# Lecture 10: Inference

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

## Section 1: Why Inference Dominates Compute, Core Metrics, and Transformer Shapes

Training a large language model is an expensive one-time event, whereas inference runs continuously across every production user query, every evaluation benchmark, and every post-training reinforcement learning (RL) rollout. The scale imbalance between these two phases is stark. At production scale, OpenAI processes roughly $8.6$ trillion tokens per day, compared against $32$ trillion total pretraining tokens for DeepSeek-V4 and $45$ trillion pretraining tokens for DeepSeek-V4.1-Flash. A single large-scale deployment consumes an entire frontier pretraining budget's worth of tokens every few days.

The nature of inference workloads has also shifted. Early deployments served human-paced chat, where a user typed a short prompt and read a few hundred tokens of output. Modern workloads are increasingly dominated by unbounded **agentic traces**, in which a single user query triggers multi-step internal reasoning, repeated tool calls, and long intermediate context before producing a final output. Because every generated token consumes hardware resources, tokens generated directly equal compute spent. To serve these workloads efficiently, production systems rely on specialized inference engines: **vLLM** (known for PagedAttention), **SGLang** (known for RadixAttention and agentic workloads), **TensorRT-LLM** (NVIDIA's inference engine), and **llama.cpp** (for local CPU and edge execution). Throughout the serving-systems sections of this lecture, we will use SGLang as our running example.

To understand why inference requires a completely different engineering toolkit than training, compare how tokens flow through a Transformer during training versus during inference generation. Recall that during standard autoregressive training, the entire target sequence $t_1, t_2, t_3, t_4$ is already known in advance. Because causal attention masks prevent future tokens from influencing earlier ones, all four tokens are fed into the model simultaneously and processed together in a single large matrix multiplication. That large matrix multiplication keeps the processor's arithmetic units saturated, making training **compute-bound**.

Inference generation breaks this parallelism. During generation, token $t_2$ cannot even be constructed until the model has finished computing token $t_1$, token $t_3$ depends on $t_2$, and so on. Consider the flow of an inference request as a two-stage timeline: first, a blue block representing the user's `prompt` is processed in one pass to produce the very first output token; next, arrows connect strictly sequential yellow blocks labeled `step 1`, `step 2`, and `step 3`, each producing a single token before the next step can begin. Crucially, every single sequential decode step must reload the model's weights and the stored history (the KV cache) from memory just to generate one token. This autoregressive dependency prevents parallelizing across generated tokens and makes high GPU utilization hard to achieve.

Because inference splits into an initial prompt-processing stage and a sequential token-generation stage, we cannot evaluate an inference system with a single speed number. Instead, we measure three distinct metrics, two of which map directly onto the timeline above:

**DEFINITION**

**Time-to-First-Token (TTFT)** is the elapsed time from when a request arrives until the first generated response token appears, which corresponds to the duration of the initial prompt-processing (**prefill**) pass and governs interactive responsiveness.

**DEFINITION**

**Time-Per-Output-Token (TPOT)**, also called per-token **latency**, is the time required to generate each subsequent output token for a single request (measured in seconds per token), governing interactive reading speed and agent execution speed.

**DEFINITION**

**Throughput** is the aggregate number of tokens generated per second across all concurrent requests being served simultaneously (measured in tokens per second), governing total hardware cost efficiency.

**COMMON MISCONCEPTION**

You might think that minimizing TPOT and maximizing throughput are the same objective—that making the server "faster" automatically improves both. That breaks because serving many concurrent requests in a single batch increases total tokens produced per second (raising throughput) while forcing each individual request to wait longer for its turn through larger shared memory reads (worsening TPOT). We will derive this exact mathematical tradeoff in Section 3.

To analyze the exact compute and memory costs of inference, we first fix our Transformer architecture diagram and tensor dimension notation. A single Transformer layer transforms an input tensor $X$ of shape $[B, T, D]$ into an output tensor $Y$ of the same shape $[B, T, D]$ through two sequential sub-blocks:

1. **Attention (repeated across $L$ layers):** From the input $X$, the layer projects queries $Q$ of shape $[B, T, N, H]$ and retrieves the stored past keys and values from the **KV cache**, which holds tensors of shape $[B, S, K, H]$. The block computes attention scores $A = QK^\top$, applies $\mathrm{softmax}(A)V$, and projects the result back through the output weight matrix $W_O$.
2. **Gated MLP / MoE Experts:** The attention output enters a gated feed-forward block. It computes two projections into the intermediate dimension $F$: $U = XW_{\text{up}}$ and $G = XW_{\text{gate}}$, both of shape $[B, T, F]$. It applies a nonlinearity to the gate and multiplies elementwise with $U$, then projects back to the model dimension: $Y = (\mathrm{act}(G) \odot U)W_{\text{down}}$, producing $[B, T, D]$.

Throughout our calculations, we use an **einops-style contraction** notation to track matrix multiplications cleanly across multi-dimensional tensors. When writing a product such as $BT\underline{D} \times \underline{D}H \to BTH$, the shared dimension $D$ is the **contracting dimension** (summed over in dot products), while $B, T$, and $H$ survive into the output shape. When a shared dimension is instead a **batching dimension**—meaning the multiplication happens independently for each slice along that axis—we keep it in the output, as in $\mathbf{B}\underline{D} \times \mathbf{B}\underline{D} \to B$, where $B$ batches and $D$ contracts.

The core dimension symbols used throughout this lecture are:

- $B$: batch size (number of concurrent sequences processed together)
- $L$: number of Transformer layers
- $V$: vocabulary size
- $S$: cached context length (total sequence length stored in the KV cache)
- $T$: number of active tokens processed in the current forward pass
- $D$: model hidden dimension
- $F$: MLP intermediate dimension
- $N$: number of query attention heads
- $K$: number of key-value (KV) attention heads
- $H$: per-head dimension

Standard architectural conventions connect several of these dimensions: $D = NH$, the query heads divide evenly into groups sharing KV heads so $N = KG$ (where $G$ is the group size), and for dense models $F \approx 4D$. Most importantly for inference analysis, the active token count $T$ distinguishes the two phases of serving: during **prefill**, all prompt tokens run in parallel so $T = S$; during **decode**, each sequence generates one new token at a time so $T = 1$.

> **WORKED EXAMPLE**
>
> **Worked Example: Tracing Tensor Shapes in Prefill vs. Decode**
>
> Suppose we run a Transformer layer with $D = 4096$, $F = 16384$, $N = 32$ query heads, $K = 8$ KV heads, and head dimension $H = 128$ (notice $NH = 32 \times 128 = 4096 = D$, and group size $G = N/K = 4$). Consider a batch of $B = 16$ requests, each with a prompt of length $S = 1024$.
>
> 1. **Prefill pass ($T = S = 1024$):**
>    - Input tensor $X$ has shape $[B, T, D] = [16, 1024, 4096]$, containing $16 \times 1024 \times 4096 = 67{,}108{,}864$ elements.
>    - Query tensor $Q$ has shape $[B, T, N, H] = [16, 1024, 32, 128]$.
>    - Cached keys $K_{\text{cache}}$ and values $V_{\text{cache}}$ each have shape $[B, S, K, H] = [16, 1024, 8, 128]$.
>    - In the gated MLP, $U = XW_{\text{up}}$ performs the contraction $BTD \times DF \to BTF$, where $W_{\text{up}}$ has shape $[4096, 16384]$ and $U$ has shape $[16, 1024, 16384]$.
> 2. **First decode step ($T = 1$, context length now $S = 1025$):**

- Input tensor $X$ shrinks to $[16, 1, 4096]$, containing only $65{,}536$ elements—over $1000\times$ fewer active activations than prefill.
- Query tensor $Q$ has shape $[16, 1, 32, 128]$, yet it must attend against the full KV cache of shape $[16, 1025, 8, 128]$.
- Meanwhile, the MLP weight matrices $W_{\text{up}}, W_{\text{gate}}, W_{\text{down}}$ retain their full $[4096, 16384]$ and $[16384, 4096]$ shapes—the exact same number of parameters must be read to transform $16$ tokens as to transform $16{,}384$ tokens.

**Interpretation:** Moving from prefill ($T = S$) to decode ($T = 1$) collapses the number of active tokens processed per pass by a factor of $S$, while leaving both the model's weight matrices and the KV cache at full size.

**KEY IDEA**

Inference dominates lifetime LLM compute and splits into two structurally opposite phases: parallel **prefill** ($T = S$, governing TTFT) and strictly sequential **decode** ($T = 1$, governing TPOT and throughput), where every generated token must touch the full model weights and growing KV history.

Seeing that decode shrinks active tokens by a factor of $S$ while keeping weight and cache sizes unchanged immediately raises a quantitative question: how do we measure whether a given operation is bottlenecked by arithmetic speed or by memory transfer speed, and where do MLP layers and attention layers land on that spectrum? That is the subject of Section 2.

### ✎ Check Your Understanding

1. Fill in the blank: During prefill, the active token dimension satisfies $T =$ _____, and the latency of this stage determines the metric known as _____.

   **Answer:** $T = S$ (the prompt length), and its duration determines Time-to-First-Token (TTFT).

2. Does increasing the batch size $B$ generally improve per-request TPOT, or does it primarily improve aggregate throughput? Justify your answer in one to two sentences.

   **Answer:** Increasing $B$ primarily improves aggregate throughput (tokens/s across all requests) while worsening per-request TPOT, because batching amortizes fixed weight reads across more tokens at the cost of reading a larger combined KV cache on every step.

## Section 2: Arithmetic Intensity, the Roofline Model, and the KV Cache

To determine whether an inference step is limited by the speed of mathematical computation or by the speed at which data moves from high-bandwidth memory (HBM) to the compute units, we need a quantitative ratio comparing work done to bytes moved.

> **DEFINITION**
> **Arithmetic intensity** ($I$) is the ratio of floating-point operations performed to bytes transferred to and from main memory (HBM) for a given kernel or operation, measured in FLOPs per byte ($\text{FLOP}/\text{byte}$).

> **DEFINITION**
> The **hardware roofline** is a log-log performance bound plotting achieved throughput ($\text{FLOP}/\text{s}$) on the vertical axis against arithmetic intensity ($\text{FLOP}/\text{byte}$) on the horizontal axis, consisting of a sloped **memory-bound** ramp where performance equals $\text{BW} \times I$ and a flat **compute-bound** ceiling at the hardware's peak $\text{FLOP}/\text{s}$.

Visualize the roofline diagram carefully. The horizontal axis is arithmetic intensity in FLOP/byte on a logarithmic scale; the vertical axis is achieved FLOP/s on a logarithmic scale. Starting from the lower left, a thick blue line rises diagonally with slope 1 (in log-log coordinates, $\log(\text{BW} \cdot I) = \log(\text{BW}) + \log(I)$). In this sloped region, the processor is starved for data: its compute units sit partly idle waiting for bytes to arrive from HBM, so performance increases linearly if you raise arithmetic intensity. At a specific threshold called the **ridge point**, the diagonal ramp meets a flat horizontal plateau at the processor's maximum compute rate. To the right of the ridge point, the operation is **compute-bound**: HBM supplies data fast enough to keep the arithmetic units busy, and further increases in arithmetic intensity cannot push performance above the flat ceiling.

The ridge point occurs exactly where memory-limited performance equals compute-limited performance:

$$I^* = \frac{\text{peak FLOP}/\text{s}}{\text{BW}}$$

Here, $\text{peak FLOP}/\text{s}$ is the processor's maximum arithmetic throughput for the chosen numeric precision, and $\text{BW}$ is its HBM memory bandwidth in bytes per second. Any operation with $I < I^*$ lies to the left of the ridge point in the red memory-bound zone (typical of decode with small batch size $B$); any operation with $I > I^*$ lies to the right in the green compute-bound zone (typical of prefill or decode with very large $B$).

Let us compute the arithmetic intensity of the fundamental building block of a Transformer: multiplying an activation matrix $X$ of shape $[B \times D]$ by a weight matrix $W$ of shape $[D \times F]$ in BF16 precision ($2$ bytes per scalar).

- **Floating-point operations:** Each of the $B \cdot F$ output elements is an inner product of length $D$, requiring $D$ multiplications and $D$ additions, or $2D$ FLOPs. Summing across all outputs gives:

$$\text{FLOPs} = 2BDF$$

- **Memory traffic:** Reading $X$ moves $2BD$ bytes, reading $W$ moves $2DF$ bytes, and writing the output matrix $Y$ of shape $[B \times F]$ moves $2BF$ bytes:

$$\text{Bytes} = 2BD + 2DF + 2BF$$

- **Arithmetic intensity:** Dividing FLOPs by bytes yields:

$$I = \frac{2BDF}{2BD + 2DF + 2BF}$$

Now examine what happens when the batch size $B$ is much smaller than the matrix dimensions $D$ and $F$ ($B \ll D, F$), which is typical during decode. In the denominator, $2BD$ and $2BF$ are tiny compared to the weight matrix term $2DF$, so the denominator is dominated by $2DF$. The ratio simplifies dramatically:

$$I \approx \frac{2BDF}{2DF} = B \text{ FLOP/byte}$$

At $B = 1$, this matrix multiplication degenerates into a matrix-vector product with $I \approx 1 \text{ FLOP/byte}$ : we must stream the entire $D \times F$ weight matrix from HBM just to perform $2DF$ FLOPs.

Compare that against real hardware ridge points. On the historical NVIDIA H100 baseline in BF16, peak compute is $989 \text{ TFLOP/s}$ and HBM bandwidth is $3.35 \text{ TB/s}$, placing the ridge point at:

$$I^*_{\text{H100, BF16}} = \frac{989 \times 10^{12} \text{ FLOP/s}}{3.35 \times 10^{12} \text{ B/s}} \approx 295 \text{ FLOP/byte}$$

Therefore, a single BF16 matrix multiplication on an H100 is compute-bound if and only if $B > 295$. On our late-2026 reference GPU, the NVIDIA Blackwell B300—equipped with $288 \text{ GB}$ of HBM ($287 \text{ GB}$ usable) and native MXFP8 and NVFP4 tensor cores—decode likewise stays firmly memory-bound unless the batch size $B$ is large.

