CHAPTER 13 · PART IV — ADVANCED INFERENCE

# Speculative Decoding & Mixture of Experts

*Breaking the Sequential Bottleneck — Draft-Verify Loops, Expert Routing, and Infrastructure at Scale*

📖 ~55 pages 🔗 Prereqs: Ch 6 (Inference), Ch 11 (KV Cache), Ch 4 (Memory)

➡ Next: Ch 14 (Networking & Collective Comm.)

---

> *"The two most important techniques in modern LLM inference are speculative decoding and Mixture of Experts — one exploits the asymmetry between verification and generation; the other exploits the asymmetry between model knowledge and active computation. Both break the fundamental cost model your intuition was built on."*
>
> — Production Inference Engineering Principle

Standard autoregressive decoding has a deep structural flaw: every token requires a full forward pass through the entire model, and these passes are *sequential* — token N cannot start until token N−1 is complete. At batch size 1, a 70B model on H100 runs at approximately 0.6% GPU compute utilization. The GPU is spending 99.4% of its time waiting for memory. This chapter covers the two most powerful techniques for attacking this problem from opposite directions.

**Speculative decoding** exploits the asymmetry between *generation* and *verification*. A small draft model proposes K tokens cheaply; the large target model verifies all K in a single parallel forward pass. If the draft is right often enough, you get K tokens per iteration instead of 1 — at a fraction of the cost increase. **Mixture of Experts (MoE)** restructures the model itself: instead of one dense FFN that all tokens pass through, MoE has N specialized experts, routing each token to only K of them. DeepSeek-V3 has 671B total parameters but processes each token through only ~37B — achieving frontier quality at LLaMA-3 70B active compute cost. Together, these techniques represent the production frontier for inference efficiency in 2024–2026.

### LEARNING OBJECTIVES

**🟢 FOUNDATIONAL**

- Explain the draft-verify loop and why output quality is preserved
- Describe how MoE replaces a dense FFN and why it reduces compute
- Understand why expert load balance matters for throughput

**🟡 INTERMEDIATE**

- Derive the speculative decoding speedup formula and calculate for given α, K, c
- Calculate All-to-All communication volume for a given MoE EP configuration
- Profile acceptance rate and expert balance using production metrics

**🟣 PRINCIPAL LEVEL**

- Design optimal spec decode configuration for a production serving system
- Architect expert placement strategy minimizing cross-node All-to-All
- Quantify when disaggregated prefill-decode justifies infrastructure complexity

# 13.1 Speculative Decoding — Exploiting Verification Asymmetry

## 13.1.1 Why LLM Decode Is Memory-Bound

Before understanding why speculative decoding works, you need to understand exactly *why* standard decoding is so inefficient. At batch size 1, each decode step requires loading the full model weights from HBM — once, to process exactly one token.

```
DECODE MEMORY-BOUND ANALYSIS (LLaMA-3 70B, batch=1, BF16):

Model memory footprint:
  70B parameters × 2 bytes/param (BF16) = 140 GB in HBM

Per-token computation:
  FLOPs per token = 2 × N_params = 2 × 70×10⁹ = 140 GFLOPs
  Memory read required = 140 GB (entire model must be streamed)

H100 SXM5 specs [SHIPPED]:
  Peak BF16 FLOPS:      989 TFLOPS
  HBM3 bandwidth:       3.35 TB/s

Arithmetic Intensity (AI):
  AI = FLOPs / Memory_bytes = 140 GFLOPs / 140 GB = 1.0 FLOP/byte

H100 ridge point: ~295 FLOP/byte (where compute = bandwidth limited)
Actual AI: 1.0 FLOP/byte  ←  295× BELOW ridge point

Time analysis:
  Time (memory-bound):    140 GB / 3.35 TB/s = 41.8 ms/token
  Time (compute-bound):   140 GFLOPs / 989 TFLOPS = 0.14 ms/token
  Actual bottleneck: MEMORY — by 295× margin

GPU compute utilization: 0.14ms / 41.8ms = 0.34% ← the GPU is nearly idle!
```

The conclusion is stark: at batch size 1, the H100 is 99.7% idle waiting for memory. Speculative decoding attacks this by ensuring that each time we load the full 140 GB model from memory, we extract more than one token of value from that load.

