
   Answer: Each token's inter-node All-to-All fan-out would be strictly capped at 2 devices instead of 4 (cutting cross-node communication), though restricting M below 3 risks degrading quality compared to unrestricted Top-K.

Section 11: Beyond Attention and MoE — mHC, Multi-Token Prediction, and the Late-2026 Frontier Synthesis

Standard Transformers pass information from layer to layer through a single residual stream: $\mathbf{x}_{l+1} = \mathbf{x}_l + F_l(\mathbf{x}_l)$. **Hyper-connections** widen that residual highway into n parallel residual streams (typically $n = 4$) and learn small $n \times n$ mixing matrices H_{res} before and after each sublayer so streams can route and recombine information across depth.

Why didn't unconstrained hyper-connections immediately replace standard residuals at frontier scale? Because unconstrained mixing matrices break the identity path that keeps deep networks trainable. When you multiply unconstrained $n \times n$ matrices across 60 or 80 layers, eigenvalues larger than 1 cause activations and gradients to explode exponentially, while eigenvalues smaller than 1 cause signals to vanish.

DeepSeek-V4 (2026) resolves this stability crisis with **Manifold-Constrained Hyper-Connections (mHC)**.

DEFINITION

Manifold-Constrained Hyper-Connections (mHC) widen the Transformer residual path into n parallel streams (typically $n = 4$) and project each layer's learned residual mixing matrix H_{res} onto the manifold of **doubly stochastic matrices** via **Sinkhorn–Knopp iterations** (up to 20 iterations in DeepSeek-V4), preserving stable signal propagation across arbitrary depth.

Trace the mHC block diagram from bottom to top. Four parallel vertical residual streams ($n = 4$) enter from the bottom into a **Mix Matrix** H_{res} block labeled `Sinkhorn–Knopp (≤20 iters)`. Four mixed streams proceed straight up toward the **Update n Streams** block at the top; simultaneously, a side branch feeds from the mixing stage into the **Attn / MoE Sublayer** F on the right, whose output returns to merge inside **Update n Streams**, emitting the updated $n = 4$ parallel streams upward to the next layer.

Why does constraining H_{res} to be **doubly stochastic** restore deep stability? Recall from linear algebra that a non-negative square matrix is doubly stochastic if every row sums to 1 and every column sums to 1. First, every row of H_{res} forms a convex combination of the incoming streams, so no output stream can exceed the maximum of the input streams. Second, because every column also sums to 1, the average across streams is strictly conserved, and the largest singular value (spectral norm) of any doubly stochastic matrix is bounded by 1. Third, the set of doubly stochastic matrices is closed under multiplication: multiplying 80 doubly stochastic matrices across an 80-layer network still yields a doubly stochastic matrix, completely preventing exponential blow-up across depth. To project any positive learned matrix onto this manifold, **Sinkhorn–Knopp iteration** simply alternates dividing every row by its row sum and dividing every column by its column sum until convergence.

WORKED EXAMPLE

Start with an unconstrained positive 2×2 mixing matrix $A^{(0)} = \begin{pmatrix} 3 & 1 \\ 1 & 1 \end{pmatrix}$ (note that applying $A^{(0)}$ to a unit vector $(1, 1)^\top$ gives $(4, 2)^\top$, amplifying the average signal by $3\times$ in a single layer). Run 2 iterations of Sinkhorn–Knopp normalization:

1. **Iteration 1:**

- **Row normalize** (divide row 1 by $3+1=4$, row 2 by $1+1=2$):

$$R^{(1)} = \begin{pmatrix} 0.750 & 0.250 \\ 0.500 & 0.500 \end{pmatrix} \qquad (\text{column sums: } 1.250,\ 0.750)$$

- **Column normalize** (divide col 1 by 1.250, col 2 by 0.750):

$$A^{(1)} = \begin{pmatrix} 0.600 & 0.333 \\ 0.400 & 0.667 \end{pmatrix} \qquad (\text{row sums: } 0.933,\ 1.067)$$

2. **Iteration 2:**

- **Row normalize** (divide row 1 by 0.933, row 2 by 1.067):

$$R^{(2)} = \begin{pmatrix} 0.643 & 0.357 \\ 0.375 & 0.625 \end{pmatrix} \qquad (\text{column sums: } 1.018,\ 0.982)$$

- **Column normalize** (divide col 1 by 1.018, col 2 by 0.982):

$$A^{(2)} \approx \begin{pmatrix} 0.632 & 0.364 \\ 0.368 & 0.636 \end{pmatrix} \qquad (\text{row sums: } 0.996,\ 1.004;\ \text{col sums: } 1.000,\ 1.000)$$

After just 2 iterations, $A^{(2)}$ is within 0.4% of the exact doubly stochastic limit $\begin{pmatrix} 0.634 & 0.366 \\ 0.366 & 0.634 \end{pmatrix}$. Applying $A^{(2)}$ to $(1,1)^\top$ yields $(0.996, 1.004)^\top \approx (1,1)^\top$: the streams mix cleanly while preserving signal scale across depth.