> **WORKED EXAMPLE**
> **Worked Example: Single Matmul Arithmetic Intensity at $B = 1$ vs. $B = 512$**
>
> Consider multiplying $X[B \times D]$ by $W[D \times F]$ in BF16 with $D = 5120$ and $F = 13824$, evaluated against the H100 BF16 ridge point $I^* \approx 295 \text{ FLOP/byte}$.
>
> 1. **Case 1 ($B = 1$):**
>    - $\text{FLOPs} = 2(1)(5120)(13824) = 1.416 \times 10^8 \text{ FLOPs}$.
>    - $\text{Bytes} = 2(1)(5120) + 2(5120)(13824) + 2(1)(13824) = 10{,}240 + 141{,}557{,}760 + 27{,}648 = 141{,}595{,}648 \text{ bytes}$.

- $I = \frac{1.4156 \times 10^8}{1.4160 \times 10^8} \approx 0.9997 \text{ FLOP/byte} \approx 1$.
- Because $I \approx 1 \ll 295$, the operation uses roughly $1/295 \approx 0.34\%$ of the GPU's peak BF16 compute!

2. **Case 2 ($B = 512$):**
   - $\text{FLOPs} = 2(512)(5120)(13824) = 7.248 \times 10^{10}$ FLOPs.
   - $\text{Bytes} = 2(512)(5120) + 2(5120)(13824) + 2(512)(13824) = 5.243 \times 10^6 + 1.4156 \times 10^8 + 1.4156 \times 10^7 = 1.6096 \times 10^8$ bytes.
   - $I = \frac{7.2478 \times 10^{10}}{1.6096 \times 10^8} \approx 450.3 \text{ FLOP/byte}$.
   - Notice why $I$ is $450.3$ rather than $512$: at $B = 512$, the activation terms $2BD$ and $2BF$ are no longer negligible (they add about $13.7\%$ to the total bytes). Still, because $450.3 > 295$, the operation has crossed the ridge point and is compute-bound.

**Interpretation:** Batching reuses the same weight matrix $W$ across $B$ inputs without re-reading $W$ from HBM, lifting the operation up the memory-bound slope until it reaches the compute ceiling.

Before we apply arithmetic intensity to full Transformer layers, we must address why inference stores a KV cache in HBM in the first place. Suppose you generate tokens **naively** without caching anything across decode steps. Start with prompt tokens $[p_1, p_2]$ and run a full forward pass to produce $t_1$. To produce $t_2$, you feed the entire history $[p_1, p_2, t_1]$ back through the model from scratch. To produce $t_3$, you feed $[p_1, p_2, t_1, t_2]$ back through from scratch. Because self-attention over a sequence of length $T$ requires $O(T^2)$ FLOPs, running a full pass at every single step from length $1$ up to $T$ sums to $\sum_{t=1}^{T} O(t^2) = O(T^3)$ total FLOPs!

The **KV cache** eliminates this redundant recomputation by observing that causal attention never modifies the key and value vectors of past tokens once they have been computed. Instead of re-encoding past tokens on every step, we split inference into two passes:

**DEFINITION**

The **KV cache** is a persistent HBM buffer storing the previously computed key and value projection vectors across all layers for every active sequence, allowing each decode step to reuse past keys and values rather than recomputing them from scratch.

Picture the two-stage cached workflow side by side against the naive approach:

1. **Prefill ($T = S$):** Encode all $S$ prompt tokens in parallel, writing their key and value vectors into the HBM KV cache of shape $[B, S, L, K, H] \times 2$ (the factor of $2$ accounts for both keys and values).
2. **Decode ($T = 1$):** Feed only the single newly generated token, compute its query, key, and value vectors, read all past $K$ and $V$ vectors from the HBM cache to compute attention, and append the new token's $K$ and $V$ vectors to the cache.

Every sequence therefore stores:

$$\text{KV scalars per sequence} = 2 \cdot S \cdot L \cdot K \cdot H$$

Let us inspect each factor: $2$ for keys plus values, $S$ for the number of tokens stored in the sequence, $L$ for the number of layers, $K$ for the number of KV heads per layer, and $H$ for the dimension of each head. By introducing the KV cache, we reduce total generation compute from $O(T^3)$ down to $O(T^2)$—a massive computational savings—in exchange for paying HBM storage capacity and HBM read bandwidth on every decode step.

Now we can derive the exact arithmetic intensity of the two main building blocks of a Transformer layer: the **gated MLP** and **attention**.

### Arithmetic Intensity of Gated MLP Layers

Recall from Section 1 that a gated MLP computes $U = XW_{\text{up}}$, $G = XW_{\text{gate}}$, and $Y = (\text{act}(G) \odot U)W_{\text{down}}$ in BF16 ($2\ \text{bytes/param}$), where $X$ and $Y$ have shape $[B, T, D]$, $U$ and $G$ have shape $[B, T, F]$, and each of the three weight matrices has $D \cdot F$ parameters. Accounting step by step:

| Operation | FLOPs | HBM Bytes (BF16) |
|---|---|---|
| Read $X$ $[B, T, D]$ and weights $W_{\text{up}}, W_{\text{gate}}, W_{\text{down}}$ | $0$ | $2BTD + 6DF$ |
| Compute $U = XW_{\text{up}}, G = XW_{\text{gate}}$; write $U, G$ | $4BTDF$ | $4BTF$ |
| Compute $Y = (\text{act}(G) \odot U)W_{\text{down}}$; write $Y$ | $2BTDF$ | $2BTD$ |
| **Total** | $6BTDF$ | $4BTD + 4BTF + 6DF$ |

Notice how each entry arises. Reading the three $[D, F]$ weight matrices in BF16 costs $3 \times 2DF = 6DF$ bytes. Computing both $U$ and $G$ requires two matrix multiplies of size $[BT, D] \times [D, F]$, contributing $2 \times 2BTDF = 4BTDF$ FLOPs and writing $2 \times 2BTF = 4BTF$ bytes (assuming the intermediate activations $U$ and $G$ are written to and read back—or counting the activation traffic as written here; note also the elementwise activation and multiply cost $O(BTF)$ FLOPs, which is negligible compared to $BTDF$). Computing the down-projection adds $2BTDF$ FLOPs and writes $2BTD$ bytes for $Y$.

Dividing total FLOPs by total HBM bytes yields the MLP arithmetic intensity:

$$I_{\text{MLP}} = \frac{6BTDF}{4BTD + 4BTF + 6DF}$$

When the total number of active tokens $B \cdot T$ is small compared to $D$ and $F$ ($BT \ll D, F$), the weight-reading term $6DF$ dominates the denominator, and the $6DF$ factors cancel cleanly:

$$I_{\text{MLP}} \xrightarrow{BT \ll D,F} B \cdot T$$

This single expression explains both inference regimes for MLP layers:

- **Prefill ($T = S$):** $I_{\text{MLP}} \approx B \cdot S$. Even at batch size $B = 1$, a prompt of $S = 512$ or $1024$ tokens easily pushes the MLP far past the hardware ridge point into the compute-bound regime.
- **Decode ($T = 1$):** $I_{\text{MLP}} \approx B$. Because every sequence in the batch multiplies against the *same* shared weights $W_{\text{up}}, W_{\text{gate}}, W_{\text{down}}$, increasing batch size $B$ directly raises MLP arithmetic intensity.

### Arithmetic Intensity of Attention (and a Classic Pitfall)

Now run the exact same analysis on attention using FlashAttention (recall that FlashAttention keeps the intermediate $S \times T$ attention score matrix on-chip so it is never written to or read from HBM). Consider $N$ query heads, $K$ KV heads (with group size $G = N/K$), cached context length $S$, active tokens $T$, head dimension $H$, and BF16 precision ($2\ \text{bytes/scalar}$).

- **FLOPs:** For each of the $B$ sequences and $N$ query heads, multiplying $Q[T, H]$ by $K^\top[H, S]$ costs $2STH$ FLOPs, and multiplying the attention weights $[T, S]$ by $V[S, H]$ costs another $2STH$ FLOPs. Summing over $B$ sequences and $N$ heads gives:

$$\text{FLOPs} = 2BSTNH + 2BSTNH = 4BSTNH$$

- **HBM Bytes:** Reading $Q[B, T, N, H]$ moves $2BTNH$ bytes; writing the output $Y[B, T, N, H]$ moves another $2BTNH$ bytes; reading both $K$ and $V$ caches, each of shape $[B, S, K, H]$, moves $2 \times (2BSKH) = 4BSKH$ bytes. Summing these gives:

$$\text{HBM Bytes} = 4BTNH + 4BSKH$$

Dividing FLOPs by HBM bytes, notice immediately that the common factors $4$, $B$, and $H$ cancel out completely from numerator and denominator! Dividing numerator and denominator by $K$ as well gives the exact arithmetic intensity of attention:

$$I_{\text{attn}} = \frac{4BSTNH}{4BTNH + 4BSKH} = \frac{S\,T\,(N/K)}{T\,(N/K) + S}$$

Look closely at every symbol in the final simplified fraction: $S$ is the cached context length, $T$ is the number of active tokens, and $N/K$ is the number of query heads sharing each KV head. Most strikingly, $B$ **has vanished completely**.

Why did $B$ cancel out of $I_{\text{attn}}$ when it survived in $I_{\text{MLP}}$? Because in an MLP, every sequence in the batch shares the exact same weight matrices, so doubling $B$ doubles FLOPs while barely changing bytes moved. In attention, every sequence in the batch has its own distinct conversation history and therefore its own separate KV cache. Doubling $B$ doubles both the attention FLOPs *and* the KV cache bytes read from HBM, leaving their ratio untouched.

Now evaluate $I_{\text{attn}}$ across prefill and decode:

- **Prefill ($T = S$, standard Multi-Head Attention with $K = N$):** Substituting $T = S$ and $N/K = 1$ gives:

$$I_{\text{attn}} = \frac{S \cdot S \cdot 1}{S \cdot 1 + S} = \frac{S^2}{2S} = \frac{S}{2}$$

For a long prompt (for example, $S = 2048$ gives $I_{\text{attn}} = 1024\ \text{FLOP/byte}$), prefill attention is cleanly compute-bound.

- **Decode ($T = 1$):** Substituting $T = 1$ gives:

$$I_{\text{attn}} = \frac{S\,(N/K)}{(N/K) + S}$$

**COMMON MISCONCEPTION**

You might think—based on a classic rule of thumb often repeated about Transformer inference—that decode attention arithmetic intensity is fundamentally stuck below $1\ \text{FLOP/byte}$ and can never be improved. That breaks because the bound $I_{\text{attn}} < 1$ holds *only* for Multi-Head Attention (MHA, where $K = N$). In MHA, $N/K = 1$, so $I_{\text{attn}} = \frac{S}{S+1} < 1$ regardless of batch size or context length. The moment we share each KV head across multiple query heads—as in Grouped-Query Attention (GQA, where $K < N$)—we get $I_{\text{attn}} = \frac{S(N/K)}{S+N/K} \approx N/K$ whenever $S \gg N/K$, because every loaded key/value byte now serves $N/K$ query heads. Taking this idea even further, in absorbed Multi-Head Latent Attention (MLA), every query head reads a single shared compressed latent vector (576 values in GLM-5.3, allowing all $64$ query heads to share each latent byte).

**WORKED EXAMPLE**

**Worked Example: Comparing Decode Attention Intensity in MHA vs. GQA**

Evaluate decode attention arithmetic intensity ($T = 1$) at context length $S = 1024$ for a model with $N = 40$ query heads, comparing MHA ($K = 40$) against GQA ($K = 8$, group size $N/K = 5$), at both $B = 1$ and $B = 256$.

1. **MHA ($K = 40$, $N/K = 1$):**

$$I_{\text{attn, MHA}} = \frac{1024 \times 1}{1 + 1024} = \frac{1024}{1025} \approx 0.999\ \text{FLOP/byte}$$

   This holds at $B = 1$ and remains $0.999\ \text{FLOP/byte}$ at $B = 256$.

2. **GQA ($K = 8$, $N/K = 5$):**

$$I_{\text{attn, GQA}} = \frac{1024 \times 5}{5 + 1024} = \frac{5120}{1029} \approx 4.976\ \text{FLOP/byte}$$

   Again, this is independent of $B$, yet it is nearly $5\times$ higher than MHA.

**Interpretation:** Batching cannot raise decode attention intensity because each request brings its own KV cache. Raising decode attention intensity—alongside shrinking total KV memory capacity—is the primary reason modern models adopt GQA and MLA, which we will examine in full architectural detail in Section 4.

**KEY IDEA**

MLP decode intensity scales linearly with batch size ($I_{\text{MLP}} \approx B$), whereas attention decode intensity is independent of $B$ and scales instead with KV head sharing ($I_{\text{attn}} \approx N/K$ for $S \gg N/K$).

We now know *why* decode is memory-bound for both MLPs (at small $B$) and attention. Next, we need a quantitative system model that translates HBM bytes and memory bandwidth directly into the metrics users and operators care about—latency (TPOT), throughput, and memory capacity—and reveals a surprising breakdown when we move from dense models to Mixture-of-Experts (MoE) architectures.

**✎ Check Your Understanding**

1. Compute the decode arithmetic intensity ($T = 1$) of a GQA layer with $N = 64$ query heads, $K = 8$ KV heads, and context length $S = 2048$.

   **Answer:** Here $N/K = 64/8 = 8$, so $I_{\text{attn}} = \frac{S(N/K)}{S+N/K} = \frac{2048 \times 8}{2048+8} = \frac{16384}{2056} \approx 7.97\ \text{FLOP/byte}$.

2. What would happen to $I_{\text{MLP}}$ and $I_{\text{attn}}$ during decode ($T = 1$) if you doubled the batch size from $B = 32$ to $B = 64$ (assuming $B \ll D, F$)?

   **Answer:** $I_{\text{MLP}}$ would roughly double (from $\approx 32$ to $\approx 64\ \text{FLOP/byte}$) because all requests share the same MLP weights, whereas $I_{\text{attn}}$ would stay completely unchanged because $B$ cancels out of the attention intensity formula when each sequence owns a separate KV cache.