### 13.1.2 The Draft-Verify Loop

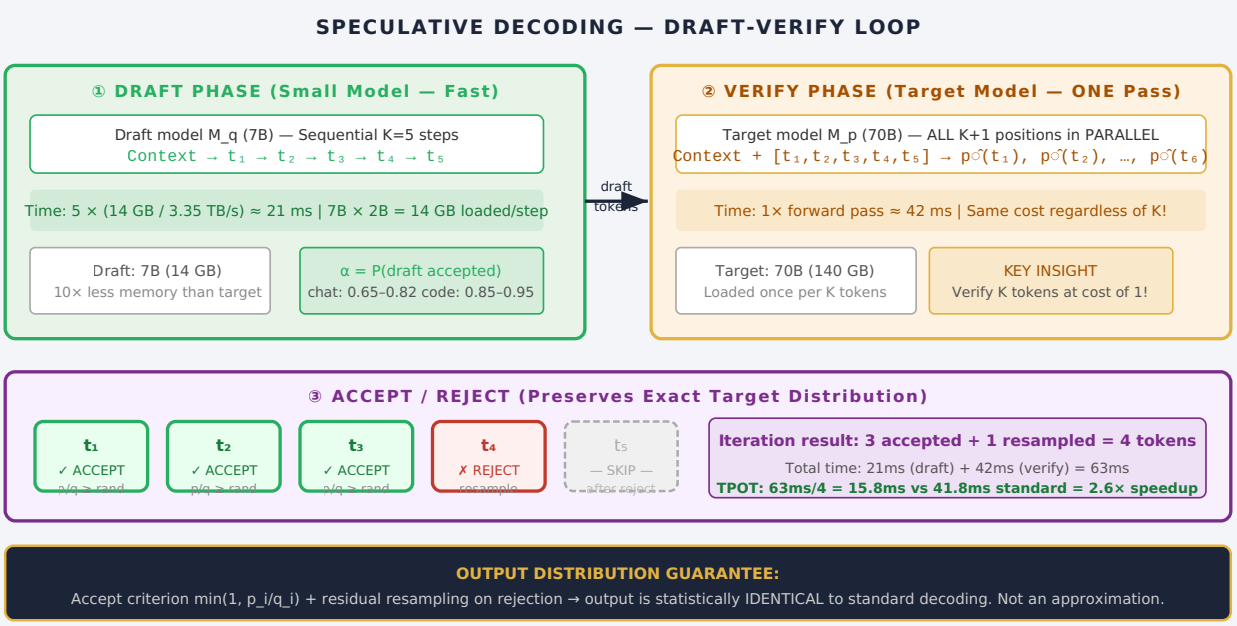



*Fig 13.1 — The Speculative Decoding draft-verify loop. The target model (70B) is loaded from HBM exactly once per iteration to verify K draft tokens in parallel — the same cost as generating 1 token. At α=0.75 and K=5, this delivers 2.4–2.6× TPOT improvement with zero quality degradation.*

