25 Advanced Topics: Revisiting Sequence Models, Transformers, and Large Language Models

Chapter Abstract. Chapter 24 ended with an audio sequence model whose quality depends on codec tokens, long contexts, and latency. This chapter revisits recurrent neural networks, Transformers, and large language models after the book has introduced the basic versions of those ideas. The running example is a course learning assistant that must answer a learner question from long course notes while staying accurate, fast, and cheap enough to serve during office-hour peaks. The chapter explains why modern architectures often return to older principles: recurrent state for long memory, selective state-space models such as Mamba, scaled gated memory in xLSTM, efficient Transformer attention and KV caches, residual-stream design in DeepSeek-style Hyper-Connections and manifold-constrained Hyper-Connections, and sparse conditional computation in mixture-of-experts models. The goal is not to rank model names. The goal is to give beginners a precise applied vocabulary for deciding what to try first, what to measure, and how to report accuracy, training time, inference latency, memory, cost, and stability together. An architecture sales pitch becomes useful only after it names the measured bottleneck it is supposed to fix.

25.1 Why Revisit Architectures?

Chapter 9 introduced recurrent neural networks as models that read a sequence one step at a time. Chapter 10 introduced attention and the Transformer as a way for positions in a sequence to compare directly with one another. Chapter 11 then showed how decoder-only Transformers became the standard architecture behind many large language models. This chapter returns to those same ideas because modern applied systems make the old questions sharper: what should the model remember, how long is the input, how fast must the answer arrive, and how much memory and money can the system spend?

The running example is a small course learning assistant for an introductory applied deep learning course. A learner asks, “Using the course notes, explain why my recurrent model trains slowly on long reviews, compare it with a Transformer, and give one practice exercise.” The assistant must search or read long notes, keep the learner’s question in mind, generate a correct answer, and respond quickly enough that the learner does not abandon the page. This is not a frontier benchmark. It is exactly the kind of applied constraint that decides whether an architecture is useful.

Definition 25.1.1 (Applied architecture choice) *An applied architecture choice is a decision between model or serving designs for a fixed task, data source, evaluation rule, and deployment setting. A useful choice reports the baseline, measured quality, training or adaptation time, inference latency, memory use, cost, and known risks, then states which tradeoff the project accepts.*

This definition is intentionally practical. It does not turn the choice into a single universal score; it forces the project to name the tradeoff. If one assistant answers 83 of 100 validation questions correctly with median latency 0.8 seconds, and another answers 87 correctly with median latency 4.9 seconds, the second model is not automatically better. For a homework help page where a learner can wait, four extra correct answers may be worth the delay. For a live chat widget used by many learners at once, the first model may be the better system because it is fast enough to answer more learners within the same server budget.

Measure the tradeoff. Architecture choice is a budget decision: quality, latency, memory, cost, and risk move together.

Modern architectures revisit older principles because each principle solves a different bottleneck. Recurrence gives a compact state that can be updated as a sequence streams by. Attention gives direct comparisons between positions, which often improves accuracy but can cost memory on long context. Residual connections help deep networks train by preserving a path through many layers. Mixture-of-experts models add capacity by routing tokens to selected expert networks instead of using every parameter for every token. The rest of the chapter treats these ideas as engineering choices that must be measured against a baseline.

25.2 The Measurement Vocabulary for Advanced Architectures

Before comparing Mamba, xLSTM, efficient attention, Hyper-Connections, or mixture-of-experts models, a beginner needs a small set of measurements. These measurements are deliberately ordinary. They do not require advanced theory, but they prevent vague claims. A model comparison that says only "Model B is better" is incomplete. A useful comparison says what task was measured, what baseline was used, how quality changed, how long training took, how much latency the user saw, and what memory or cost was required. This is the guardrail against architecture tourism: a new block should answer a measured problem, not merely add a newer acronym to the diagram.

Name the unit. State the task, baseline, metric, latency, memory, and cost before claiming an upgrade.

Definition 25.2.1 (Serving token budget) *For a language-model service, separate the prompt length T_{prompt} from the generated length T_{gen}. The prompt length counts system instructions, user input, retrieved context, and other tokens processed before the first generated token. The generated length counts output tokens emitted during decode, and*

$$T_{\text{total}} = T_{\text{prompt}} + T_{\text{gen}}.$$

A report should state which length is being measured because prefill work, cache memory, and output cost depend on different parts of the interaction.

In the course assistant, if the prompt contains 800 tokens of learner text, system instructions, and retrieved course-note passages, and the answer generates 200 tokens, the serving system has handled a 1,000-token interaction. The exact token count matters because many costs grow with token length, but not all costs grow in the same phase.

Definition 25.2.2 (Interactive serving latency and throughput) *In this chapter, inference latency is end-to-end wall-clock time from request arrival to a usable answer; for streaming systems, a report may also include time to first token. Throughput is completed answers or generated tokens per second under a stated load, batching policy, and hardware or API setup.*

Latency and throughput are related but not identical. A server may generate many total tokens per second when it batches 64 requests together, yet a single learner request may still wait too long. For an interactive assistant, the median latency reports the typical experience, while a high percentile such as p95 latency reports the slow cases. If 100 requests have latencies and the 95th fastest request takes 7.2 seconds, then p95 latency is 7.2 seconds. That number matters because the slow tail is often what users notice.