## Section 3: Memory-Bound Performance Modeling, Hardware Case Studies, and MoE Batch Collapse

When a decode step operates to the left of the hardware ridge point (memory-bound), execution time is governed almost entirely by how many bytes must be streamed out of HBM divided by HBM bandwidth. Assuming perfect overlap between compute and memory I/O, each decode step streams all active model parameters plus the entire batch's KV cache from HBM. This yields a clean, three-equation performance model:

$$\text{Memory} = B \cdot \text{KV}_{\text{seq}} + \text{Params}_{\text{bytes}}, \qquad \text{Latency} = \frac{\text{Memory}}{\text{BW}}, \qquad \text{Throughput} = \frac{B}{\text{Laten}}$$

Let us unpack each equation term by term:

1. $\text{Memory} = B \cdot \text{KV}_{\text{seq}} + \text{Params}_{\text{bytes}}$**:** Total bytes read from HBM in one decode step (which also closely approximates the HBM capacity required to hold the model weights and active KV caches).

Here $\text{Params}_{\text{bytes}}$ is the fixed byte size of the model's active weights read per step, and $B \cdot \text{KV}_{\text{seq}}$ is the combined byte size of the KV caches across all $B$ concurrent sequences.

2. $\text{Latency} = \text{Memory}/\text{BW}$**:** Dividing total streamed bytes by the GPU's HBM bandwidth $\text{BW}$ gives the wall-clock duration of one decode step—which is directly TPOT in seconds per token.
3. $\text{Throughput} = B/\text{Latency}$**:** Because one decode step produces $1$ new token for each of the $B$ concurrent requests, it produces $B$ tokens in $\text{Latency}$ seconds. Substituting $\text{Latency}$ yields:

$$\text{Throughput} = \frac{B \cdot \text{BW}}{B \cdot \text{KV}_{\text{seq}} + \text{Params}_{\text{bytes}}} = \frac{\text{BW}}{\text{KV}_{\text{seq}} + \text{Params}_{\text{bytes}}/B}$$

Look at that rewritten throughput equation. It exposes the fundamental latency-throughput tradeoff of batch size $B$:

- **Increasing $B$ worsens latency:** In $\text{Latency} = (B \cdot \text{KV}_{\text{seq}} + \text{Params}_{\text{bytes}})/\text{BW}$, the KV cache term grows linearly as $O(B)$, so each user waits longer between consecutive tokens.
- **Increasing $B$ improves throughput—up to a ceiling:** Increasing $B$ shrinks the amortized weight-reading cost $\text{Params}_{\text{bytes}}/B$. When $B$ is small ($\text{Params}_{\text{bytes}} \gg B \cdot \text{KV}_{\text{seq}}$), doubling $B$ nearly doubles throughput. Once $B$ becomes large enough that $B \cdot \text{KV}_{\text{seq}}$ dominates $\text{Params}_{\text{bytes}}$, throughput hits diminishing returns and saturates at the asymptotic cap $\text{BW}/\text{KV}_{\text{seq}}$.

This model also explains why prefill and decode want opposite batching policies: during **prefill**, keeping $B$ small minimizes TTFT (a single prompt is already compute-bound); during **decode**, pushing $B$ large maximizes throughput by amortizing weight reads. Furthermore, it clarifies the two ways to scale across $M$ GPUs:

- **Data replication ($M$ independent copies of the model):** Each GPU runs its own independent batch, so per-request **latency is unchanged**, while aggregate **throughput multiplies by $M$**.
- **Sharding weights and KV across GPUs (model parallelism):** Required whenever a model's weights exceed a single GPU's HBM capacity, and also cuts single-request latency because multiple GPUs stream shards of the weights and KV cache in parallel using their combined HBM bandwidth.

To see these equations operate on concrete hardware, let us work through two complete case studies: first a historical 2023 dense model on an H100, then a 2026 frontier MoE model on a Blackwell B300.

> **WORKED EXAMPLE**
>
> **Worked Example: Historical Baseline — Llama 2 13B (2023) on H100**
>
> Consider the historical 2023 dense model Llama 2 13B in BF16 ($2\ \text{B}/\text{param}$) with $L = 40$ layers, $D = 5120$, $F = 13824$, $N = 40$ query heads, $H = 128$, and context length $S = 1024$, running on a single Hopper H100 ($80\ \text{GB}$ HBM, $\text{BW} = 3.35\ \text{TB/s} = 3350\ \text{GB/s}$, using decimal GB/MB where $1\ \text{GB} = 10^9\ \text{B}$).
>
> 1. **KV cache per sequence (MHA, $K = 40$):**
>
> $$\text{KV}_{\text{seq, MHA}} = 2 \cdot S \cdot L \cdot K \cdot H \cdot 2\ \text{B} = 2 \times 1024 \times 40 \times 40 \times 128 \times 2\ \text{B} = 838{,}860{,}8($$

With $\text{Params}_{\text{bytes}} \approx 26.06$ GB (for the 13B BF16 parameters):

- At $B = 1$: $\text{Memory} = 26.06 + 1(0.839) = 26.9$ GB.

$$\text{Latency} = \frac{26.9 \text{ GB}}{3350 \text{ GB/s}} = 8.0 \text{ ms/tok}, \qquad \text{Throughput} = \frac{1}{0.0080 \text{ s}} = 125 \text{ tok/s}$$

- At $B = 64$: $\text{Memory} = 26.06 + 64(0.839) = 79.7$ GB (barely fits in 80 GB).

$$\text{Latency} = \frac{79.7}{3350} = 23.8 \text{ ms/tok}, \qquad \text{Throughput} = \frac{64}{0.0238} = 2{,}689 \text{ tok/s}$$

- At $B = 256$: $\text{Memory} = 26.06 + 256(0.839) = 240.8$ GB (**OOM** on 80 GB!). Even ignoring capacity, $\text{Latency} = 240.8/3350 = 71.9$ ms/tok and $\text{Throughput} = 256/0.0719 = 3{,}562$ tok/s—quadrupling batch size from $64$ to $256$ increased throughput by only $1.32\times$ because KV reads ($214.7$ GB) completely drowned out weight reads ($26.1$ GB).

2. **Now replace MHA with GQA ($K = 8$, a $1:5$ ratio):**

- KV cache per sequence drops $5\times$: $\text{KV}_{\text{seq, GQA}} = 838.9 \text{ MB}/5 \approx 168$ MB.
- Shrinking $W_K$ and $W_V$ by $5\times$ across all 40 layers also trims $3.36$ GB from the BF16 weights, bringing $\text{Params}_{\text{bytes}}$ to $\approx 22.7$ GB.
- At $B = 64$: $\text{Memory} = 22.7 + 64(0.168) = 33.4$ GB (fits easily).

$$\text{Latency} = \frac{33.4}{3350} = 10.0 \text{ ms/tok}, \qquad \text{Throughput} = \frac{64}{0.0100} = 6{,}417 \text{ tok/s}$$

- At $B = 256$: $\text{Memory} = 22.7 + 256(0.168) = 65.6$ GB (still fits in 80 GB!).

$$\text{Latency} = \frac{65.6}{3350} = 19.6 \text{ ms/tok}, \qquad \text{Throughput} = \frac{256}{0.0196} = 13{,}068 \text{ tok/s}$$

| Config | KV / seq | Batch $B$ | Total Memory | Latency | Throughput | Fits 80 GB? |
|---|---|---|---|---|---|---|
| MHA ( $K = 40$ ) | 839 MB | 1 | 26.9 GB | 8.0 ms/tok | 125 tok/s | Yes |
| MHA ( $K = 40$ ) | 839 MB | 64 | 79.7 GB | 23.8 ms/tok | 2,689 tok/s | Barely |
| MHA ( $K = 40$ ) | 839 MB | 256 | 240.8 GB | 71.9 ms/tok | 3,562 tok/s | OOM |
| **GQA ( $K = 8$, 1:5)** | 168 MB | 64 | 33.4 GB | 10.0 ms/tok | 6,417 tok/s | Yes |
| **GQA ( $K = 8$, 1:5)** | 168 MB | 256 | 65.6 GB | 19.6 ms/tok | 13,068 tok/s | Yes |

**Interpretation:** Cutting KV size $5\times$ with GQA simultaneously lowers latency, frees HBM capacity, and unlocks $B = 256$ at $13{,}068$ tok/s—nearly $5\times$ the maximum throughput achievable on one H100 under MHA.

Now move from a 2023 dense model to a 2026 frontier model on our reference GPU.

WORKED EXAMPLE

**Worked Example: 2026 Frontier Model — GLM-5.3 on Blackwell B300**

From the GLM-5.3 architecture specification: $L = 78$ layers, hidden dimension $D = 6144$, $N = 64$ query heads, Multi-Head Latent Attention (MLA) caching $512 + 64 = 576$ values per token per layer, DSA sparse attention selecting top-$2048$ tokens, $256$ Mixture-of-Experts (MoE) routed experts ($8$ active plus $1$ shared expert, $F_{\text{moe}} = 2048$), and $1$ Multi-Token Prediction (MTP) layer. Let us compute the KV cache bytes per token across all $78$ layers under three designs:

1. **Hypothetical Full MHA ($NH = D = 6144$, BF16):**

$$2\ (\text{K and V}) \times 6144 \times 78\ \text{layers} \times 2\ \text{B} = 1{,}916{,}928\ \text{B/token} \approx 1{,}917\ \text{KB}\ (\ 1.87\ \text{MB}$$

2. **GLM-5.3 MLA (BF16):** Instead of storing $2 \times 6144 = 12{,}288$ scalars per layer, MLA stores one shared latent of $576$ scalars per layer:

$$576 \times 78\ \text{layers} \times 2\ \text{B} = 89{,}856\ \text{B/token} \approx 89.9\ \text{KB}\ (\ 88\ \text{KiB/token})$$

   This is $1{,}916{,}928 / 89{,}856 = 21.3\times$ smaller than MHA BF16.

3. **GLM-5.3 MLA (FP8, $1\ \mathrm{B}/\mathrm{scalar}$):**

$$576 \times 78 \text{ layers} \times 1\ \mathrm{B} = 44{,}928\ \mathrm{B}/\mathrm{token} \approx 44.9\ \mathrm{KB}\ (\,44\ \mathrm{KiB}/\mathrm{token})$$

This is $42.7\times$ smaller than MHA BF16.

| KV Cache Design (78 layers) | Formula (bytes / token) | Size / token | vs. MHA BF16 |
|---|---|---|---|
| Full MHA ($NH = D = 6144$, BF16) | $2 \times 6144 \times 78 \times 2$ | $1{,}917$ KB (~1.87 MB) | $1.0\times$ |
| **GLM-5.3 MLA (BF16)** | $576 \times 78 \times 2$ | $89.9$ KB (~88 KiB) | $21.3\times$ **smaller** |
| **GLM-5.3 MLA (FP8)** | $576 \times 78 \times 1$ | $44.9$ KB (~44 KiB) | $42.7\times$ **smaller** |

**Interpretation:** Three major systems facts follow immediately from this specification. First, **multi-GPU serving is mandatory**: GLM-5.3's checkpoint is roughly $704$ GB on disk, so the model weights alone exceed a single $288$ GB ($287$ GB usable) Blackwell B300. Second, because of **DSA sparse attention**, once context length passes $2048$ tokens, the per-token attention-core cost stops growing with context length; only a lightweight indexer continues to scan the full context. Third, combining **FP8 KV cache with NVFP4 weights** is the standard B300 serving setup, which relieves KV capacity pressure so dramatically that the dominant bottleneck shifts to **MoE weight bandwidth**.

Why does MoE weight bandwidth become such a severe bottleneck during decode? Recall from Section 2 that for a dense MLP, batching $B = 256$ tokens raises arithmetic intensity to $I_{\text{MLP}} \approx 256$, bringing the matrix multiplication near the compute-bound roof. In a sparse Mixture-of-Experts layer, however, each token does *not* pass through the same MLP weights—it is routed to only $k$ out of $E$ total experts.

Trace what happens to a decode batch of $B = 256$ tokens entering a layer with $E = 256$ routed experts and a top-$8$ router ($k = 8$), as in GLM-5.3:

- On the left, a decode batch of $B = 256$ tokens enters the **Top-8 Router**.
- Because each token selects $8$ experts, the router emits $256 \times 8 = 2{,}048$ total token-to-expert assignments.
- Those $2{,}048$ assignments fan out across all $256$ distinct expert MLPs (`Expert 0`, `Expert 1`, `Expert 2`, …, `Expert 255`).
- Under uniform routing, every single expert receives on average only:

$$\text{tokens} \,/\, \text{expert} = \frac{B \cdot k}{E}$$

For $B = 256$, $k = 8$, and $E = 256$:

$$\text{tokens / expert} = \frac{256 \times 8}{256} = \frac{B}{32} = 8 \text{ tokens per expert}$$

**COMMON MISCONCEPTION**

You might think that running a large decode batch like $B = 256$ makes MoE feed-forward layers compute-bound just as it does for a dense model. That breaks because routing fractures the batch across $E$ distinct weight matrices: every expert still has to stream its entire set of weights from HBM, yet at $B = 256$ with $E = 256$ and $k = 8$, each expert GEMM sees an effective batch size of only $8$ tokens—deeply memory-bound and far below the hardware ridge point.

**WORKED EXAMPLE**

**Worked Example: MoE Batch Dilution in DeepSeek-V4.1-Flash**

DeepSeek-V4.1-Flash uses $E = 384$ routed experts with $k = 6$ active per token. Suppose a single GPU runs a decode batch of $B = 256$ tokens.

1. Total token-expert assignments: $B \cdot k = 256 \times 6 = 1{,}536$.
2. Average tokens per expert:

$$\text{tokens / expert} = \frac{1{,}536}{384} = 4 \text{ tokens per expert}$$

3. Suppose we wanted each expert GEMM to reach an effective batch size of $128$ tokens. The required global decode batch would be:

$$B_{\text{req}} = \frac{128 \times 384}{6} = 8{,}192 \text{ concurrent decode tokens}$$

**Interpretation:** Fine-grained MoE architectures dilute per-expert batch size by the factor $k/E$ ( $6/384 = 1/64$ here). Reaching high utilization on expert GEMMs is impossible on a single GPU's local batch; we must pool decode batches across many GPUs via **wide expert parallelism**, which we will analyze in Section 8.