### 13.1.3 The Speedup Formula — Derived from First Principles

```
SPEEDUP DERIVATION:

Let:
  α  = acceptance rate (P(any individual draft token is accepted))
  K  = draft tokens proposed per iteration
  c  = cost_draft / cost_target = ratio of forward pass times
       (7B draft / 70B target: c ≈ 7/70 = 0.10)

Expected tokens accepted per iteration:
  E[tokens] = Σ_{i=0}^{K} (i+1) × P(exactly i accepted)
            = (1 - α^{K+1}) / (1 - α)

  Intuition: if α=1.0 all the time, E[tokens] = K+1 (perfect, always accept all)
             if α=0.0 all the time, E[tokens] = 1 (always reject first token)

Time per iteration (normalized to T_target = 1):
  T_iter = K × T_draft + T_target = K × c + 1

Speedup over standard (1 token per T_target):
  Speedup = E[tokens] / T_iter
          = [(1 - α^{K+1}) / (1 - α)] / (K × c + 1)

PROFITABILITY CONDITION (when is spec decode worth it?):
  Speedup > 1.0
  ⇔ (1 - α^{K+1}) / (1 - α) > (K × c + 1)
  ⇔ E[tokens] > K × c + 1

  For c=0.10, K=5: need E[tokens] > 1.5 → need α > 0.26
  For c=0.10, K=3: need E[tokens] > 1.3 → need α > 0.19
  In practice: if model families are related (7B → 70B), α ≥ 0.65 almost always

NUMERICAL EXAMPLES (c = 0.10):
  K=5, α=0.60:  E=2.10,  T_norm=1.50  → Speedup = 1.40×
  K=5, α=0.70:  E=2.83,  T_norm=1.50  → Speedup = 1.89×
  K=5, α=0.80:  E=3.48,  T_norm=1.50  → Speedup = 2.32×
  K=5, α=0.90:  E=4.21,  T_norm=1.50  → Speedup = 2.81×
  K=5, α=0.95:  E=4.58,  T_norm=1.50  → Speedup = 3.05×

  K=8, α=0.80:  E=4.36,  T_norm=1.80  → Speedup = 2.42×  ← K=8 barely better than K=5!
  K=8, α=0.90:  E=5.70,  T_norm=1.80  → Speedup = 3.17×

KEY INSIGHT: Speedup is FAR more sensitive to α than to K.
  Increasing K from 5→8 at α=0.80: 2.32→2.42 = +4% improvement
  Increasing α from 0.80→0.90 at K=5: 2.32→2.81 = +21% improvement
  Invest in improving acceptance rate (better draft model, EAGLE), not just K.
```

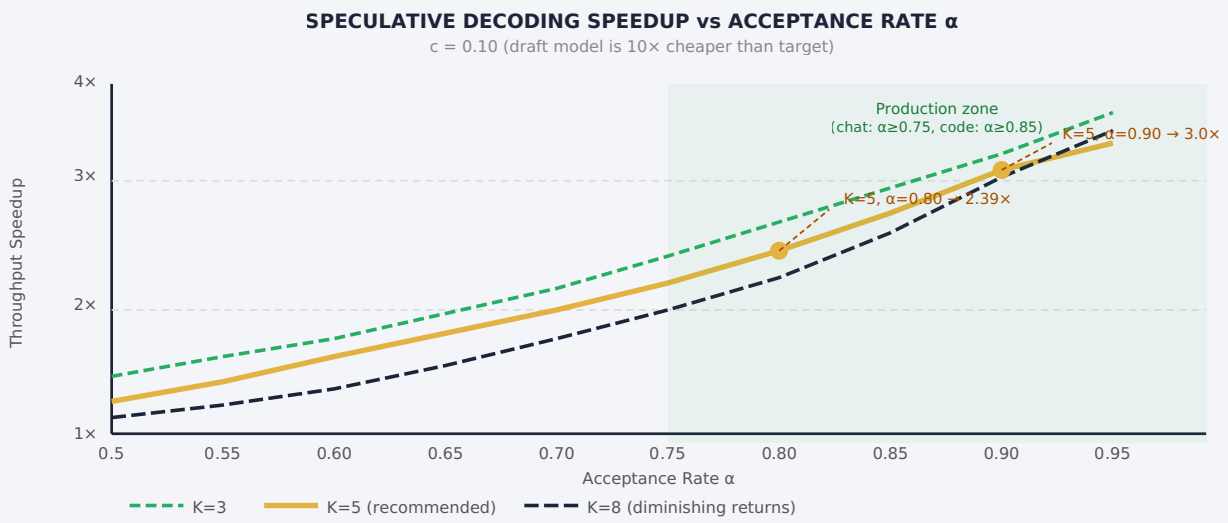



*Fig 13.2 — Speculative decoding speedup as a function of acceptance rate α for K=3, 5, and 8. Note that K=8 provides minimal improvement over K=5 at low α. The production sweet spot is K=5 with α≥0.75, achievable with a same-family draft model (e.g., LLaMA-3 8B drafting for LLaMA-3 70B).*