Definition 25.2.3 (Serving baseline) *A serving baseline is a recorded reference system for the same task, data split, prompt or retrieval policy, decoding settings, hardware or API configuration, and evaluation rule. It reports quality, latency, memory or cost, and representative failures before a new architecture or serving method is compared against it.*

For the course assistant, a reasonable first baseline is not a large new model. It is a simple retrieval or small language-model setup whose quality and latency are recorded. Only after that baseline exists can a learner make a defensible claim such as, "Grouped-query attention reduced KV-cache memory in our serving configuration while preserving the same validation answer score," or, "The MoE model improved accuracy but made p95 latency worse on our hardware." The language of this chapter is built to make such claims possible.

Quantity	What it asks	Example report field
Quality Q	Did the answer solve the task?	Exact match, rubric score, citation recall
Training time	How long did fitting or adaptation take?	42 minutes on one GPU
Latency	How long did one user wait?	Median 0.8 s, p95 2.4 s
Memory	How much state had to be stored?	1.1 GB KV cache at $T = 8{,}192$
Cost	How much did usable output cost?	Dollars per 1,000 successful answers

25.3 RNN Revisited: Modern Long-Sequence Memory

The original appeal of a recurrent neural network was simple: a sequence can be processed one item at a time while the model carries a state. Elman's recurrent network made this idea concrete, and LSTMs later improved long-range memory through gates and a more stable cell state [50, 101]. The old RNN chapter emphasized that recurrence is not just a diagram with a loop. It is a rule for updating memory.

Definition 25.3.1 (Compact recurrent state) *A compact recurrent state is a fixed-width summary updated once per sequence position:*

$$h_t = F_\theta(h_{t-1}, x_t), \qquad \hat{y}_t = G_\theta(h_t),$$

where x_t is the input at position t, h_t is the state after reading position t, and θ denotes learned parameters. The state size does not grow with the prefix length, so the model trades long-context storage for a bottleneck: information not encoded in h_t cannot be recovered by later positions.

Compact state. Recurrence trades direct lookup for a memory vector that must carry what matters.

For the course assistant, h_t is a compressed memory of what has been read so far. If a policy sentence appears early in the notes and the learner asks about it later, the useful state should preserve the policy information. The strength of this pattern is that the work grows roughly linearly with sequence length: one update per token. If a model needs 1 unit of recurrent work per token, then 1,000 tokens require about 1,000 updates and 10,000 tokens require about 10,000 updates.

The weakness is that the state is also a bottleneck. A simple recurrent model must compress all relevant past information into h_t. During training, gradients must pass backward through many repeated state updates. Repeated multiplication can make gradients shrink or explode, which is why early work on long dependencies and later analyses of recurrent training difficulty remain important [209]. LSTMs and GRUs responded with gates: learned controls that decide how much information should be written, kept, or forgotten.

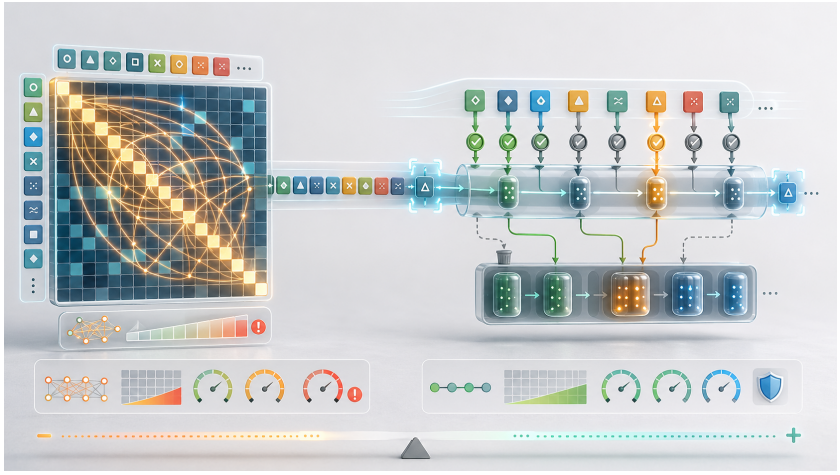

Figure 25.1: AI-generated conceptual illustration of the long-sequence tradeoff: direct attention compares many token pairs, while recurrent, gated, or state-space-style processing carries a compact memory forward through the sequence. The image is schematic, not a measured model trace.

The modern return to recurrence is therefore not a return to weak simple RNNs. It is a return to a practical question: can a model process very long sequences with a compact memory while still learning content-sensitive behavior? That question leads naturally to state-space models such as S4 and Mamba [82, 81], and to scaled gated-memory designs such as xLSTM [8]. Both families keep the applied promise of linear or near-linear sequence processing, but they try to avoid the old failure mode of using a state that is too rigid or too hard to train.

The applied lesson is conservative. If the task is short and ordinary Transformer inference is fast enough, recurrence may not be necessary. If the task is streaming, very long, or dominated by memory cost, recurrent or state-space alternatives deserve a measured comparison. The comparison must use the same data, the same quality metric, and a timing protocol that reports both training and inference.