## Three Levers to Accelerate Inference

Everything we have derived so far—the roofline, KV cache scaling, and MoE batch collapse—organizes all modern inference optimizations into three overarching levers, which structure the remainder of this lecture:

1. **Lossy Shortcuts (Sections 4 and 5):** Shrink bytes and FLOPs per token without hurting model quality, using KV-cache architectures (GQA, MLA, CLA, sparse attention, and hybrid recurrent layers), quantization (NVFP4 weights, FP8/FP4 KV cache), and pruning plus distillation. Two recipes produce these faster models: train a faster architecture from scratch, or initialize from a larger model and repair it via distillation.
2. **Lossless Shortcuts (Section 6):** Preserve the *exact* target output distribution while executing fewer memory-bound decode steps, using speculative decoding (MTP heads, draft trees, and block-parallel

drafters) to draft $k$ tokens cheaply and verify them in one parallel pass.

3. **Dynamic Serving Systems (Sections 7 and 8):** Saturate Blackwell clusters under messy live traffic, using continuous batching, chunked prefill, Paged/Radix/Hybrid KV cache managers, prefill/decode disaggregation, wide expert parallelism, overlap scheduling, and cache-aware routing.

> **KEY IDEA**
> In memory-bound decode, throughput is bounded by $\mathrm{BW}/(\mathrm{KV}_{\mathrm{seq}} + \mathrm{Params}_{\mathrm{bytes}}/B)$—exposing a direct tradeoff between batch latency and aggregate throughput—while sparse MoE routing dilutes each expert's effective batch size by $k/E$, keeping expert matrix multiplications deeply memory-bound unless batches are pooled across GPUs.

We start with Lever 1: how do modern Transformer architectures shrink KV cache capacity and raise decode attention intensity before we even touch quantization or serving systems?

> **✎ Check Your Understanding**
>
> 1. A memory-bound model has $\mathrm{Params}_{\mathrm{bytes}} = 20\ \mathrm{GB}$, $\mathrm{KV}_{\mathrm{seq}} = 0.5\ \mathrm{GB}$, and runs on a GPU with $\mathrm{BW} = 4000\ \mathrm{GB/s}$. Compute the single-step latency and aggregate throughput at batch size $B = 20$.
>
>    **Answer:** Total memory streamed is $\mathrm{Memory} = 20 \times 0.5 + 20 = 30\ \mathrm{GB}$. Latency is $30/4000 = 7.5\ \mathrm{ms/tok}$, and aggregate throughput is $20/0.0075 = 2{,}667\ \mathrm{tok/s}$.
>
> 2. True or False: Replicating a model across $M$ GPUs (data replication) cuts single-request decode latency by a factor of $M$. Explain briefly.
>
>    **Answer:** False. Data replication multiplies aggregate throughput by $M$ by serving $M$ independent batches, while single-request latency remains unchanged; cutting single-request latency requires sharding weights and KV across GPUs.

## Section 4: Architectural KV-Cache Compression — GQA, MLA, CLA, Sparse Attention, and Recurrent Hybrids

Section 2 established that standard Multi-Head Attention (MHA) suffers a double penalty during decode: it stores $2SLNH$ KV scalars per sequence, and its decode arithmetic intensity is trapped below $1\ \mathrm{FLOP/byte}$. As context lengths pushed toward $1$ million tokens, storing and streaming full MHA caches became physically impossible. Modern architectures attack the KV cache along **three orthogonal axes**: sharing or compressing **across heads** (GQA and MLA), sharing **across layers** (CLA), and pruning or replacing storage **across sequence positions** (local attention, sparse attention, and fixed-state recurrent layers).

## Sharing Across Heads: Grouped-Query Attention (GQA) and Multi-Query Attention (MQA)

Recall from Section 2 that attention decode intensity is $I_{\text{attn}} = \frac{S(N/K)}{S+N/K} \approx N/K$ for $S \gg N/K$. Comparing the wiring between query heads and KV heads across three schemes shows how sharing transforms both memory and intensity:

- **MHA ($K = N$):** Four query heads $Q_1, Q_2, Q_3, Q_4$ each connect vertically to their own dedicated key-value head $KV_1, KV_2, KV_3, KV_4$. Every KV byte serves only one query head, yielding $1\times$ KV size and decode intensity $< 1$.
- **GQA ($1 < K < N$):** Query heads are partitioned into groups—for instance, $Q_1$ and $Q_2$ both connect to $KV_1$, while $Q_3$ and $Q_4$ both connect to $KV_2$. KV cache size drops to $K/N$ of MHA, and decode intensity jumps to $\approx N/K$.
- **MQA ($K = 1$):** All query heads $Q_1, Q_2, Q_3, Q_4$ share a single $KV_1$ head. KV cache size drops to $1/N$ of MHA, and decode intensity rises to $\approx N$.

> **DEFINITION**
> **Grouped-Query Attention (GQA)** partitions the $N$ query heads into $K$ groups ($1 < K < N$) that each share a single key head and value head, delivering a dual payoff: shrinking KV cache size by $N/K$ and raising decode attention arithmetic intensity from $< 1$ to $\approx N/K$.

GQA matches MHA quality closely while unlocking dramatically larger decode batches (as we saw in Section 3's Llama 2 13B calculation). Frontier 2026 architectures push this head-sharing ratio to its limit: **DeepSeek-V4.1-Flash** (Sept 2026) uses an extreme MQA-style ratio of $K = 1$ KV head shared across $N = 64$ query heads, paired with a wide head dimension of $H = 512$.

## Compressing Across Heads: Multi-Head Latent Attention (MLA)

Sharing raw KV heads across groups is simple, but forces all heads in a group to read the exact same key and value vectors. Can we achieve even higher compression *and* higher modeling accuracy than GQA? **Multi-Head Latent Attention (MLA)** accomplishes this through low-rank joint compression of keys and values.

> **DEFINITION**
> **Multi-Head Latent Attention (MLA)** projects each token's hidden state $h \in \mathbb{R}^D$ into a single compact low-rank latent vector $c = W_c h$ cached in HBM, then algebraically absorbs the per-head up-projection matrices into the query and output projections during decode so every query head attends directly over the shared latent.

Trace the two branches of MLA starting from an input hidden state $h$ of dimension $D$:

1. **Cached Latent Branch:** $h$ is down-projected into a compressed vector $c = W_c h$ of dimension `kv_lora_rank` ($512$ in both historical DeepSeek-V2 and 2026 frontier GLM-5.3). Conceptually, full

per-head content keys $K_C = W_{UK}c$ and values $V = W_{UV}c$ can be reconstructed from $c$ via per-head up-projection matrices $W_{UK}$ and $W_{UV}$.

2. **Cached RoPE Key Branch:** $h$ is also projected into a small decoupled positional key vector $k_R$ of dimension `qk_rope_head_dim` ($64$), to which Rotary Position Embedding ($\mathrm{RoPE}(k_R)$) is applied.

Only $c$ ($512$ values) and $k_R$ ($64$ values) are stored in the KV cache—a total of $512 + 64 = 576$ scalars per token per layer, regardless of how many query heads $N$ the model has. For historical DeepSeek-V2, this compressed $NH = 16{,}384$ down to $576$; for GLM-5.3, $576$ cached values serve all $N = 64$ query heads.

Why do we not have to reconstruct the huge $K_C$ and $V$ matrices during decode? Look at the linear algebra of a single attention score between query $q_i$ (for head $i$) and reconstructed key $K_{C,i} = W_{UK,i}c$:

$$q_i^\top K_{C,i} = q_i^\top (W_{UK,i}c) = \left(W_{UK,i}^\top q_i\right)^\top c$$

By matrix associativity, instead of expanding every past cached latent $c$ up to full key dimension, we multiply the *single* current query $q_i$ by $W_{UK,i}^\top$—folding $W_{UK}$ into $Q$—and take dot products directly against the compact cached latents $c$! Likewise, in the value-weighted sum $\sum_j \alpha_{i,j}(W_{UV,i}c_j) = W_{UV,i}\left(\sum_j \alpha_{i,j}c_j\right)$, we accumulate the weighted sum directly in the $512$-dimensional latent space and fold $W_{UV,i}$ into the output projection $W_O$. This **absorbed decode** executes directly on the cached $576$-value latent, so every loaded latent byte serves all $N$ query heads—achieving MQA-level bandwidth efficiency while empirically matching or beating full MHA accuracy (and outperforming GQA).

Why does MLA need that separate $64$-dimensional **RoPE wrinkle**? Recall that RoPE multiplies queries and keys by a position-dependent orthogonal rotation matrix $R_m$ at token position $m$. If RoPE were applied to $K_C = W_{UK}c$, the attention score would be $q^\top R_n^\top R_m W_{UK}c = q^\top R_{m-n}W_{UK}c$. Because $R_{m-n}$ depends on the relative token position $m - n$ and sits *between* $q$ and $W_{UK}$, it does not commute with $W_{UK}$—preventing us from folding $W_{UK}$ into $q$. Decoupling $64$ dedicated RoPE dimensions $k_R$ that bypass $W_{UK}$ keeps the content branch cleanly absorbable while preserving full positional awareness.

> **WORKED EXAMPLE**
>
> **Worked Example: Absorbed MLA Dimensions and Memory Ratio in GLM-5.3**
>
> In GLM-5.3, $D = 6144$, $N = 64$ heads, `kv_lora_rank` $= 512$, and `qk_rope_head_dim` $= 64$. Suppose a single decode token ($T = 1$) attends to a cached context of $S = 4096$ tokens in one layer (in BF16).
>
> 1. **Without absorption (naive reconstruction):** Expanding $c \in \mathbb{R}^{512}$ into full keys and values across $S = 4096$ tokens for $64$ heads (with $H = D/N = 96$ per head) would materialize $2 \times 4096 \times 6144 \times 2\ \mathrm{B} = 100.66\ \mathrm{MB}$ of keys and values per layer—destroying the memory-bandwidth benefit of MLA.
> 2. **With absorbed decode:**

- Transform each head's content query $q_i \in \mathbb{R}^H$ once via $\tilde{q}_i = W_{UK,i}^\top q_i \in \mathbb{R}^{512}$ (done only for the $T = 1$ active token).
- Stream the cached $[S, 576] = [4096, 576]$ latent+RoPE buffer once from HBM: $4096 \times 576 \times 2\ \text{B} = 4.72\ \text{MB}$ per layer.
- All $64$ query heads attend over those same $4.72\ \text{MB}$ of latents, then fold $W_{UV}$ into $W_O$ on the $T = 1$ output vector.

**Interpretation:** Absorption turns MLA from a storage-only compression trick into a $21.3\times$ HBM bandwidth reduction during decode while serving all $64$ query heads from a single $576$-scalar latent per token.

## Sharing Across Layers and Bounding Context: CLA, Local Attention, and 2026 Sparse Attention

Compressing across heads leaves two more axes of the KV cache ($L$ layers and $S$ sequence positions) untouched.

DEFINITION

**Cross-Layer Attention (CLA)** shares key and value projections across adjacent Transformer layers—layer $\ell$ computes and caches $Q, K, V$, while layer $\ell + 1$ computes only $Q$ and reuses layer $\ell$'s cached $K$ and $V$—extending the head-sharing principle of GQA along the depth axis.

Visualizing CLA alongside local attention clarifies the three orthogonal KV axes: on the left, layer $\ell$ holds $(Q, K, V)$ and passes its KV cache upward to layer $\ell + 1$, which holds $Q$ only; on the right, a **local (sliding-window) attention** mask shows a $4 \times 5$ attention grid where each row attends only to a diagonal band of the most recent $W$ tokens rather than the entire lower triangle.

CLA pushes out the accuracy versus KV-size Pareto frontier and is deployed in 2026 frontier designs such as DeepSeek-V4.1-Flash (which combines cross-layer KV reuse with $K = 1$ head sharing and FP4 KV storage). Local (sliding-window) attention—used in earlier architectures such as Longformer and Mistral 7B—makes a layer's KV cache size $O(W)$, completely independent of sequence length $S$, while allowing the effective receptive field to grow linearly with depth $L$. However, pure local attention degrades exact long-range recall across distant tokens, so practical models interleave local layers with global layers.

At **1M-token contexts**, even a handful of dense global layers become prohibitively slow because every decode step must attend over all $S = 1{,}000{,}000$ positions. Frontier 2026 models solve this with **dynamic sparse attention** and **compressed-block attention**. Trace the two-path sparse attention pipeline:

1. A full context of up to $1$ million tokens feeds into either (a) a **cheap indexer / selection stage** that scores all $S$ tokens cheaply and picks only the top-$k$ most relevant tokens, or (b) a **compressed-blocks stage** that summarizes every $m$ contiguous tokens into $1$ compressed representation (as in CSA/HCA).

2. The selected top-$k$ tokens enter a **bounded attention core**, where full attention runs *only* on those selected tokens. Once context length exceeds $k$, core decode attention cost stops growing with $S$.

Four 2026 frontier models illustrate how this principle is deployed in practice:

- **GLM-5.3 DSA:** A lightweight indexer scans all $S$ tokens, and full attention runs only on `index_topk = 2048` selected tokens.
- **DeepSeek-V4 and DeepSeek-V4.1-Flash (Sept 2026):** Combine **Compressed Sparse Attention (CSA / CSA2)** and **Heavily Compressed Attention (HCA)** with cross-layer KV reuse and an FP4 KV cache, reducing global KV storage to just $890\ \text{bytes/token}$ at $1\text{M}$ context.
- **MiniMax-M3 (June 2026):** Uses blockwise top-$k$ sparse attention built on GQA, matching dense GQA quality on a $109\text{B}$ model while cutting per-token attention compute by $28.4\times$ at $1\text{M}$ context.
- **Qwen3.8-Flash-Next (Aug 2026):** Converts its full-attention layers into **Qwen Sparse Attention (QSA)** during continued pretraining.