## 13.2 Draft Model Variants — Four Approaches

The core algorithm is the same for all variants; what differs is how draft tokens are generated. The tradeoff is memory overhead vs acceptance rate vs deployment complexity.

| Approach | Draft Source | Memory Overhead | Acceptance Rate α | Typical Speedup | Best Use Case |
|---|---|---|---|---|---|
| **Separate Small Model** | Same-family smaller model (e.g. LLaMA-3 8B → 70B) | **+16 GB (BF16 8B)** | **0.75–0.92** | **2.3–3.6×** | Code generation, structured output |
| **EAGLE / EAGLE-2** | ~200M param head on target hidden state | **~400 MB** | **0.85–0.93** | **3.0–3.96×** | Best overall; fine-tune on 200K samples |
| **Medusa Heads** | 4–5 MLP heads on frozen target | **~2 GB** | 0.65–0.80 | 2.0–2.5× | Memory-constrained; one-day fine-tune |
| **NGram / Lookup** | Prompt n-gram pattern matching | **~0 MB** | 0.40–0.70 | **1.3–1.7×** | RAG workloads, document Q&A, zero cost |

**EAGLE: WHY HIDDEN-STATE DRAFTS WIN**

Standard spec decode uses the draft model's *output tokens* to generate proposals. EAGLE uses the target model's *last hidden state* as additional context for the draft head — giving it access to the target's internal representation of what comes next. This is why EAGLE achieves α=0.85–0.93 even with a ~200M param draft head, vs ~0.80–0.88 for a 7B separate model. The fine-tuning requirement is modest: 1 GPU-hour on 200K samples.

**DRAFT MODEL SELECTION CONSTRAINT — NON-NEGOTIABLE**

Draft and target models MUST share an identical vocabulary and tokenizer. If tokenization differs even by one token ID remapping, the accepted tokens represent different text than the target expected — producing corrupted output. This is why Mistral-7B is the canonical draft for Mixtral-8×7B, not an unrelated 7B model. Before deploying any spec decode configuration, verify: `draft_tokenizer.vocab == target_tokenizer.vocab`.

## 13.3 Mixture of Experts — Architecture and Routing

### 13.3.1 The Dense FFN Problem and MoE Solution

In a standard transformer, every token passes through the same Feed-Forward Network (FFN) at every layer. This is computationally straightforward but wasteful: a token about astrophysics and a token about cooking both activate the same FFN weights, even though those tasks require entirely different specialized knowledge. The MoE insight is to replace one monolithic FFN with N smaller specialized expert FFNs, routing each token to only the K most relevant ones.

DENSE vs MoE COMPUTE COMPARISON:

Dense FFN:
  $FFN_{dense}(x) = W_2 \cdot GELU(W_1 \cdot x)$
  $W_1 \in R^{4d \times d}$,  $W_2 \in R^{d \times 4d}$

  FLOPs per token: $2 \times 2 \times d \times 4d = 16d^2$
  Memory required: $8d^2$ parameters (all loaded every token)

MoE FFN:
  $FFN_{MoE}(x) = \Sigma_{i \in top_K(x)} g_i(x) \cdot Expert_i(x)$

  $Expert_i$ = independent FFN with its own weights
  $g_i(x)$ = router weight (Softmax output, renormalized over top-K)

  FLOPs per token: $K \times 16d^2$ (only K experts execute)
  Memory required: $N \times 8d^2$ total, but only $K \times 8d^2$ loaded per token

COST COMPARISON (d=4096, N=8 experts, K=2):
  Dense: compute = $16 \times 4096^2$ = 268 MFLOPs/token
         memory = $8 \times 4096^2 \times 2$ bytes = 134 MB loaded per token
  MoE:   compute = $2 \times 268$ = 536 MFLOPs/token (2× more than dense!)
         But expert FFNs are 1/8 the size of dense FFN
         So total MoE weights = $8 \times 134/8$ = 134 MB total, $2/8 \times 134$ = 33 MB per token

KEY INSIGHT: MoE has SAME total parameter count flexibility, but only loads
K/N of the expert weights per token. The quality comes from having 8× more
parameter capacity, but compute cost is only 2× (K/1 = 2× a single dense FFN).