25.4 Mamba and Selective State-Space Models

State-space models come from the idea that a system can be described by a state that changes over time. In deep learning, a state-space sequence layer turns a token sequence into output vectors by updating an internal state. S4 showed that structured state-space models could model long sequences efficiently, and Mamba made a particularly influential change: the state update can be selective, meaning it can depend on the current input token [82, 81].

Definition 25.4.1 (Discrete state-space sequence layer) *A simplified discrete state-space layer can be written as*

$$s_t = A_t s_{t-1} + B_t x_t, \qquad y_t = C_t s_t,$$

where s_t is the state at position t, x_t is the input vector, y_t is the output vector, and A_t, B_t, and C_t control how the state is updated and read.

If A_t, B_t, and C_t are fixed for all tokens, the layer treats every token with the same update rule. Selective state-space models make these quantities depend on the input. In plain language, the model can treat "deadline," "optional," and "however" differently. For the course assistant, this matters because a long note may contain filler, examples, and policy statements. The model should preserve important assignment rules and let irrelevant wording fade.

Selective scan. The model still scans left to right, but the update rule can change with the token.

A more faithful selective sketch writes those controls as functions of the current token:

$$\Delta_t = \text{softplus}(W_\Delta x_t), \qquad B_t = W_B x_t, \qquad C_t = W_C x_t.$$

Starting from continuous-time parameters, a simple discretization view is

$$\bar{A}_t = \exp(\Delta_t A), \qquad \bar{B}_t = (\bar{A}_t - I)A^{-1}B_t, \qquad s_t = \bar{A}_t s_{t-1} + \bar{B}_t x_t.$$

The selective scan is the efficient algorithmic organization of this recurrence across many tokens; the model does not need to form a full $T \times T$ attention matrix to carry state forward.

Figure 25.2 shows the structural difference that makes this tradeoff easy to remember.

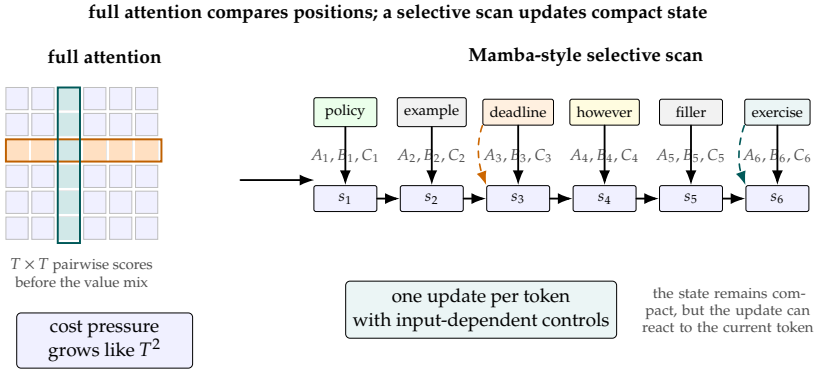

Figure 25.2: Full attention forms a score matrix over pairs of positions. A selective state-space layer scans left to right with compact state, while the update controls can change from token to token.

The numerical attraction is easy to see. Attention over T positions creates roughly T^2 pairwise comparisons. A state update that performs a fixed amount of work at each position scales more like T. For $T = 1{,}000$, the pairwise count is 1,000,000. For $T = 10{,}000$, it is 100,000,000. A linear scan over the same lengths grows from 1,000 to 10,000 updates. These counts do not by themselves predict wall-clock time, because hardware kernels matter, but they explain why long-sequence architecture research cares about subquadratic sequence processing.

Mamba is also a reminder that an algorithmic idea is not enough. The Mamba paper emphasizes a hardware-aware parallel algorithm because a method that is linear on paper can still be slow if it moves data inefficiently on the GPU [81]. For a beginner, the important point is not to memorize the kernel details. The important point is to measure actual training time and inference time rather than assuming that a better asymptotic formula always wins on the available machine.

For the course assistant, a Mamba-style model is a candidate when long context or streaming input is the bottleneck. It is less obviously the first choice if the main problem is exact retrieval of a specific passage, where a retrieval system from Chapter 12 may be simpler and easier to audit. A careful experiment would compare a small Transformer baseline, a retrieval-augmented baseline, and a state-space model on the same questions, then report accuracy, latency, and memory together.

That comparison is the difference between a measured bottleneck and an architecture sales pitch. If the assistant is wrong because the relevant note was never retrieved, a faster sequence layer is solving the wrong problem.

25.5 xLSTM and Scaled Gated Memory

The LSTM was one of the most important practical sequence models because it introduced an explicit memory cell and gates that control writing, forgetting, and reading [101]. xLSTM asks a modern question about that old idea: how far can LSTM-style memory go if it is redesigned with current scaling practices, normalization, and residual blocks? The xLSTM paper introduces exponential gating and modified memory structures, including scalar memory and matrix memory variants, to make LSTM-like models more competitive at larger language-model scales [8].

Gates are controls. Forget and input gates decide preservation and overwrite, not just notation.

The word *gate* should be understood literally. A gate is a learned value that controls how much information passes. If a gate value is near 0, the model mostly blocks that information. If it is near 1, the model mostly allows it. In a course note, a gate might learn to preserve "late submissions lose 10 percent per day" because it is policy information, while allowing a less important transition sentence to fade.