> **WORKED EXAMPLE**
> **Worked Example: 1M-Token Context KV Footprint Across Architectural Generations**
>
> Compare the single-request KV cache footprint at $S = 1{,}000{,}000$ tokens across three designs:
>
> 1. **Hypothetical 78-layer Full MHA (BF16, $D = 6144$):**
>
> $$1{,}000{,}000 \times 1{,}916{,}928\ \text{B/tok} \approx 1{,}917\ \text{GB}\ (\,1.92\ \text{TB})$$
>
> A *single* user request would require nearly seven $288\ \text{GB}$ Blackwell B300 GPUs just for its KV cache.
>
> 2. **GLM-5.3 MLA (FP8, $44{,}928\ \text{B/tok}$):**
>
> $$1{,}000{,}000 \times 44{,}928\ \text{B/tok} \approx 44.9\ \text{GB}$$
>
> A single request now fits comfortably in a fraction of one B300, though batching many 1M-token users is still tight.
>
> 3. **DeepSeek-V4.1-Flash (CSA2 + cross-layer KV reuse + FP4 KV, $890\ \text{B/tok}$):**
>
> $$1{,}000{,}000 \times 890\ \text{B/tok} = 890\ \text{MB} \approx 0.89\ \text{GB}$$
>
> **Interpretation:** Combining compression across heads, layers, positions, and numeric precision shrinks a 1M-token KV cache from nearly $2\ \text{TB}$ down to under $1\ \text{GB}$—a $> 2{,}000\times$ reduction.

## Hybrid Recurrent–Attention Architectures

A fourth architectural breakthrough goes beyond compressing attention caches: **removing attention entirely from most layers**. Linear attention and state-space layers—such as **Mamba 2** and **Gated DeltaNet**—compress the entire history of a sequence into a **fixed-size recurrent state** whose memory

footprint is completely independent of sequence length $S$. Pure recurrent models still struggle on exact verbatim recall over long histories, so 2026 frontier models interleave recurrent layers with sparse or compressed attention layers (commonly in a $3:1$ ratio):

| 2026 Frontier Model | Layer Ratio & Mechanism | MoE & Serving Notes |
|---|---|---|
| **Kimi K3 / K3.1** (July 2026) | 3 Kimi Delta Attention : 1 gated MLA (1M context) | 2.8T total / 104B active; 896 experts (16 active + 2 shared); MXFP4 QAT on experts |
| **Qwen3.8-Flash-Next** (Aug 2026) | 3 Gated DeltaNet : 1 QSA sparse attention | 125B total / 6B active; 512 experts (10 active + 1 shared); 51B n-gram embedding table in host RAM |
| **Nemotron 3.5** (Oct 2026) | Hybrid Mamba–Transformer MoE | Fixed-size Mamba state slashes KV footprint alongside sparse Transformer layers |

Replacing $75\%$ of attention layers with fixed-state recurrent layers removes most of the KV cache outright. However, it introduces a major systems complication that we will need when we get to Section 7: it breaks the classical serving assumption that every layer stores uniform KV bytes per token that can be arbitrarily sliced at any sequence position.

> **KEY IDEA**
> Modern models compress KV caches across three orthogonal axes—heads (GQA, absorbed MLA), layers (CLA), and sequence positions (local windows, top-$k$ sparse attention, compressed blocks)—and increasingly replace $75\%$ of attention layers with fixed-state recurrent layers (DeltaNet, Mamba) to make 1M-token contexts practical.

Architectural changes shrink the *count* of stored weights and KV scalars. Our next question is how to shrink the *number of bytes per scalar*—and even remove redundant structural units after training—without destroying model quality.

**✎ Check Your Understanding**

1. What would happen to absorbed MLA decode if RoPE were applied directly to the reconstructed content keys $K_C = W_{UK}c$ instead of to a separate decoupled vector $k_R$?

   **Answer:** The position-dependent RoPE rotation matrix would sit between the query and $W_{UK}$, preventing $W_{UK}$ from being algebraically folded into the query projection and forcing the model to reconstruct full keys from the cached latent on every decode step.

2. Fill in the blank: In a hybrid recurrent–attention model such as Qwen3.8-Flash-Next or Kimi K3, the recurrent layers maintain a _____ state whose memory footprint is independent of sequence length $S$.

   **Answer:** Fixed-size recurrent state.

---

## Section 5: Model Compression — Quantization, FP4 KV Caches, Pruning, and Distillation

Even after architectural KV compression, model weights and KV caches still dominate HBM capacity and bandwidth. Because memory-bound decode latency scales directly with bytes transferred ($\text{Latency} = \text{Memory}/\text{BW}$), cutting the precision of each stored scalar from $2$ bytes (BF16) to $1$ byte (FP8) or $0.5$ bytes (FP4) cuts memory traffic—and directly speeds up decode—by roughly $2\times$ to $4\times$.

### Quantization Mechanics and Blackwell Number Formats

**DEFINITION**

**Quantization** maps continuous higher-precision floating-point values onto a discrete lower-bit-width grid using a scale factor (and optionally a zero-point offset), reducing both HBM storage footprint and memory bandwidth proportionally to the bit-width reduction.

In standard affine integer quantization, converting a floating-point value $x$ to an integer $x_q$ and reconstructing its approximation $\hat{x}$ takes two lines of arithmetic:

$$x_q = \text{round}\left(\frac{x}{\text{scale}}\right) + \text{zero_point}, \qquad \hat{x} = (x_q - \text{zero_point}) \times \text{scale}$$

Here `scale` sets the step size between adjacent discrete grid points, and `zero_point` shifts the integer grid so that real value $0.0$ maps cleanly onto an exact integer value even for asymmetric distributions.

**WORKED EXAMPLE**

**Worked Example: Affine Quantization and Dequantization**

Quantize $x = 5.2342$ using $\text{scale} = 0.1$ and $\text{zero_point} = 4$:

1. **Quantize:**

$$x_q = \text{round}\left(\frac{5.2342}{0.1}\right) + 4 = \text{round}(52.342) + 4 = 52 + 4 = 56$$

2. **Dequantize:**

$$\hat{x} = (56 - 4) \times 0.1 = 52 \times 0.1 = 5.2$$

3. **Quantization error:**

$$\hat{x} - x = 5.2 - 5.2342 = -0.0342$$

**Interpretation:** Storing the integer $56$ instead of a 32-bit float loses only $-0.0342$ (at most half a step size, $\pm\text{scale}/2 = \pm 0.05$), provided `scale` is chosen small enough for resolution yet large enough to cover the tensor's dynamic range without clipping.

In modern LLM inference, choosing a number format means balancing byte width against dynamic range and hardware tensor-core support:

| Format | Bytes | Range / Role |
|---|---|---|
| **FP32** | 4 | Training master weights and optimizer states |
| **BF16** | 2 | Historical inference default |
| **FP8 E4M3** | 1 | NVIDIA OCP `e4m3fn`: $[-448, 448]$ |
| **INT8 / INT4** | 1 / 0.5 | $[-128, 127]$ / $[-8, 7]$; inference-only |
| **MXFP8 / NVFP4** | 1 / 0.5 | Blackwell native block-scaled tensor cores (NVFP4: one E4M3 scale per 16 values) |

**COMMON MISCONCEPTION**

You might think that FP8 E4M3 always has a maximum representable magnitude of $\pm 240$. That breaks on NVIDIA GPUs because $\pm 240$ is the range of the `e4m3fnuz` variant; NVIDIA's OCP `e4m3fn` format represents values across $[-448, 448]$. Confusing the two causes silent overflow or overly conservative scaling when writing or debugging FP8 quantization routines.

Why do **MXFP8** and **NVFP4** matter so much on Blackwell B300? Earlier 4-bit integer formats needed software dequantization back to BF16 before matrix multiplication, or suffered severe precision loss when a single large outlier forced a huge shared scale across an entire tensor or row. Blackwell introduces native **block-scaled tensor cores**: in NVFP4, every tiny block of just 16 4-bit values shares its own FP8

E4M3 scale factor. Because the hardware consumes the block scales natively during matrix multiplication, **NVFP4 weights paired with an FP8 KV cache** is the standard serving configuration on Blackwell B300, not an exotic optimization.

> **WORKED EXAMPLE**
>
> **Worked Example: Effective Weight Bandwidth of NVFP4 vs. BF16**
>
> Compute the exact average bytes per weight for NVFP4 (4-bit values with one 1-byte FP8 E4M3 scale per $16$ values) and determine the weight-streaming speedup over BF16.
>
> 1. **Bytes per 16-value block:**
>    - $16$ FP4 values $\times 0.5\ \text{B/value} = 8$ bytes.
>    - $1$ E4M3 block scale $= 1$ byte.
>    - Total per block $= 9$ bytes.
> 2. **Average bytes per weight:**
>
> $$\frac{9\ \text{bytes}}{16\ \text{weights}} = 0.5625\ \text{B/weight} \quad (4.5\ \text{bits/weight})$$
>
> 3. **Compression ratio vs. BF16 ($2\ \text{B/weight}$):**
>
> $$\frac{2.0}{0.5625} \approx 3.56\times$$
>
> **Interpretation:** Even accounting for the per-16-element E4M3 scale metadata, NVFP4 cuts model weight storage and HBM weight-streaming traffic by $3.56\times$ relative to BF16 while keeping local dynamic range tight across every $16$ weights.

## Quantization Methods: QAT, PTQ, GPTQ, and AWQ

Simply rounding trained weights to 4 bits or 3 bits destroys accuracy because neural network activations contain rare, extreme **outlier channels**. Two broad families of quantization methods prevent this collapse:

1. **Quantization-Aware Training (QAT):** During training, simulated quantize-and-dequantize operations are inserted into the forward pass so gradient updates teach the weights to be resilient to rounding noise. For example, Kimi K3 trains its MoE experts directly with MXFP4 QAT. QAT achieves the highest accuracy, at the cost of requiring large-scale training compute.
2. **Post-Training Quantization (PTQ):** Applies cheap calibration *after* training is finished using a small sample of data. Two classic PTQ methods show how to preserve accuracy without retraining:
   - **GPTQ:** Uses second-order Hessian curvature information to sequentially quantize weights while updating the remaining unquantized weights to compensate for the quantization error already introduced.
   - **AWQ (Activation-Aware Weight Quantization):** Observes that not all weights are equally important—only the $0.1$–$1\%$ **of salient weights** that multiply against large-magnitude activation

outlier channels dominate output error.

Picture the AWQ mechanism visually: across the top sits a bar chart of activation channel magnitudes $|X|$, where five channels have short blue bars and two channels spike upward as tall red **outlier** bars. Directly underneath sits the weight matrix $W$ partitioned into matching columns; the two columns aligned with the red activation outliers are highlighted in yellow with red borders. Rather than storing those columns in a different precision (which complicates hardware execution), AWQ protects those $0.1$–$1\%$ salient weight columns by scaling them up by a factor $s > 1$ before quantization and inversely scaling the corresponding activation channels by $1/s$. On FP16 $\rightarrow$ INT3, AWQ achieves $4\times$ less memory and $3.2\times$ speedup while preserving accuracy.

**WORKED EXAMPLE**

**Worked Example: Why AWQ Per-Channel Scaling Protects Activation Outliers**

Consider a 2-element dot product $y = w_1 x_1 + w_2 x_2$ where $w_1 = 0.24$, $w_2 = 0.24$, $x_1 = 1.0$ (normal channel), and $x_2 = 100.0$ (outlier channel). The exact result is $y = 0.24(1) + 0.24(100) = 24.24$. Suppose we quantize both weights jointly with a single shared $\text{scale} = 0.1$ and $\text{zero_point} = 0$.

1. **Naive quantization (no AWQ scaling):**
   - Both $w_1$ and $w_2$ round to $\hat{w}_1 = \hat{w}_2 = \text{round}(0.24/0.1) \times 0.1 = 0.20$ (error $-0.04$ each).
   - Output: $\hat{y} = 0.20(1) + 0.20(100) = 20.20$.
   - Output error: $|20.20 - 24.24| = 4.04$, almost entirely caused by the $-0.04$ error on $w_2$ getting amplified $100\times$ by $x_2$!
2. **AWQ protection (scale salient channel 2 by $s = 4$, divide $x_2$ by 4):**
   - Scaled weights: $w'_1 = 0.24$, $w'_2 = 4 \times 0.24 = 0.96$. Scaled activations: $x'_1 = 1.0$, $x'_2 = 100/4 = 25.0$.
   - Quantize with the same $\text{scale} = 0.1$: $\hat{w}'_1 = 0.20$, $\hat{w}'_2 = \text{round}(0.96/0.1) \times 0.1 = 1.00$ (error $+0.04$ on the scaled weight, which is only $+0.04/4 = +0.01$ on the original weight!).
   - Output: $\hat{y} = 0.20(1.0) + 1.00(25.0) = 25.20$.
   - Output error drops from $4.04$ down to $|25.20 - 24.24| = 0.96$—a $4.2\times$ error reduction with zero change to the quantization bit-width!

**Interpretation:** Multiplying a salient weight column by $s$ before quantization reduces its relative rounding error by $s\times$ (provided the shared scale does not grow proportionally), neutralizing the amplification caused by activation outliers.

## FP4 KV Cache: Memory Win vs. Long-Context Risk

Given the success of NVFP4 for model weights, should we also quantize the KV cache below FP8 down to FP4? An **NVFP4 KV cache** halves KV memory footprint and HBM traffic relative to FP8 KV—and is already used in DeepSeek-V4.1-Flash to hit its $890\ \mathrm{B/token}$ global KV footprint. In production implementations, KV entries remain stored as FP4 in HBM and are **dequantized to FP8 inside the attention kernel** on the fly.

However, long-context retrieval accuracy under FP4 KV quantization is actively disputed in the literature:

- **NVIDIA (vendor-reported):** $< 1\%$ accuracy loss across its benchmark suite.
- **BitDecoding-FP4 (paper-reported):** Up to $8.5$ **points lost** on long-context retrieval benchmarks (RULER and MRCR) when quantization scales are uncalibrated.

> **COMMON MISCONCEPTION**
> You might think that if an FP4 KV cache shows $< 1\%$ perplexity or accuracy drop on standard short-context evaluations, it is safe to deploy in production. That breaks because quantization noise in keys and values accumulates selectively on exact needle retrieval across tens or hundreds of thousands of tokens. The engineering rule is strict: never validate KV quantization on short-context benchmarks alone—always test needle-in-a-haystack and long-context retrieval benchmarks such as RULER and MRCR.

## Structured Pruning and Knowledge Distillation

Quantization compresses every parameter to fewer bits; **pruning and distillation** compress the model by physically removing whole structural components and repairing the resulting damage. Trace the three-step NVIDIA "initialize & repair" pipeline:

1. **Score Importance:** Pass a small calibration set ($1{,}024$ calibration samples) through the original large **teacher** model to score the importance of entire Transformer layers, attention heads, and hidden MLP channels.
2. **Prune:** Drop the weakest units—removing whole layers, heads, or channels. This is **structured pruning**: unlike unstructured pruning (which zeroes out isolated individual weights in a scattered sparse pattern that tensor cores cannot easily accelerate), structured pruning leaves clean, smaller dense matrices that run at full efficiency on GPU tensor cores.
3. **Distill (Repair):** Train the pruned student model using supervision from the original teacher model so the student matches the teacher's output logits.