ACTUAL MODEL COMPARISON [SHIPPED]:

| Model | Total Params | Active/Token | MMLU | GPU Memory (BF16) |
|---|---|---|---|---|
| LLaMA-3 70B | 70B | 70B | 80.3% | 140 GB (2× H100) |
| Mixtral 8×7B | 47B | 13B | 70.6% | 94 GB (2× H100) |
| Mixtral 8×22B | 141B | 39B | 77.8% | 282 GB (4× H100) |
| DeepSeek-V3 | 671B | 37B | 88.5% | 1342 GB (10× H100!) |
| GPT-4 (est) | ~1.8T (MoE) | ~220B (est) | 86.4% | ~3.6 TB (est) |

PRINCIPLE: Memory scales with total params. Compute scales with active params.
DeepSeek-V3: frontier quality at LLaMA-3 70B compute, but requires 10× H100s!

### 13.3.2 Router Mechanics — How Tokens Are Assigned

```
ROUTER ALGORITHM (per transformer layer, N experts, top-K routing):

Input: token hidden state x ∈ R^{d_model}

Step 1 — Score all experts:
  logits = W_router · x        [W_router ∈ R^{N × d_model}]
  scores = Softmax(logits)     [scores ∈ R^N, sum = 1.0]

Step 2 — Select top-K experts:
  top_k_indices = argsort(scores)[-K:]
  top_k_weights = scores[top_k_indices]
  top_k_weights /= sum(top_k_weights)  # Renormalize to sum=1

Step 3 — Compute MoE output:
  output = Σ_{i∈top_k_indices}  top_k_weights[i] × Expert_i(x)

DEEPSEEK-V3 INNOVATION — Shared + Routed Experts:
  Total: 64 routed experts + 2 shared experts
  Top-K: top-6 from routed + all 2 shared = K_total = 8 active per token
  Shared experts ALWAYS activate (learn universal patterns)
  Routed experts specialize (prevent expert collapse)

  Sparsity: 6/64 = 9.4% of routed experts active per token
  This is why DeepSeek-V3 can scale to 671B total with 37B active.
```