Definition 25.5.1 (Modern gated-memory update) *A modern gated-memory block preserves the LSTM idea of an additive memory path, which can be sketched as*

$$c_t = f_t \odot c_{t-1} + i_t \odot u_t,$$

where c_t is the memory, u_t is candidate new information, f_t is a forget gate, i_t is an input gate, and $\odot$ means elementwise multiplication. The modern design question is not merely whether gates exist, but how the gate parametrization, normalization, memory shape, and residual-block placement let the preserve/write mechanism remain stable at larger sequence-model scale.

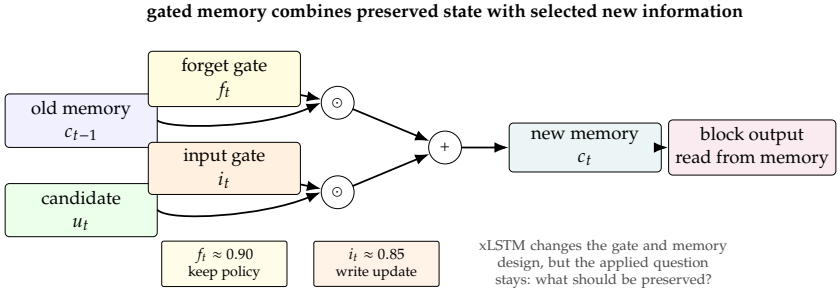

Figure 25.3: A gated memory update has one path for preserving old state and one path for writing candidate information. xLSTM modernizes this recurrent idea so deeper language-model blocks can use scaled gated memory.

This equation says that the new memory has two parts. The first part keeps some of the old memory c_{t-1}. The second part writes some new information u_t. If $f_t = 0.90$ and $i_t = 0.20$ for one memory coordinate, the model mostly preserves the old value and writes a smaller amount of new information. If $f_t = 0.10$ and $i_t = 0.85$, it mostly replaces that coordinate. The xLSTM family changes the details of gating and memory so that this old idea can be used in deeper, wider language-model blocks [8].

xLSTM-style exponential gating can be summarized by gates that are positive before normalization:

$$\tilde{i}_t = \exp(a_t), \qquad \tilde{f}_t = \exp(b_t),$$

followed by a normalization step that keeps the memory scale controlled. A matrix-memory variant replaces the scalar or vector memory cell with an outer product update,

$$M_t = f_t \odot M_{t-1} + i_t \odot (v_t k_t^\top), \qquad o_t = M_t q_t,$$

where k_t, v_t, and q_t play key, value, and query-like roles. The equations are more advanced than the chapter needs for implementation, but they make clear that xLSTM is not only the old LSTM cell with a new name.

The distinction between Mamba-style state spaces and xLSTM-style gated memory is useful but should not be exaggerated for beginners. Both are attempts to build sequence memory without ordinary full attention at every position. Mamba emphasizes selective state-space dynamics and hardware-aware scans. xLSTM emphasizes scaled recurrent memory with modern gates and residual block design. The applied question is the same in both cases: under the same training data and serving budget, which model gives acceptable quality at the lowest latency and memory cost?

For the course assistant, xLSTM would be a candidate if the main workload is long sequential text or streaming lecture transcripts. A fair report would not say, "xLSTM is better than Transformers." It would say something like, "On this long-transcript classification task, the xLSTM candidate reached 84% validation accuracy with 1.3 seconds median latency, while the small Transformer reached 86% with 2.7 seconds median latency." That sentence names the condition under which the tradeoff exists.

25.6 Transformer Revisited: Attention, Context, and KV Cache

The Transformer became central because attention gives each position a direct way to compare with other positions [292]. In language, this is powerful. The word "it" can attend to the earlier noun it refers to. A learner question can attend to the relevant sentence in the course notes. The same direct comparison also creates the cost that motivates many modern Transformer variants.

Definition 25.6.1 (Full attention score matrix) *For a sequence of T token representations, a full self-attention head forms query and key matrices $Q, K \in \mathbb{R}^{T\times d}$ and the score matrix*

$$S = QK^\top \in \mathbb{R}^{T\times T}.$$

Every output position can score every input position before masking and softmax, so the number of pairwise scores per head scales as T^2.

The matrix $S = QK^\top$ contains pairwise scores between positions. If the sequence has T tokens, ordinary full attention forms roughly T^2 such scores in each attention head. For $T = 100$, that is 10,000 scores. For $T = 1{,}000$, it is 1,000,000. For $T = 10{,}000$, it is 100,000,000. The numbers grow quickly, which is why long context is not just a matter of allowing a larger input box.

Long context has phases. Prefill pays attention work; decode pays cache memory and bandwidth.

During autoregressive generation, another memory issue appears. A decoder-only Transformer does not recompute every previous key and value from scratch at each generated token. It stores them in a key-value cache, usually called a KV cache.

Definition 25.6.2 (KV cache estimate) *A simplified estimate for KV-cache memory is*

$$M_{\text{KV}} \approx 2 \cdot L \cdot T \cdot H_{\text{kv}} \cdot d_h \cdot b,$$

where L is the number of layers, T is the cached sequence length, H_{kv} is the number of key-value heads, d_h is the head dimension, and b is bytes per stored number. The factor 2 appears because both keys and values are stored.