> **DEFINITION**
> **Structured pruning with knowledge distillation** ("initialize and repair") removes entire layers, attention heads, or hidden dimensions from a trained teacher model based on calibration importance scores, then trains the smaller dense model to match the teacher's logits—achieving better quality per training-compute dollar than training a small model from scratch.

**KEY IDEA**

Low-precision block-scaled formats (NVFP4 weights + FP8 KV on Blackwell B300), outlier-aware quantization (QAT, GPTQ, AWQ), and structured pruning with distillation slash memory bytes per decode step—though aggressive FP4 KV compression must always be audited on long-context retrieval benchmarks like RULER and MRCR.

Every method in Sections 4 and 5 is **lossy**: it alters the model's architecture, precision, or parameter count, requiring careful quality verification. Can we accelerate memory-bound decode *without changing the model's output probability distribution by even a single bit*? That brings us to Lever 2: speculative decoding.

**✎ Check Your Understanding**

1. Using affine quantization with $\text{scale} = 0.25$ and $\text{zero_point} = 2$, compute the quantized integer $x_q$ and reconstructed value $\hat{x}$ for $x = 1.6$.

   **Answer:** $x_q = \text{round}(1.6/0.25) + 2 = \text{round}(6.4) + 2 = 6 + 2 = 8$, and $\hat{x} = (8 - 2) \times 0.25 = 1.5$.

2. Why does structured pruning remove entire layers, heads, or hidden channels rather than individual scattered weights?

   **Answer:** Removing entire structural units produces smaller, dense weight matrices that execute efficiently on GPU tensor cores, whereas scattered unstructured zeros do not map cleanly to dense tensor-core matrix multiplies.

---

## Section 6: Lossless Acceleration via Speculative Decoding

Recall the core physical insight from Section 2: during memory-bound decode at small batch size $B$, reading the model's weights from HBM takes almost the entire step time while the GPU's tensor cores sit largely idle. Look at the arithmetic intensity formula for an MLP: $I_{\text{MLP}} \approx B \cdot T$. Standard decode sets $T = 1$, leaving enormous headroom between $I \approx B$ and the compute ridge point $I^*$. That means running a target model forward pass on $T = k + 1$ tokens in parallel costs nearly the *same* wall-clock time as running it on $T = 1$ token!

**Speculative decoding** exploits this asymmetry—**checking beats generating**. Compare standard decode against speculative decode on a four-token timeline:

- **Standard decode:** Runs four full, sequential target-model passes ($q \to x_1$, $q \to x_2$, $q \to x_3$, $q \to x_4$), paying four full target-model weight reads from HBM.
- **Speculative decode:** First, a tiny, fast **draft model** $p$ autoregressively guesses $k = 4$ candidate tokens $\tilde{x}_1, \tilde{x}_2, \tilde{x}_3, \tilde{x}_4$ at a tiny fraction of the cost. Then, the large **target model** $q$ scores all $k + 1$ positions

$[\tilde{x}_1, \tilde{x}_2, \tilde{x}_3, \tilde{x}_4]$ in a *single* parallel forward pass, taking roughly the wall-clock time of one decode step.

**DEFINITION**

**Speculative decoding** accelerates autoregressive generation without altering the target model's output distribution by using a cheap drafter $p$ to propose $k$ candidate tokens sequentially and then verifying all $k$ candidates in parallel with one forward pass of the target model $q$.

## Modified Rejection Sampling and Exactness Proof

Naively accepting draft tokens only when they match the target's argmax only works for greedy decoding (temperature $0$). In general stochastic sampling, we need a verification rule that guarantees the emitted tokens follow the *exact* probability distribution $q$, even when $p$ and $q$ disagree. **Modified rejection sampling** achieves this in three steps:

1. **Draft & parallel score:** The draft model $p$ proposes $k$ tokens $\tilde{x}_1, \ldots, \tilde{x}_k$, recording their draft probabilities $p_i(\tilde{x}_i)$. The target model $q$ runs one parallel forward pass over the prefix plus $\tilde{x}_1, \ldots, \tilde{x}_k$, yielding exact target distributions $q_1, \ldots, q_{k+1}$ at all $k+1$ positions.
2. **Sequential acceptance test:** For each position $i = 1, \ldots, k$ in order, accept $\tilde{x}_i$ with probability:

$$\alpha_i = \min\left(1,\ \frac{q_i(\tilde{x}_i)}{p_i(\tilde{x}_i)}\right)$$

   Notice the two cases: if the draft model *undersampled* $\tilde{x}_i$ relative to the target ($p_i(\tilde{x}_i) \leq q_i(\tilde{x}_i)$), then $q_i/p_i \geq 1$, so we accept with probability $1$. If the draft model *oversampled* $\tilde{x}_i$ ($p_i(\tilde{x}_i) > q_i(\tilde{x}_i)$), we trim the excess probability by accepting only a fraction $q_i(\tilde{x}_i)/p_i(\tilde{x}_i)$ of the time.
3. **Residual correction or bonus token:** On the *first* rejected token at position $i$, discard $\tilde{x}_i, \ldots, \tilde{x}_k$, resample a replacement token $x_i$ from the normalized **residual distribution**

$$q_i'(x) = \text{norm}\big(\max(q_i(x) - p_i(x),\ 0)\big) = \frac{\max(q_i(x) - p_i(x),\ 0)}{\sum_v \max(q_i(v) - p_i(v),\ 0)}$$

   and stop verifying. If all $k$ draft tokens are accepted, sample one extra **bonus token** $x_{k+1} \sim q_{k+1}$ (which the target pass already computed for free!).

Let us prove formally why this produces an exact sample from $q$ at a given position on a two-token vocabulary $\{A, B\}$. Suppose the draft oversamples $A$, meaning $p(A) > q(A)$, which forces $p(B) < q(B)$ since probabilities sum to $1$.

- Examine the residual distribution $\max(q - p, 0)$:
  - For token $A$: $q(A) - p(A) < 0$, so $\max(q(A) - p(A), 0) = 0$.
  - For token $B$: $q(B) - p(B) > 0$, so after normalization the residual places probability $0$ on $A$ and probability $1$ on $B$.

- Now compute the total probability of outputting $A$ (either drafted and accepted, or drafted $B$, rejected, and resampled as $A$):

$$P(\text{sample } A) = \underbrace{p(A) \cdot \frac{q(A)}{p(A)}}_{\text{draft } A \text{ and accept}} + \underbrace{p(B) \cdot (1-1) \cdot 0}_{\text{draft } B \text{ and reject}} = q(A)$$

- Next compute the total probability of outputting $B$ (either drafted $B$ and accepted with probability $1$, or drafted $A$, rejected with probability $1 - q(A)/p(A)$, and resampled as $B$ with residual probability $1$):

$$P(\text{sample } B) = \underbrace{p(B) \cdot 1}_{\text{draft } B \text{ and accept}} + \underbrace{p(A) \cdot \left(1 - \frac{q(A)}{p(A)}\right) \cdot 1}_{\text{draft } A\text{, reject, resample } B} = p(B) + p(A) - q(A) = 1 - q($$

Every token's marginal probability matches $q$ identically. Notice also a crucial advantage over classical textbook rejection sampling: classical rejection sampling loops repeatedly until an accept occurs, whereas modified rejection sampling replaces a rejection immediately using the residual distribution, guaranteeing **at least $1$ exact target token per verification pass** (and up to $k+1$ tokens when all $k$ drafts succeed).

> **WORKED EXAMPLE**
>
> **Worked Example: Modified Rejection Sampling on a 3-Token Vocabulary**
>
> Suppose at a single position the vocabulary is $\{A, B, C\}$ with target distribution $q = (0.30, 0.50, 0.20)$ and draft distribution $p = (0.60, 0.30, 0.10)$.
>
> 1. **Acceptance probabilities** $\min(1, q(x)/p(x))$**:**
>    - For $A$: $\min(1, 0.30/0.60) = 0.50$ (rejects $50\%$ of the time).
>    - For $B$: $\min(1, 0.50/0.30) = 1.00$ (always accepted).
>    - For $C$: $\min(1, 0.20/0.10) = 1.00$ (always accepted).
> 2. **Total rejection probability:** Rejection only occurs when $A$ is drafted and then rejected:
>
> $$P(\text{reject}) = p(A)(1 - 0.50) = 0.60 \times 0.50 = 0.30$$
>
> 3. **Residual distribution** $q'(x) = \text{norm}(\max(q(x) - p(x), 0))$**:**
>    - Unnormalized $\max(q - p, 0) = (\max(-0.30, 0),\ \max(0.20, 0),\ \max(0.10, 0)) = (0,\ 0.20,\ 0.10)$.
>    - Sum of positive deficits: $0.20 + 0.10 = 0.30$ (notice this *always* equals $P(\text{reject})$!).
>    - Normalized residual: $q' = (0,\ 2/3,\ 1/3)$.
> 4. **Verify exact output probabilities:**
>    - $P(\text{output } A) = 0.60(0.50) + 0.30(0) = 0.30 = q(A)$.

- $P(\text{output } B) = 0.30(1.00) + 0.30(2/3) = 0.30 + 0.20 = 0.50 = q(B)$.
- $P(\text{output } C) = 0.10(1.00) + 0.30(1/3) = 0.10 + 0.10 = 0.20 = q(C)$.

**Interpretation:** The excess probability shaved off oversampled tokens ($0.30$ from $A$) equals the exact sum of probability deficits across undersampled tokens ($0.20$ for $B$ and $0.10$ for $C$), and the residual distribution redistributes that shaved mass in exact proportion to each deficit.

### Speculative Decoding as Deployed: EAGLE, MTP, Draft Trees, and Late-2026 Drafters

Early speculative decoding paired two completely separate models—for example, a $70\text{B}$ target model with an $8\text{B}$ draft model, or an $8\text{B}$ target with a $1\text{B}$ drafter. Managing a separate draft model complicates serving because it requires maintaining a second KV cache, synchronizing vocabulary tokenizers, and tuning alignment between two independent networks. Drafter architectures therefore evolved rapidly along a clear progression:

$$\text{Standalone small draft model} \longrightarrow \textbf{Medusa}\ (\text{parallel heads}) \longrightarrow \textbf{EAGLE}\ (\text{uses target hi}$$

In **EAGLE** and **Multi-Token Prediction (MTP)** layers, a lightweight draft head taps directly into the target model's rich final hidden state $h_t$. Because $h_t$ already encodes the entire context history computed by the large target model, the lightweight head predicts future tokens far more accurately than a small standalone model while eliminating the need to manage a separate KV-heavy draft model. Native MTP layers are now shipped directly inside frontier checkpoints: **GLM-5.3** includes $1$ built-in MTP layer, and **DeepSeek-V4.1-Flash** includes $3$ built-in MTP layers.

Production engines amplify acceptance rates even further using **draft-tree verification**. Instead of proposing a single linear chain of $k$ tokens, the drafter proposes a branching tree of candidate continuations: for example, from root token $t$, the tree branches to two first-step candidates $a_1$ and $b_1$; $a_1$ branches further to $a_2$ and $a_2'$, while $b_1$ continues to $b_2$. All tree nodes are flattened into one sequence and verified simultaneously in a *single* target forward pass using a custom **tree attention mask** so each node attends only to its ancestors along its own branch. After verification, the longest accepted path (say $t \to a_1 \to a_2$) is committed to the KV cache and all rejected branches are discarded.

Late-2026 speculative systems push three further advances:

- **Block-parallel drafting (DFlash):** Instead of running the draft head sequentially $k$ times to produce a draft chain, DFlash drafts an entire block of candidate tokens in *one* forward pass by injecting target hidden states into every draft layer. In SGLang, LMSYS self-reported $> 4.3\times$ baseline throughput and $1.5\times$ over $7$-step MTP on Qwen3.5-397B at concurrency $1$.
- **Adaptive draft length (DSpark):** Rather than using a fixed draft length $k$ for every step (which wastes verification FLOPs on hard tokens and leaves speed on the table on easy tokens), DSpark estimates each draft token's survival probability to choose dynamically how many tokens to verify per request. In SGLang, DeepSeek self-reported $60$–$85\%$ higher per-user speed than MTP-1 in DeepSeek-V4 serving.
- **Overlapped speculative scheduling (SGLang Spec V2):** Speculative decoding adds host bookkeeping (acceptance checks, tree mask building, KV rollback/allocation). Spec V2 overlaps this

speculative bookkeeping and KV allocation for batch $t$ with GPU execution on batch $t+1$ (self-reported $+33\%$ on Qwen3-8B on one B200 at concurrency $32$).

**COMMON MISCONCEPTION**

You might think that a $4\times$ speculative decoding speedup measured at batch size $1$ translates directly into a $4\times$ throughput gain on a heavily loaded production cluster. That breaks because speculative decoding relies on spare compute headroom below the roofline ridge point ($I_{\text{MLP}} \approx B \cdot T < I^*$). As batch size $B$ grows and decode approaches the compute roof, verifying $k+1$ tokens per request pushes total active tokens $B(k+1)$ into the compute-bound regime, where verification is no longer "free" and speculative gains shrink sharply. Always benchmark speculative speedups at matched temperature, dataset, and concurrency.

**KEY IDEA**

Speculative decoding converts idle tensor-core capacity during memory-bound decode into lossless speedups via modified rejection sampling (guaranteed exact to $q$ with $\geq 1$ token per pass), evolving from separate draft models to target-conditioned EAGLE/MTP heads, draft trees, and late-2026 block-parallel/adaptive drafters.

Everything in Sections 1 through 6 assumed clean, uniform batches of requests starting and ending together. Live production traffic never looks like that. To actually deliver these speedups to real users and agents, we must turn to Lever 3: dynamic serving systems.

### ✎ Check Your Understanding

1. At a given decode position with vocabulary $\{A, B\}$, suppose $q(A) = 0.25, q(B) = 0.75$ and the draft model proposes $p(A) = 0.50, p(B) = 0.50$. If the draft model proposes token $A$, what is its acceptance probability, and what token is emitted if it is rejected?