This expressiveness comes with a systems cost: widening the residual stream by $n\times$ multiplies residual memory traffic by $n\times$ and introduces serial row/column reductions per layer, which requires **fused kernels** (DeepSeek self-reports a **6.7%** overhead of an overlapped pipeline stage).

The second non-MoE architectural companion across late-2026 frontier models is **Multi-Token Prediction (MTP)**. Standard autoregressive training predicts only the single next token t_{i+1} from position i, which provides a sparse training signal per step and generates only one token per forward pass at inference. MTP appends D lightweight sequential prediction modules that predict $t_{i+2}, t_{i+3}, \ldots, t_{i+D+1}$ in a causal chain.

DEFINITION

Multi-Token Prediction (MTP) chains lightweight sequential Transformer modules off the main model's final hidden state to predict multiple future tokens ($t_{i+2}, t_{i+3}, \ldots$) during training for denser planning supervision, while doubling as built-in draft heads for speculative decoding at inference.

Trace the MTP block diagram horizontally from left to right:

- **Main Model (Next-1):** Shared Embedding (t_i) passes up through `Transformer × L` to produce final hidden state $\mathbf{h}_i^0$, which enters the `Shared Head` to predict P_{i+1}.
- **MTP Module 1 (Next-2):** Receives $\mathbf{h}_i^0$ horizontally from the Main Model and `Shared Embedding` (t_{i+1}) from below, combines them in `1 Block + Proj` to produce $\mathbf{h}_i^1$, and feeds the `Shared Head` to predict P_{i+2}^1.
- **MTP Module 2 (Next-3):** Receives $\mathbf{h}_i^1$ horizontally from MTP Module 1 and `Shared Embedding` (t_{i+2}) from below, runs `1 Block + Proj` to produce $\mathbf{h}_i^2$, and feeds the `Shared Head` to predict P_{i+3}^2.

Formally, for MTP depth $k \geq 1$:

$$\mathbf{h}_i'^k = M_k\big[\mathrm{RMSNorm}(\mathbf{h}_i^{k-1});\ \mathrm{RMSNorm}(\mathrm{Emb}(t_{i+k}))\big], \qquad \mathbf{h}_{1:T-k}^k = \mathrm{TRM}_k(\mathbf{h}_{1:T-k}'^k),$$

Decompose these three equations hierarchically:

1. **Inner normalization and concatenation:** $\mathrm{RMSNorm}(\mathbf{h}_i^{k-1}) \in \mathbb{R}^d$ (the representation from the preceding depth $k-1$) and $\mathrm{RMSNorm}(\mathrm{Emb}(t_{i+k})) \in \mathbb{R}^d$ (the shared embedding of the next ground-truth token t_{i+k}) are normalized to matched scales and concatenated into a $2d$-dimensional vector $[\cdot;\cdot]$.
2. **Projection (M_k):** The linear projection matrix $M_k \in \mathbb{R}^{d\times 2d}$ maps the $2d$ concatenation back to model dimension d, forming $\mathbf{h}_i'^k$.
3. **Single Transformer block (TRM_k):** Processes the sequence $\mathbf{h}_{1:T-k}'^k$ (truncated to length $T-k$ because position i looks k tokens ahead at t_{i+k}) through one causal Transformer block to produce $\mathbf{h}_{1:T-k}^k$.
4. **Shared output head ($\mathrm{OutHead}$):** Projects $\mathbf{h}_i^k$ using the exact same unembedding head shared with the main model to produce the next-token probability distribution P_{i+k+1}^k over target t_{i+k+1}.

MTP delivers dual payoffs: during training, denser supervision forces the backbone hidden states $\mathbf{h}_i^0$ to plan multiple tokens ahead; during inference, the lightweight MTP modules serve as built-in draft heads for **speculative decoding** (similar to EAGLE), drafting multiple candidate tokens that the main model verifies in parallel. Adoption expanded rapidly from 1 MTP step in DeepSeek-V3 (Dec 2024) and GLM-5.3 (2026) to **3 MTP layers** in DeepSeek-V4.1-Flash (Sept 2026).

> **WORKED EXAMPLE**
> Trace MTP indexing on a 5-token sequence $(t_1, t_2, t_3, t_4, t_5)$ ($T=5$) at position $i=2$ with two MTP modules ($k=1,2$):
>
> 1. **Main Model ($k=0$):** Reads prefix (t_1, t_2), computes $\mathbf{h}_2^0$, and predicts P_3^0 against ground-truth target t_3.
> 2. **MTP Module 1 ($k=1$):** Normalizes and concatenates $\mathbf{h}_2^0$ with $\mathrm{Emb}(t_{2+1}) = \mathrm{Emb}(t_3)$, projects via M_1 to $\mathbf{h}_2'^1$, runs TRM_1 across valid sequence indices $1 \ldots T-1 = 1 \ldots 4$ to obtain $\mathbf{h}_2^1$, and predicts P_4^1 against target t_4.
> 3. **MTP Module 2 ($k=2$):** Normalizes and concatenates $\mathbf{h}_2^1$ with $\mathrm{Emb}(t_{2+2}) = \mathrm{Emb}(t_4)$, projects via M_2 to $\mathbf{h}_2'^2$, runs TRM_2 across valid indices $1 \ldots T-2 = 1 \ldots 3$ to obtain $\mathbf{h}_2^2$,

and predicts P_5^2 against target t_5.