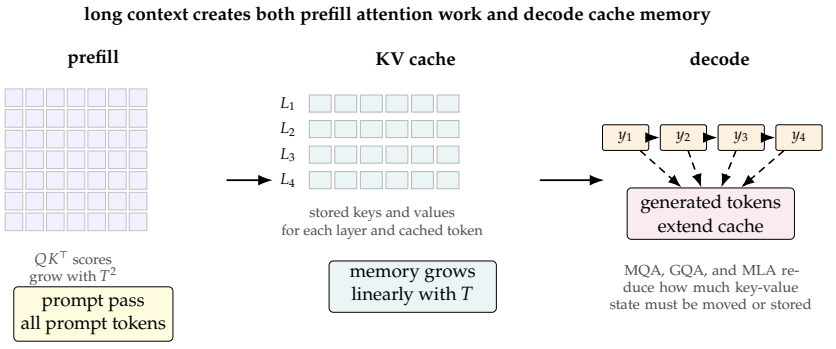

Figure 25.4: The prefill phase pays for attention over the prompt, while decoding reuses stored keys and values. Long context therefore has two practical bottlenecks: attention work during prefill and KV-cache memory during decode.

Consider a toy model with 12 layers, $T = 4{,}096$ cached tokens, $H_{\text{kv}} = 8$, head dimension $d_h = 64$, and $b = 2$ bytes for a half-precision number. The estimate is

$$2 \cdot 12 \cdot 4096 \cdot 8 \cdot 64 \cdot 2 = 100{,}663{,}296$$

bytes, or about 96 MiB. Doubling the context length to 8,192 doubles this cache to about 192 MiB for one sequence. Larger models and larger batches increase the pressure.

Different efficient-attention ideas attack different bottlenecks. Multi-query attention shares keys and values across query heads to reduce decoding memory bandwidth [250]. Grouped-query attention is a middle ground

between separate key-value heads and one shared key-value head [1]. Multi-head latent attention, used in DeepSeek-V3, compresses key-value information into a latent representation to reduce cache cost [40]. FlashAttention keeps exact attention but reduces memory traffic between GPU memory levels through an IO-aware algorithm [39]. Sparse or sliding attention reduces which positions are compared. Rotary position embeddings and context-extension methods address how position information behaves when context grows [266].

For the course assistant, this section gives a diagnostic rule. If long prompts make the first pass over notes slow, inspect prefill cost and attention length. If generating the answer is slow or memory-heavy, inspect the KV cache and decode path. An efficient attention variant should be chosen because it attacks the measured bottleneck, not because it is a fashionable acronym.

25.7 Transformer Revisited Again: Residual Streams and DeepSeek HC/mHC

Attention is not the only part of a Transformer that matters. Very deep models also depend on how information moves through layers. Residual connections, popularized in ResNets and then used throughout modern deep learning, give a network an identity-like path through depth [91, 92]. This is one reason deep networks can be trained at all.

Earlier ResNet and Transformer chapters used the standard residual form

$$z_{\ell+1} = z_\ell + F_\ell(z_\ell),$$

where z_ℓ is the representation entering layer ℓ, and F_ℓ is the learned transformation in that layer. This section uses that familiar equation only as the starting point for deeper residual-stream design.

The term z_ℓ is the important part. If the learned transformation F_ℓ is small at the beginning of training, information can still pass forward through the identity path. For the course assistant, this means that a representation of the learner's question can survive many layers while the model gradually refines it. Without stable paths through depth, training can become fragile: activations or gradients may shrink, explode, or become poorly conditioned.

Identity path. A residual stream is useful because unchanged information can survive many layers.

Hyper-Connections, abbreviated HC, generalize the residual idea by carrying multiple residual streams and allowing them to mix. This can increase the expressive pathways through the network, but unconstrained mixing can also damage the identity-like behavior that made residual connections stable [319]. DeepSeek's manifold-constrained Hyper-Connections, or mHC, address this problem by constraining the residual mixing space so that richer connections remain more stable at scale [305]. The phrase "manifold constraint" is advanced, but the beginner-level idea is simple: do not let the model learn arbitrary residual mixing if arbitrary mixing destroys stable signal flow.

In practical terms, this is not a decorative wiring choice. It is a claim that the model needs more paths through depth without losing the stable path that made deep training possible.

If $S_\ell \in \mathbb{R}^{K \times d}$ stores K residual streams, an unconstrained stream-mixing step might look like

$$S_{\ell+1} = M_\ell S_\ell + F_\ell(S_\ell),$$

where $M_\ell \in \mathbb{R}^{K \times K}$ mixes streams. The stability concern is that an arbitrary M_ℓ can amplify or collapse directions through depth. A manifold-constrained version restricts M_ℓ, for example by keeping it near an orthogonal or norm-preserving family with

$$M_\ell^\top M_\ell \approx I.$$

That constraint is the mathematical version of the prose rule: richer residual routes should not destroy stable signal flow.