   **Answer:** The acceptance probability is $\min(1, q(A)/p(A)) = \min(1, 0.25/0.50) = 0.50$. If rejected, the residual $\max(q-p, 0)$ is $(0, 0.25)$, which normalizes to $(0, 1)$, so token $B$ is emitted with probability $1$.

2. What would happen to the speedup of speculative decoding if you increased request concurrency (batch size $B$) until standard decode was already near the GPU's compute-bound ridge point?

   **Answer:** The speculative speedup would shrink dramatically (or vanish), because verification of $k+1$ tokens per request would be compute-bound rather than hidden inside idle memory-bound wait time.

# Section 7: Serving Live Traffic — Batching and KV-Cache Memory Management

During training, data loaders pack tokens into clean, rectangular tensors of shape `[B, S, D]` where every sequence starts at the same time and runs for the same length. Live inference traffic breaks every single one of those assumptions. Production serving faces three interlocking dynamic challenges:

1. **Asynchronous arrivals and finishes:** User requests arrive at unpredictable times and terminate at wildly different output lengths. Waiting to accumulate a full static batch—or holding finished requests until the longest sequence in a batch stops—stalls early requests and leaves hardware idle.
2. **Variable sequence lengths:** Prompts and active generations have ragged lengths. Padding short sequences with dummy tokens to form a rectangular tensor wastes both FLOPs and HBM bandwidth.
3. **Shared prefixes:** System prompts, multi-turn chat histories, long agent trajectories, and parallel sampling branches all share massive common prefixes. Recomputing prefill for those identical prefixes on every turn wastes huge amounts of compute.

Modern serving engines—using SGLang and vLLM as our primary systems examples—solve these three challenges through a stack of scheduling and KV-memory techniques.

## Continuous Batching, Selective Batching, and Chunked Prefill

First consider what goes wrong under **static batching**. Picture three requests running together: Request 1 finishes early, Request 2 generates a very long response, and Request 3 finishes even earlier than Request 1. Under static batching, the slots occupied by Requests 1 and 3 sit filled with dashed red **idle padding** for the entire remaining duration of Request 2; no newly arriving request can start until Request 2 completes.

**Continuous batching** (also called **iteration-level scheduling**, introduced by Orca) eliminates this batch barrier. Instead of scheduling at the granularity of whole requests, the engine schedules at the granularity of every *single decode iteration*: the exact step that Request 1 finishes, arriving Request 4 immediately takes its slot; the exact step Request 3 finishes, Request 5 immediately takes its slot.

> **DEFINITION**
> **Iteration-level scheduling (continuous batching)** updates the active batch on every single decode step—evicting finished requests and inserting waiting requests immediately without waiting for other sequences in the batch to complete.

Once requests enter and leave asynchronously, they inevitably have different sequence lengths—for instance, three active sequences of lengths $3$, $9$, and $5$ with tensor shapes $[3, H]$, $[9, H]$, and $[5, H]$. **Selective batching** handles these ragged shapes without padding by splitting Transformer operations into two categories:

- **Token-wise linear and MLP operations:** Concatenate all active tokens back-to-back into a single dense packed tensor of shape $[3 + 9 + 5, H] = [17, H]$ with zero padding, and run one large matrix multiply.

- **Attention operations:** Because tokens only attend within their own sequence, run attention per sequence against each request's own KV cache.

> WORKED EXAMPLE
> **Worked Example: Padding Waste vs. Selective Batching**
>
> Suppose four active requests in a mixed iteration have active token counts $1, 1, 1$, and $128$.
>
> 1. **Padded rectangular batch:** Every sequence must be padded to the maximum length $128$, forming a tensor with $4 \times 128 = 512$ token slots. Only $1 + 1 + 1 + 128 = 131$ tokens are real; $381$ slots ($74.4\%$) are wasted padding!
> 2. **Selective batching:** Pack the real tokens tightly into a $[131, D]$ tensor for all linear projections and MLP layers ($0\%$ padding waste), while executing attention per sequence against each sequence's KV cache.
>
> **Interpretation:** Selective batching lets requests of completely different active lengths share the same weight-matrix fetches with zero padding waste.

Continuous batching alone still suffers a severe latency glitch whenever a new request arrives with a very long prompt. Visualize the timeline without chunking: decode steps `D` run rapidly until a long prompt arrives and executes as one giant, uninterrupted **Long Prefill** block, freezing every active decode request and causing a massive spike in TPOT!

**Chunked prefill with mixed batching** resolves this interference by exploiting both selective batching and the roofline physics from Section 2. Instead of running a long prompt all at once, the scheduler slices the prompt into fixed-size chunks (`Prefill chunk 1`, `Prefill chunk 2`, `Prefill chunk 3`, `Prefill chunk 4`) and packs *one* prefill chunk alongside the active decode batch in each iteration.

> DEFINITION
> **Chunked prefill with mixed batching** splits long prompts into fixed-size token chunks and executes each chunk in the same forward pass as active decode tokens, bounding decode TPOT while filling idle tensor cores on the same weight fetch.

Look at the **roofline synergy** of mixed batching: decode tokens are memory-bound (low $T = 1$, high weight-read overhead), while prefill chunks are compute-bound. Packing them into the same iteration means *one single read of the model's weights from HBM serves both phases simultaneously*, piggybacking compute-heavy prefill tokens onto the idle tensor cores of a memory-bound decode step. Chunked prefill is standard in both SGLang and vLLM.

## PagedAttention: Virtual Memory for the KV Cache

Even with optimal batch scheduling, naive KV cache memory allocation wastes most of the GPU's HBM. Because you do not know in advance how many tokens a request will generate, a naive engine must pre-allocate a single contiguous KV buffer sized for the maximum possible sequence length for every request. This causes two classic memory failures:

- **Internal fragmentation:** Reserved space at the end of a request's contiguous buffer that the request never uses (for example, reserving $4096$ tokens for a reply that stops at $150$ tokens).
- **External fragmentation:** Scattered free gaps between active contiguous buffers that are individually too small to fit a new request's contiguous allocation even when total free HBM is large.

**PagedAttention** (introduced in vLLM) solves both forms of fragmentation by borrowing virtual memory paging from operating systems.

> **DEFINITION**
> **PagedAttention** partitions each sequence's logical KV cache into fixed-size token blocks mapped through a per-request **block table** to non-contiguous physical blocks in a shared HBM pool, allocating blocks on demand and enabling copy-on-write prefix sharing across requests.

Trace how two concurrent requests, Request A and Request B, share memory under PagedAttention when both start with the same system prompt:

1. **Logical views:** Request A has logical `Block 0 (shared sys)` followed by `Block 1 (A unique)`. Request B has logical `Block 0 (shared sys)` followed by `Block 1 (B unique)`.
2. **Block tables:**
   - `Block Table A` maps logical `0 -> Phys #2 (rc=2)` and logical `1 -> Phys #5 (rc=1)`.
   - `Block Table B` maps logical `0 -> Phys #2 (rc=2)` and logical `1 -> Phys #0 (rc=1)`. Here `rc` is the reference count tracking how many requests point to that physical block.
3. **Physical HBM KV Pool (non-contiguous):** Physical slots `#0 (B1)`, `#2 (Shared)`, and `#5 (A1)` hold live KV blocks scattered freely across HBM, while `#1`, `#3`, and `#4` remain in the free pool. Because physical blocks need not be contiguous, external fragmentation drops to zero, and internal fragmentation is bounded strictly to at most one partially filled final block per sequence. Whenever a request must modify a shared block with `rc > 1`, **copy-on-write** allocates a fresh physical block for the writer so other requests remain untouched.

> **WORKED EXAMPLE**
> **Worked Example: Bounding Internal Fragmentation with PagedAttention**
>
> Suppose a server supports a maximum sequence length of $4096$ tokens, and a request finishes at $100$ tokens.
>
> 1. **Naive contiguous pre-allocation:** Reserves $4096$ token slots; $4096 - 100 = 3996$ slots ($97.6\%$) are wasted as internal fragmentation.
> 2. **PagedAttention with block size $16$ tokens:** Allocates $\lceil 100/16 \rceil = 7$ blocks ($112$ token slots total). Only $112 - 100 = 12$ slots ($10.7\%$) are unused—and on average across random lengths, internal fragmentation is under half a block ($< 8$ tokens per sequence, regardless of maximum context length).
>
> **Interpretation:** Paged allocation shrinks memory waste from thousands of tokens per request down to a tiny constant fraction of one block, freeing HBM to support dramatically larger decode batches.

## Prefix Caching: RadixAttention and Hierarchical Caching

PagedAttention makes sharing a prefix across *simultaneously active* requests effortless, but what if User B arrives five seconds *after* User A finished, or an agent starts Turn 5 reusing Turns 1–4 from two seconds ago? If a serving engine immediately frees KV blocks when a request completes, every subsequent turn must recompute prefill from scratch.

**RadixAttention** (SGLang) solves cross-request and multi-turn prefix reuse by keeping finished requests' KV blocks cached in a **radix tree** (a compressed prefix tree) indexed by token sequences:

> **DEFINITION**
> **RadixAttention** retains the KV cache blocks of finished requests inside a token-indexed radix tree with least-recently-used (LRU) eviction on leaf nodes, allowing any new request sharing a cached prefix to hit the tree and skip prefill for all matched tokens.

Picture a live radix tree in HBM during an agentic workload:

- At the root sits `System Prompt` (`KV blocks [0..63]`).
- Two branches descend from the root: on the left, `Agent Session 1 (Turn 1–4)` sits hot in HBM and splits further into two parallel child branches, `Tool Call A` and `Tool Call B`; on the right, `Chat User B (Turn 1)` sits hot in HBM.
- When HBM fills up, the manager applies **LRU eviction on leaves** so popular shared ancestors like the system prompt stay resident.
- Rather than discarding evicted cold KV blocks completely, **hierarchical caching** spills cold KV blocks from GPU HBM down to **host DRAM** (indicated by a dashed arrow from `Chat User B` to Host DRAM). If that session resumes later, loading its KV cache back from host DRAM is far cheaper than recomputing prefill across tens of thousands of tokens—extending effective cache capacity for long-running agents.

## Hybrid KV Cache Managers for 2026 Models

Now recall the systems warning from the end of Section 4: 2026 frontier models such as Kimi K3, Qwen3.8-Flash-Next, DeepSeek-V4, and Nemotron 3.5 are **hybrids**, combining full-attention layers, compressed sparse layers (CSA/HCA), and fixed-state recurrent layers (Mamba/DeltaNet). That heterogeneity breaks the classical PagedAttention and RadixAttention assumption that every layer stores uniform KV bytes per token.

To serve 2026 hybrid models without wasting memory, modern engines—specifically the **vLLM hybrid KV cache manager** and **SGLang's Unified Radix Cache** (combining Full and Mamba components)—manage a **Unified HBM Memory Pool Grouped by Layer Type** divided into three compartments with distinct prefix-hit rules:

1. **Full-Attention Layers:** Store full KV per token using standard radix/paged blocks; a prefix cache hit can occur at **any block boundary**.
2. **Compressed Layers (e.g., CSA / HCA):** Store compressed KV at strides such as $4:1$ or $128:1$; a prefix hit must align to the **compressed stride**.

3. **Recurrent-State Layers (Mamba / DeltaNet):** Store a fixed-size state vector/matrix that overwrites itself in place on every token.

**COMMON MISCONCEPTION**

You might think that if you cache a $2000$-token sequence in a hybrid Mamba–Transformer model, a later request sharing only the first $1000$ tokens can simply slice the cached state at token $1000$ and reuse it. That breaks for the recurrent layers: because a recurrent state overwrites itself in place as tokens $1001 \ldots 2000$ are processed, the intermediate state at token $1000$ no longer exists! Prefix reuse in recurrent-state layers can therefore hit **only at explicitly saved checkpoints**.

**KEY IDEA**

Dynamic serving removes idle hardware bubbles and HBM waste through iteration-level continuous batching, padding-free selective batching, roofline-synergistic chunked prefill, non-contiguous PagedAttention blocks, and RadixAttention prefix trees—extended in 2026 to hybrid KV managers that enforce per-layer-type prefix-hit rules including checkpointed recurrent states.

These mechanisms saturate a single serving instance under ragged, prefix-heavy traffic. Yet frontier models like GLM-5.3 ($704$ GB checkpoint) and Kimi K3 ($2.8$T parameters) cannot fit on one GPU, and we still have to solve Section 3's MoE batch collapse, eliminate CPU launch bottlenecks, enforce structured outputs, and route agentic traffic across an entire cluster. That brings us to our final section.

### ✎ Check Your Understanding

1. Yes or No: In mixed chunked prefill, does combining a prefill chunk with an active decode batch require reading the model's weights from HBM twice (once for prefill and once for decode)? Explain briefly.

   **Answer:** No. Selective batching packs the prefill-chunk tokens and decode tokens together into one forward pass, so a single HBM read of the model weights serves both phases simultaneously while utilizing idle tensor cores.

2. Fill in the blank: Because recurrent layers such as Mamba or DeltaNet overwrite their fixed-size state in place at every token, hybrid KV cache managers can achieve prefix hits on recurrent layers only at _____.

   **Answer:** Explicitly saved checkpoints.

# Section 8: Cluster-Scale Serving, Low-Latency Execution, and Agentic Traffic

At cluster scale, four distinct system bottlenecks emerge: interference between compute-bound prefills and memory-bound decodes, MoE per-expert batch collapse across hundreds of experts, host-side CPU scheduling overhead as Blackwell steps shrink to microseconds, and prefix-cache misses when routing long-context agent requests across replicas. Each bottleneck has a direct architectural solution rooted in the physics we established in Sections 2 and 3.

## Prefill / Decode Disaggregation

Chunked prefill mitigates prefill-decode interference on a single instance, but prefill and decode fundamentally want different hardware configurations, batch sizes, and parallelism layouts (as Section 3 showed: small $B$ for prefill to minimize TTFT, large $B$ plus wide expert parallelism for decode to maximize throughput and keep TPOT steady). **Prefill / Decode (P/D) disaggregation** follows directly from the roofline split by dedicating separate GPU pools to the two phases.