At every depth k, the module receives the exact preceding token embedding t_{i+k} before predicting t_{i+k+1}, preserving the full causal chain.

COMMON MISCONCEPTION

You might think MTP modules predict t_{i+2} and t_{i+3} independently in parallel straight from the main model's hidden state $\mathbf{h}_i^0$, but that breaks because independent parallel heads cannot condition t_{i+3} on the choice of t_{i+2}. By feeding $\mathrm{Emb}(t_{i+k})$ and $\mathbf{h}_i^{k-1}$ sequentially into module k, MTP preserves full causal conditioning across multi-token predictions.

Putting Parts 1, 2, and 3 together reveals the complete architecture of late-2026 frontier language models:

Model (2026)	Attention	MoE (Routed / Active / Shared)	Notable Extras
DeepSeek-V4 / V4.1-Flash (Sept)	Window 128 + CSA/HCA / CSA2 (FP4 KV)	384 / 6 / 1 (552B / 16B decode)	V4: mHC; V4.1: 3 MTP, Engram
Kimi K3 / K3.1 (July)	3 KDA : 1 Gated MLA	896 / 16 / 2 (2.8T / 104B)	Stable LatentMoE, MXFP4 QAT
Qwen3.8-Flash-Next (Aug)	3 GDN : 1 QSA (budget 2048)	512 / 10 / 1 (125B / 6B)	51B host-memory n-gram table
GLM-5.3 (2026)	MLA + DSA (top-k 2048)	256 / 8 / 1 (78L, $d = 6144$)	1 MTP layer
MiniMax-M3 (June)	MSA blockwise sparse GQA	109B MSA evaluation model	28.4× less attn FLOPs at 1M

Two further frontier trends sit right at the edge of this table:

- **Low-precision frontier:** Quantization now reaches both the MoE experts and the global KV cache—Kimi K3 trains its MoE experts with **MXFP4 quantization-aware training (QAT)**, while DeepSeek-V4.1-Flash stores its global KV cache in **FP4**.
- **Conditional memory (the next step past MoE):** Just as MoE decoupled parameter capacity from FFN FLOPs, **conditional memory** decouples static lookup capacity from FFN compute altogether using external memory tables—exemplified by DeepSeek-V4.1's **Engram** and Qwen3.8-Flash-Next's **51B-parameter host-memory n-gram embedding table** (prefetched asynchronously from host memory).

DEFINITION

Conditional memory extends sparsity beyond MoE by offloading factual and lexical lookup capacity into large external memory tables (such as DeepSeek-V4.1's Engram or Qwen3.8-Flash-Next's 51B host-memory n-gram table) decoupled from FFN compute.

Across all three parts of this lecture, the same engineering principle recurs: dense Transformers force every token to pay full compute and memory for every past position and every stored parameter. Late-2026 frontier models break both bottlenecks simultaneously—combining cheap local/recurrent mixers, sparse compressed global attention, fine-grained aux-loss-free MoEs with systems-aware execution, doubly stochastic mHC residuals, multi-token prediction heads, low-precision formats, and conditional memory.

KEY IDEA

- Manifold-Constrained Hyper-Connections (mHC) widen the residual stream to n parallel paths while using Sinkhorn–Knopp projection onto doubly stochastic matrices to preserve deep training stability; Multi-Token Prediction (MTP) chains lightweight sequential prediction blocks for denser training supervision and built-in speculative decoding.
- Late-2026 frontier models synthesize three-axis attention, fine-grained `noaux_tc` MoEs, mHC, MTP, 4-bit precision (MXFP4 QAT and FP4 KV caches), and conditional memory tables to scale context to 1M tokens and capacity into the trillions of parameters.

✎ Check Your Understanding

1. **Yes or No with justification:** If two consecutive layers in an mHC Transformer both use doubly stochastic residual mixing matrices $H_{\text{res}}^{(1)}$ and $H_{\text{res}}^{(2)}$, can their product $H_{\text{res}}^{(2)} H_{\text{res}}^{(1)}$ amplify the residual stream exponentially?
   Answer: No. The set of doubly stochastic matrices is closed under matrix multiplication, so the product remains doubly stochastic (every row and column still sums to 1, bounding its spectral norm by 1 and preventing exponential amplification).
2. **Fill-in-the-blank:** In Multi-Token Prediction (MTP), module k combines $\mathrm{RMSNorm}(\mathbf{h}_i^{k-1})$ with $\mathrm{RMSNorm}(\mathrm{Emb}(t_{i+k}))$ to predict distribution P_{i+k+1}^k, providing denser planning supervision during training and acting as built-in draft heads for _____ during inference.
   Answer: speculative decoding.