Definition 25.7.1 (Residual-stream stability) *In this chapter, residual-stream stability means that representations and gradients can pass through many layers without systematic explosion or collapse. A practical sign of instability is a training run with repeated loss spikes, numerical overflow, or failure to make progress under settings where a comparable residual model trains normally.*

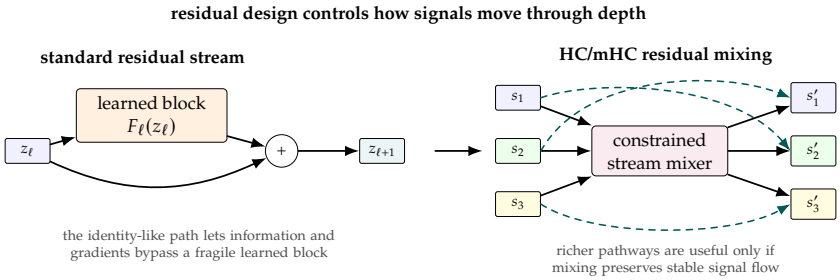

Figure 25.5: A standard residual block preserves one identity-like stream. HC and mHC-style designs carry multiple streams and mix them, so the architecture must control the mixing enough to keep deep training stable.

This topic belongs in a Transformer revisit because it changes how the Transformer block is wired, not merely how attention scores are computed. It is also a good example of applied humility. A method may improve large-scale training in a technical report, but a beginner should not add it casually to a small class project. The right takeaway is to record loss curves, gradient norms when available, optimizer settings, depth, precision, and failure cases. If a deeper model trains unstably while a shallower residual baseline trains cleanly, residual-path design is part of the evidence.

For the course assistant, HC or mHC is not the first knob to turn when answers are wrong. Data quality, retrieval, prompt construction, and ordinary model choice should be checked first. HC/mHC becomes relevant when the project is studying architecture at scale or trying to train very deep models where stability itself is the bottleneck.

25.8 LLM Revisited: Mixture of Experts and Conditional Computation

A dense neural network uses the same learned parameters for every input token. That is simple, but it ties model capacity and computation closely together: more parameters usually mean more work per token. Mixture-of-experts models, or MoE models, separate these quantities by using conditional computation. A model may contain many experts, but only a subset is active for a given token. Sparsely gated MoE layers, GShard, Switch Transformer, DeepSeekMoE, and DeepSeek-V3 are important examples of this direction [251, 158, 59, 37, 40].



Figure 25.6: AI-generated conceptual illustration of sparse expert routing. Many expert modules exist, but a token activates only selected experts; imbalance between busy and idle experts is part of the deployment problem. The image is schematic, not an exact implementation diagram.

Definition 25.8.1 (Dense and sparse active parameters) *A dense model uses the same parameter set for each token. A sparse MoE model has total parameters P_{total}, but uses only $P_{\text{active}}(x_t)$ parameters for token x_t. The sparse case is interesting when*

$$P_{\text{active}}(x_t) \ll P_{\text{total}}$$

for many tokens.

Suppose a dense toy model has 80 million parameters and uses all of them for every token. Now suppose an MoE toy model has 240 million total parameters, but its router selects experts so that only 60 million parameters are active per token. The MoE model has three times as many total parameters, but fewer active parameters per token than the dense model. This is the basic capacity-computation tradeoff.

Total vs active. MoE capacity is total expert parameters; per-token compute is selected experts.

Definition 25.8.2 (MoE routing) *Let $r_\theta(x_t)$ be router scores for token x_t. A top-k MoE layer selects*

$$S_t = \text{TopK}(r_\theta(x_t), k)$$

and combines selected expert outputs as

$$y_t = \sum_{e \in S_t} \alpha_{t,e} E_e(x_t),$$

where E_e is expert e, and $\alpha_{t,e}$ is the weight assigned to that expert for the token.

In a 4-expert toy layer with top-2 routing, each token calls two experts. For a 6-token prompt, the layer makes $6 \times 2 = 12$ expert calls. If the router sends all 6 tokens to experts 1 and 2, experts 3 and 4 do no work and the system is poorly balanced. If tokens are spread more evenly, the hardware can be used more effectively. This is why MoE papers discuss load balancing and expert specialization, not just parameter count [37].

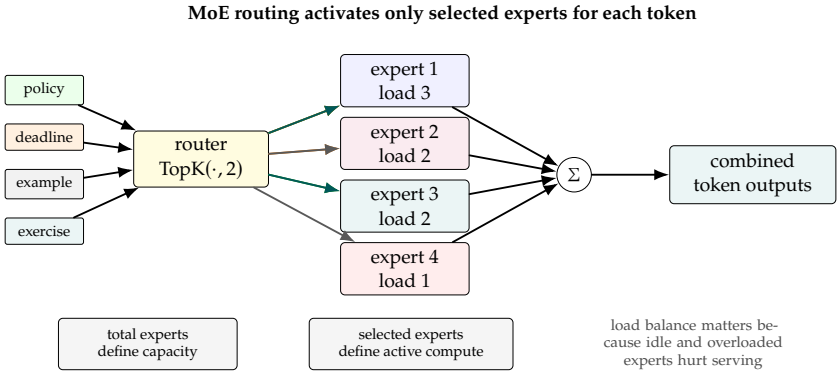

Figure 25.7: A top-k MoE layer routes each token to a small subset of experts and combines their outputs. The report must distinguish total capacity, active compute, and router balance.

MoE can be attractive for an LLM because it increases total capacity without paying dense compute for every token. It can also be awkward in deployment. Experts may live on different devices. Routing can create communication. Some experts may become overloaded while others are idle. The model may have low FLOPs per token but still suffer latency from memory movement or distributed coordination.