> **DEFINITION**
> **Prefill / Decode (P/D) disaggregation** separates prompt processing onto a compute-optimized **Prefill Pool** and token generation onto a memory-optimized **Decode Pool**, transferring the completed KV cache between pools over high-speed RDMA transports such as Mooncake or NIXL.

Trace the disaggregated cluster flow from left to right:

- **Prefill Pool (compute-bound, $T = S$):** Runs small batches for low TTFT, scales independently to match incoming prompt load, and experiences zero decode interference.
- **KV Transfer (RDMA):** Once prefill finishes, the pool ships the newly computed KV cache over RDMA (using Mooncake or NIXL transports) to an assigned decode instance.
- **Decode Pool (memory-bound, $T = 1$):** Runs large batches with wide expert parallelism for high throughput and steady TPOT, scales independently for active decode concurrency, and is never stalled by incoming long prompts.

The core tradeoff of P/D disaggregation is paying network transfer time for the KV cache. Why has P/D disaggregation become practical in production? Directly because of Sections 4 and 5: compact MLA and FP8 KV caches ($44.9\ \mathrm{KB/tok}$ in GLM-5.3 vs. $1.87\ \mathrm{MB/tok}$ in MHA BF16) shrunk the volume of bytes crossing RDMA by over $40\times$.

## Wide Expert Parallelism and Attention–FFN Disaggregation (AFD)

Recall Section 3's **MoE batch collapse**: on a single GPU running $B = 256$ decode tokens with $E = 256$ experts and top-$k = 8$, each expert GEMM sees only $Bk/E = 8$ tokens—deeply memory-bound. **Wide Expert Parallelism (wide EP)** fixes this by pooling decode batches across $M$ GPUs so every expert receives $M\times$ more tokens per step.

Trace the four-stage wide-EP decode pipeline across $M$ GPUs (`GPU 0` through `GPU M-1`):

1. **Data-Parallel Attention:** Each of the $M$ GPUs holds a distinct local decode batch ( $B_0, B_1, \ldots, B_{M-1}$) and runs KV-light MLA attention independently on its own requests. Keeping attention data-parallel avoids splitting MLA across GPUs with tensor parallelism, which would redundantly duplicate KV reads across ranks.
2. **All-to-All Dispatch (DeepEP):** Every GPU dispatches each token's hidden state across the cluster interconnect to whichever GPU hosts the $k$ experts selected by that token's router.
3. **Wide Expert Ranks:** Each GPU stores only a small disjoint slice of the $E$ experts (e.g., `GPU 0` holds `Experts 0..k` plus duplicates of heavily requested "hot" experts). Because tokens arrived from *all* $M$ GPUs, each expert now processes a large pooled batch, lifting expert GEMMs toward the compute roof.
4. **All-to-All Combine:** Expert outputs travel back across the interconnect to each token's originating attention rank.

In practice, token routing is never perfectly uniform—some "hot" experts attract far more tokens than average, creating stragglers that stall the cluster synchronization barrier. Production wide-EP stacks pair **DeepEP** (for fast all-to-all dispatch and combine) with **EPLB (Expert Parallelism Load Balancing)**, which dynamically replicates hot experts across ranks so stragglers do not stall the barrier.

> **WORKED EXAMPLE**
>
> **Worked Example: Lifting MoE Expert GEMMs with Wide Expert Parallelism**
>
> Return to GLM-5.3 ($E = 256$ experts, top-$k = 8$). Suppose each GPU holds a local data-parallel attention batch of $B_{\text{local}} = 256$ decode tokens, and we deploy wide EP across $M = 32$ GPUs.
>
> 1. **Without wide EP ($M = 1$):** Global decode batch is $256$, so each expert averages $(256 \times 8)/256 = 8$ tokens.
> 2. **With wide EP ($M = 32$):**
>    - Each GPU hosts $256/32 = 8$ unique experts (plus any EPLB hot-expert replicas).
>    - Pooled global decode batch across the $32$ data-parallel attention ranks: $B_{\text{global}} = 32 \times 256 = 8{,}192$ tokens.
>    - Average tokens per expert:
>
> $$\text{tokens / expert} = \frac{8{,}192 \times 8}{256} = 256 \text{ tokens per expert}$$
>
> **Interpretation:** Pooling across $32$ GPUs raises per-expert batch size by $32\times$ (from $8$ to $256$), pulling the MoE expert GEMMs out of the deeply memory-bound regime and near the compute-bound roofline—while keeping each GPU's local attention batch at a manageable $256$.

Can we disaggregate even further inside a decode layer by placing attention and FFN/MoE on completely separate GPUs? **Attention–FFN Disaggregation (AFD)** attempts exactly this: **Attention Ranks** hold only KV caches (zero expert weights), while **FFN / MoE Ranks** hold only MoE experts (zero KV cache footprint), allowing large pooled expert batches. To hide the per-layer network transfer latency between

the two sets of ranks, AFD uses **ping-pong microbatching**: while Attention Ranks run microbatch A and stream its hidden states to the FFN Ranks, the FFN Ranks simultaneously process microbatch B and stream microbatch B's FFN outputs back to the Attention Ranks.

However, the measured empirical reality on homogeneous GPUs is sobering: gains are modest—$0$–$17\%$ (FastAFD self-reported up to $1.17\times$ over the best vLLM wide-EP configuration on GB200 NVL72). Worse, AFD is **fragile**: because workloads fluctuate in context length and batch size, any mismatch in the provisioned ratio of attention GPUs to FFN GPUs leaves one side idle and actually *loses* throughput compared to co-located wide EP.

## Hiding CPU Overhead: Overlap Scheduling, CUDA Graphs, Sync-Free Runners, and Megakernels

As Blackwell GPUs execute small decode steps in microseconds, the bottleneck frequently shifts off the GPU altogether onto **host-side Python and CUDA scheduling overhead**. Look at a naive execution timeline: the CPU runs `CPU sched t` while the GPU sits completely idle, then launches `GPU batch t` one kernel at a time, waits for it to finish, runs `CPU sched t+1` while the GPU stalls again, and so on.

Modern engines eliminate these host-induced bubbles through four progressively deeper techniques:

1. **Overlap scheduling:** The CPU asynchronously prepares scheduling metadata and KV block tables for batch $t+1$ *while* the GPU is still executing batch $t$. On the parallel CPU/GPU timeline, `Prep batch t+1` executes directly above `GPU (t)`, `Prep batch t+2` above `GPU (t+1)`, leaving zero idle bubbles between GPU steps.
2. **CUDA graphs:** Instead of having the CPU submit dozens or hundreds of individual kernel launches per Transformer layer, the engine captures an entire decode step once into a single replayable graph. The CPU then triggers the entire step with **1 launch per step**, ensuring per-op launch latency never dominates small batches.
3. **Fused kernels:** Kernels such as FlashAttention, FlashDecoding, and fused paged-block reads combine multiple operations into one pass, eliminating intermediate HBM round-trips and extra launches.
4. **Sync-free model runners and persistent megakernels:**
   - **Sync-free model runners** remove per-step CPU–GPU synchronization barriers. Rather than copying sampled token IDs and metadata back to the CPU at the end of every step just to prepare the next step's inputs, the runner builds next-step inputs directly on the GPU. On one GB200, **vLLM Model Runner V2** self-reported $+56\%$ **output throughput** on Qwen3-0.6B by eliminating per-step host syncs.
   - **Persistent megakernels** fuse an entire decode step—or even a multi-layer loop—into a single long-lived GPU kernel, removing inter-kernel launch gaps and barrier overhead entirely. Shipped as opt-in low-latency paths in SGLang (low-latency mode) and vLLM (specialized decode), persistent megakernels deliver their largest gains at **batch** $1$ (single-user latency), whereas standard kernels still win at high batch sizes.

## Constrained Decoding and Batch-Invariant Serving

Production agent and RL workloads also impose two strict correctness constraints that must coexist with high-speed serving:

- **Constrained decoding (structured outputs):** Tool-calling agents and API pipelines often require outputs that strictly conform to a JSON schema or context-free grammar. At every decode step, the engine computes a token mask over the vocabulary setting the logits of all syntactically invalid tokens to $-\infty$ before sampling. **XGrammar** is the default backend across vLLM, SGLang, and TensorRT-LLM: it splits the vocabulary into precomputed context-independent tokens plus a fast runtime pushdown stack. The critical **systems constraint** is that mask computation must finish inside the overlapped CPU scheduler window so it introduces zero GPU stalls.
- **Batch-invariant and deterministic serving:** Recall from floating-point arithmetic that $(a + b) + c \neq a + (b + c)$ due to rounding. In standard serving kernels, the floating-point reduction order inside matrix multiplies and attention varies with batch size $B$, causing a model to emit slightly different outputs for the exact same prompt depending on concurrent server load. For reproducible RL rollouts and auditing, **DeepSeek DeepInfer** ships **batch-invariant inference** and deterministic sampling. Determinism carries a real throughput cost: SGLang self-reported roughly $34\%$ **overhead** with CUDA graphs, while research systems report single-digit overhead.

## Cache-Aware Routing and Real Agentic Benchmarks

Finally, step back to the cluster load balancer sitting in front of multiple serving replicas. Suppose an agent enters Turn $k$ with a massive conversation prefix between $88\text{k}$ and $675\text{k}$ tokens. If a standard random or load-only router sends Turn $k$ to `Replica 2` (which has a cold cache for this session), `Replica 2` must re-prefill the entire $88\text{k}$–$675\text{k}$ token history from scratch!

**Cache-aware routing** solves this by tracking live KV-cache residency across replicas and scoring every candidate instance using a combined objective:

$$\text{score}(\text{replica}) = \text{prefix hit} + \text{queue load}$$

A **KV-aware router** directs Turn $k$ to `Replica 1`, which already holds the session's prefix in its RadixAttention cache—skipping $99\%$ of prefill and delivering sub-second TTFT.

> **WORKED EXAMPLE**
>
> **Worked Example: Impact of Prefix-Cache Hit Rate on Agentic Traffic**
>
> Real agentic traces measured in the SemiAnalysis AgentX benchmark (Claude Code replay on DeepSeek-V4-Pro) show a striking asymmetry between input and output lengths:
>
> - **Input length:** median ($\text{p}50$) $88{,}000$ **tokens**, tail ($\text{p}99$) $675{,}000$ **tokens**.
> - **Output length:** median ($\text{p}50$) only $413$ **tokens**.
>
> Compare what happens on a median $88{,}000$-token agent turn when routed to a cold replica versus a KV-cached replica with a $99\%$ prefix hit:

1. **Cold replica ($0\%$ hit):** Must run prefill on all $88{,}000$ input tokens before generating $413$ output tokens. Prefill processes $88{,}000/413 \approx 213\times$ more tokens than decode!
2. **Cached replica ($99\%$ hit):** Only the $1\%$ new turn tokens ($880$ tokens) require prefill—a $100\times$ **reduction in prefill work**.

In the **llm-d benchmark** (self-reported), precise KV-event routing drops P90 TTFT to $0.54$ s, compared to $93$–$95$ s under load-only or random routing—over a $170\times$ latency improvement!

**Interpretation:** On real multi-turn agent workloads, where inputs dwarf outputs by more than $200\times$, **KV hit rate and prefix-aware routing matter more than peak decode tokens/s**.

**COMMON MISCONCEPTION**

You might think that optimizing an agentic inference cluster means focusing primarily on maximizing raw decode tokens/s per GPU. That breaks on real agent traffic (where median inputs are $88$k tokens and median outputs are only $413$ tokens): a single cache miss that forces re-prefilling $88$k tokens costs tens of seconds of TTFT, dwarfing even a $2\times$ difference in decode speed over $413$ tokens.

## Synthesis: The Full Anatomy of Fast LLM Inference

Across all eight sections, fast LLM inference emerges as a single co-designed stack spanning roofline physics, model architecture, compression, lossless acceleration, and cluster-scale serving systems:

| Layer | Core Bottleneck | 2026 Solution |
|---|---|---|
| **Roofline Physics** | Decode is memory-bound; MoE dilutes per-expert batch by $k/E$ | Batching, KV sharing, Blackwell NVFP4/MXFP8, wide EP |
| **Architecture** | KV capacity and $O(S)$ attention blow up at $1$M context | GQA, MLA ($N$ heads/latent), CLA, DSA/CSA/HCA, recurrent hybrids |
| **Compression** | Model and KV bytes limit batch size and latency | NVFP4 weights, FP8 KV (test FP4 KV on long context), pruning + KD |
| **Lossless Speedup** | Sequential $1$-token steps waste GPU FLOPs | Speculative decoding: MTP layers, draft trees, DFlash/DSpark, Spec V2 |
| **Serving Systems** | Ragged traffic, prefix recompute, host overhead | Continuous/chunked batching, Paged/Radix/Hybrid KV, P/D disaggregation, overlap + CUDA graphs, KV-aware routing |

**KEY IDEA**

Cluster-scale 2026 serving co-designs P/D disaggregation, wide expert parallelism (DeepEP + EPLB), host-overhead elimination (overlap scheduling, CUDA graphs, sync-free runners, persistent megakernels), and KV-aware prefix routing—where on real agent workloads ($88$k p50 input vs. $413$ p50 output), cache hit rate dominates end-to-end responsiveness.

### ✎ Check Your Understanding

1. What would happen to single-expert arithmetic intensity during MoE decode if you replaced wide expert parallelism across $32$ GPUs with $32$ independent data-replicated instances (each running $B_{\text{local}} = 128$ tokens across all $E = 256, k = 8$ experts)?

   **Answer:** Without wide EP, tokens are no longer pooled across the $32$ GPUs, so each expert's effective batch size drops by $32\times$—from $(32 \times 128 \times 8)/256 = 128$ tokens/expert down to $(128 \times 8)/256 = 4$ tokens/expert—pushing the expert GEMMs deeply into the memory-bound regime.

2. Why can a standard load-only router degrade P90 TTFT by more than $100\times$ on agentic workloads compared to a KV-aware router?

   **Answer:** Agentic turns carry huge repeated prefixes (e.g., p50 $88\text{k}$ tokens) with short outputs (p50 $413$ tokens); routing a turn to a slightly less loaded replica that lacks the cached prefix forces full prefill recomputation instead of skipping roughly $99\%$ of prefill via a RadixAttention cache hit.