For the course assistant, an MoE model is worth considering when the baseline is capacity-limited and a sparse model is available through reliable tooling. The report should distinguish total parameters from active parameters, and it should include latency. A model with impressive total capacity is not a useful course assistant if p95 latency is unacceptable during the busiest hour.

25.9 Inference Revisited: Making Advanced Models Usable

Architecture is only one part of applied deep learning. A model that is accurate in a notebook can be unusable as a service if it is too slow, too memory-hungry, or too expensive. This is especially visible for language models, where serving a request has two different phases.

Definition 25.9.1 (Prefill and decode) *Prefill is the prompt-processing phase that builds activations and initial KV-cache entries before the first generated token. Decode is the repeated generation phase after the prompt has been processed; each step uses the cache, emits or selects new token information, and appends new cache entries. Reports should separate these phases because prompt length, generated length, cache bandwidth, and batching affect them differently.*

For the course assistant, prefill processes the learner's question plus course notes. Decode produces the answer. If the prompt contains 6,000 tokens and the answer contains 200 tokens, the system may spend substantial time on both phases. Efficient attention can help prefill. KV-cache design, batching, and decoding algorithms often matter during decode.

Separate phases. Optimizing prefill does not automatically speed decode, and vice versa.

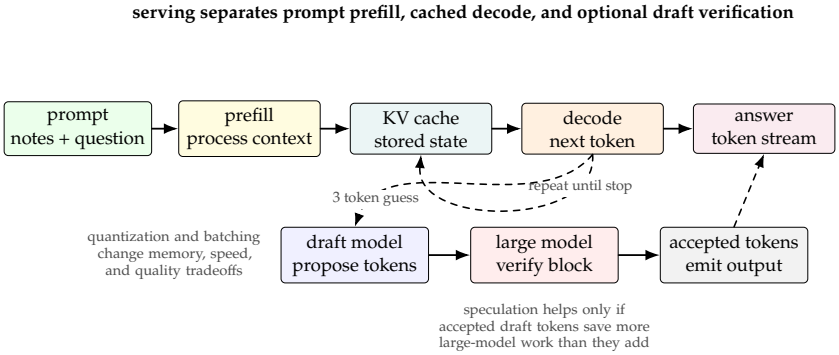

Figure 25.8: A serving report should separate prefill, decode, and optional speculative decoding. Each phase has different bottlenecks, so latency, throughput, cache size, precision, and quality must be reported together.

Quantization is another serving tool. It stores or computes some values in lower precision, such as 8-bit or 4-bit representations, to reduce memory and increase speed. The applied risk is that lower precision can hurt quality or numerical stability. A responsible report therefore gives both the latency change and the quality change. "The 4-bit model was faster" is incomplete. "The 4-bit model reduced median latency from 2.6 seconds to 1.5 seconds while lowering validation answer accuracy from 86% to 84%" is an applied statement.

Speculative decoding attacks a different bottleneck: ordinary autoregressive decoding generates one token at a time with the large model. In speculative decoding, a smaller draft model proposes several tokens, and the larger model checks them in a way that can preserve the target distribution [159]. Speculative sampling develops a related idea for large language model decoding [24]. DeepSeek-V3 also uses a multi-token prediction objective as part of its training and inference story [40].

A toy calculation shows the appeal. If ordinary decoding needs one large-model call per output token, a 24-token answer needs 24 large-model calls. If speculative decoding accepts an average of 3 draft tokens per large-model verification step, the same answer needs about $24/3 = 8$ large-model verification calls. The actual speedup will be smaller because the draft model and verification have costs, but the calculation explains why the method can reduce latency.

The practical serving report for an advanced model should include median latency, p95 latency, throughput, batch size, context length, output length, precision, hardware or API backend, and quality. Without those details, the claim cannot be reproduced. With those details, even a small classroom experiment can teach the central applied lesson: a deployment technique is successful only when it improves the whole measured workflow.

25.10 Worked Architecture Decision Audit

The chapter's running assistant is now ready for a small decision audit. The numbers below are deliberately modest: they are the kind of measurements a team could collect before changing architecture families. The task is fixed: answer 100 held-out course-note questions with citations, using the same retrieval corpus and grading rubric. The baseline is a compact decoder-only model with a standard Transformer cache. The proposed change is not allowed to win by changing the prompt, retriever, or validation set.

Run	Quality	Median latency	p95 latency	What the row proves
Baseline Transformer	83/100 correct	0.8 s	2.4 s	The current system is fast enough but misses some long-context evidence.
Long-context attention variant	87/100 correct	4.9 s	11.8 s	Quality improves, but interactive latency may be unacceptable.
Speculative decoding variant	83/100 correct	0.55 s	1.5 s	Serving improves only if quality stays fixed under the same rubric.

This table is not a leaderboard. It is a decision record. If the product is an offline grading assistant, the long-context variant may be worth testing next because four extra correct answers matter more than a few seconds of latency. If the product is a live help widget, the speculative decoding variant may be more valuable because it improves the learner's wait time

without changing the answer score. A mixture-of-experts model, Mamba-style block, or xLSTM variant would enter this table only after naming the bottleneck it is meant to fix.

The audit should also name the claim that remains unsupported. The table above does not prove that long-context attention is generally better, that speculative decoding is always safe, or that recurrent state would fail. It proves a narrower and more useful point: for this assistant, under this validation rule, the next experiment should be chosen from the measured bottleneck. That is the applied habit advanced architecture work needs.

Pause and decide

The long-context row improves quality but makes p95 latency almost five times worse than the baseline. Write the one sentence that would justify keeping it, and the one sentence that would justify dropping it.

25.11 Summary

This chapter revisited architecture from an applied point of view. Recurrent models, Transformers, residual connections, and mixture-of-experts models are not separate fashion cycles. They answer recurring questions: how should information be remembered, how should positions interact, how should signals pass through depth, and how much of the model should be used for one token?

Modern recurrent and state-space models return to the value of compact sequence memory. Mamba uses selective state-space updates so the model can decide what to keep or forget based on the current token. xLSTM revisits gated memory with modern scaling and stabilization choices. These approaches matter when long sequences, streaming inputs, or memory cost are central bottlenecks.

Transformers remain powerful because attention lets positions compare directly with other positions. That same direct comparison creates cost. Pairwise attention grows rapidly with sequence length, and autoregressive decoding stores a KV cache. MQA, GQA, MLA, FlashAttention, sparse attention, RoPE-style position handling, and context-extension methods respond to specific bottlenecks; they are not interchangeable upgrades.

Residual-stream design treats training stability as part of architecture. Deep residual paths help information and gradients move through depth. Hyper-Connections and manifold-constrained Hyper-Connections revisit that path by allowing richer residual streams while trying to preserve stable identity-like behavior. This matters most when depth and large-scale training stability are the problem.

Mixture-of-experts models revisit conditional computation. They can have many total parameters while activating only a subset for each token. The applied report must therefore distinguish total parameters, active parameters,

routing balance, latency, memory, and communication cost. The practical conclusion is the same across the chapter: choose the next architecture experiment from the measured bottleneck, compare against a baseline, and report quality together with speed, memory, cost, and limitations.

Mini audit. Before trying an advanced block, write one line each for the baseline failure, bottleneck category, proposed method, fixed comparison controls, keep-or-drop evidence, and claim the experiment still cannot support. Then ask what is slow, unstable, too expensive, or too inaccurate, and choose the smallest experiment that could change that evidence. Chapter 26 uses that habit to close the book with a curated reading map for what to study, build, and measure after these chapters.

25.12 Exercises

Exercise 1. A course assistant answers 84 of 100 validation questions correctly with median latency 0.9 seconds. A larger model answers 88 correctly with median latency 4.2 seconds. State one application where the larger model is defensible and one where it is not.

Exercise 2. Explain why a model with higher total throughput can still feel slow to a single interactive user.

Exercise 3. A prompt has 4,000 tokens and the answer has 200 generated tokens. Which serving phase is dominated by the prompt, and which phase repeats once per generated token?

Exercise 4. In one sentence, explain why KV-cache memory matters more during autoregressive decoding than during a one-token classification problem.

Exercise 5. Give one task where recurrent or state-space memory may be more appealing than full pairwise attention, and name the measurement that would support the choice.

Exercise 6. A speculative decoding setup accepts an average of two draft tokens per large-model verification step. For a 60-token answer, estimate the number of large-model verification steps and state why the real speedup may be smaller.

Exercise 7. A mixture-of-experts model has 64 billion total parameters and 8 billion active parameters per token. Which number matters for model storage, and which number matters more for per-token compute?

Exercise 8. Why should a report on grouped-query attention include quality as well as KV-cache memory?

Exercise 9. Name two fixed controls that must stay the same when comparing a baseline Transformer with a Mamba-style model on the course-assistant task.

Exercise 10. Write a keep-or-drop rule for an advanced architecture experiment that includes quality, latency, memory, and one inspected failure case.

25.13 Homework: Audit an Advanced Architecture Claim

Choose one advanced architecture or serving claim from this chapter: long-context attention, grouped-query attention, Mamba-style state-space memory, xLSTM, Hyper-Connections, mixture-of-experts routing, quantization, or speculative decoding. The goal is not to reproduce a frontier paper. The goal is to turn a claim into a small applied decision record.

Required Work

Define a task with a baseline and a fixed validation set. A suitable classroom task is the course-assistant example: answer a small set of course-note questions with a scoring guide for correctness, citation support, concision, and safety. If running models is not feasible, perform a paper-and-system audit using reported numbers from one paper or model card, but keep the same table structure and clearly mark which values were measured locally and which were reported by the source.
The report must include:

- the baseline method and the proposed advanced method;
- the bottleneck the proposed method is supposed to fix;
- the fixed controls, including data, prompts, scoring rule, context length, output length, hardware or API backend, and batch policy;
- quality, median latency or runtime, p95 latency when available, memory or model-size evidence, and cost when available;
- one inspected success and one inspected failure;
- a keep, drop, or defer recommendation.

Required Decision Table

Run	Quality	Latency or runtime	Memory or cost	Decision evidence
Baseline	Fixed metric and split	Same clock definition	Same meter or source	What fails now
Advanced method	Same metric and split	Same clock definition	Same meter or source	What changed and why

Recommendation

End with one paragraph that a project lead could act on. A strong conclusion does not say merely that the method is newer. It says whether the method should be kept, dropped, or deferred for this task, and it names the evidence that would change that decision.