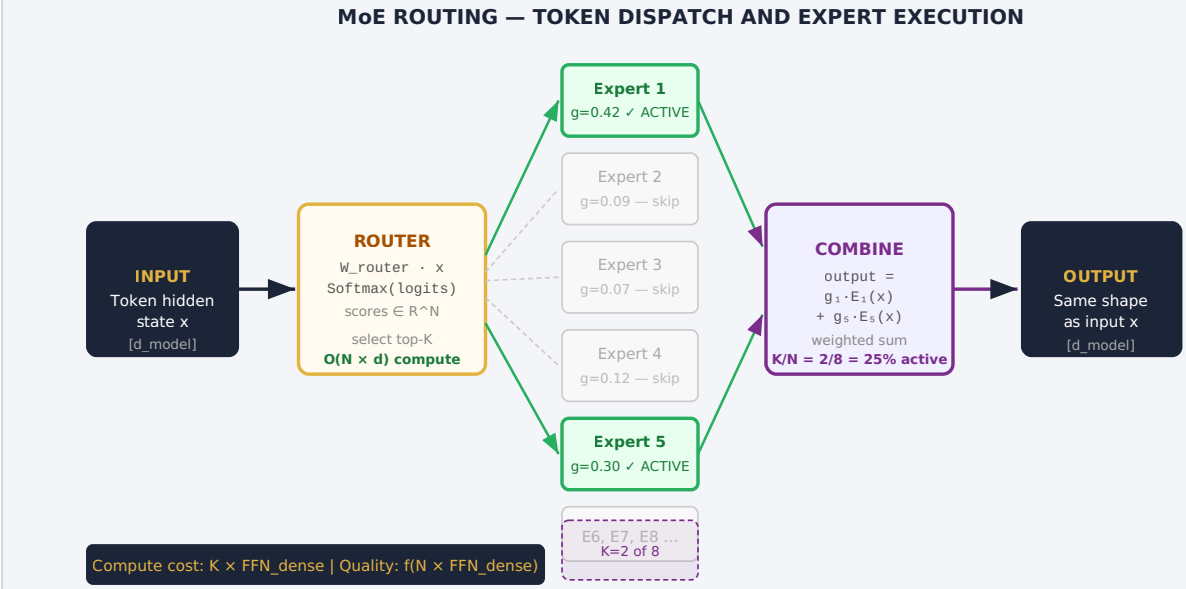



*Fig 13.3 — MoE routing for a single token through 8 experts with K=2. The router scores all N experts in O(N×d) compute, selects the top-K, executes those experts in parallel, and combines outputs. The 6 non-selected experts contribute zero compute but their weights remain in memory.*

### 13.3.3 Expert Collapse and Load Balancing

Without explicit balancing constraints, MoE training degenerates into *expert collapse* — a positive-feedback loop where the router concentrates tokens on a few experts, those experts improve fastest due to more gradient signal, and the router routes even more tokens to them. The end result: a model with 8 experts that behaves like a model with 1–2.

```
LOAD BALANCE LOSS (Auxiliary Loss Approach):

L_total = L_LM + λ × L_balance

Where:
  L_LM      = standard language modeling loss (cross-entropy)
  λ         = balance coefficient (typically 0.01–0.1)

  L_balance = α × N × Σᵢ (fᵢ × Pᵢ)

  fᵢ = fraction of tokens dispatched to expert i in current batch
  Pᵢ = average router probability assigned to expert i
  N  = number of experts

  Perfect balance: fᵢ = Pᵢ = 1/N for all i → L_balance = 1/N
  Collapsed state: f₁=1, f₂…fₙ=0 → L_balance = 1 (N× higher)

CAPACITY FACTOR (Inference guardrail):

  capacity_i = (batch_size × K / N) × capacity_factor

  capacity_factor = 1.0: zero token dropping (wastes HBM)
  capacity_factor = 1.25: allows 25% overflow (standard production)

  Tokens exceeding capacity: dropped or passed through residual only

DEEPSEEK-V3 INNOVATION — Auxiliary-Loss-Free Load Balancing:
  Maintains per-expert bias term b_i, updated each step:

  router_score_i = g_i(x) + b_i       (biased routing)
  update: if expert_i overloaded → b_i -= γ
          if expert_i underloaded → b_i += γ

  γ = 0.001 (very small updates, prevents routing oscillation)

  Result: Near-perfect load balance without adding noise to L_LM.
  This is why DeepSeek-V3 can scale to 256 experts with stable training.
```

## 13.4 MoE Infrastructure — Expert Parallelism and All-to-All

### 13.4.1 Expert Parallelism: A New Dimension

Dense models use Tensor Parallelism (TP) and Pipeline Parallelism (PP) during inference. MoE introduces a third dimension: Expert Parallelism (EP), where different experts reside on different GPUs. This enables serving a 671B MoE model, but introduces a communication pattern that is fundamentally different from TP AllReduce — and in many ways more expensive.

```
EXPERT PARALLEL COMMUNICATION: All-to-All vs AllReduce

ALLREDUCE (Tensor Parallelism):
  ALL GPUs send SAME data to ALL GPUs
  Pattern: symmetric, predictable, well-pipelined by NCCL
  Algorithm: Ring-AllReduce, latency-bound for small messages
  Use: aggregate partial GEMM results across TP group

ALL-TO-ALL (Expert Parallelism):
  Each GPU sends DIFFERENT data to EACH OTHER GPU
  Pattern: IRREGULAR — depends on routing decisions for this batch!
  Message sizes vary per GPU pair (depends on token distribution)
  Cannot be predicted statically → NCCL scheduling is harder

TOKEN DISPATCH SEQUENCE (EP=8, 8 experts, 1 per GPU):

  Step 1: LOCAL ROUTING
    Each GPU runs router on its local token batch
    → Get (token_id, expert_id) pairs for all local tokens

  Step 2: ALL-TO-ALL DISPATCH
    GPU 0 sends tokens for Expert 1 → GPU 1
    GPU 0 sends tokens for Expert 5 → GPU 5
    (simultaneously all GPUs send to all other GPUs)
    Volume = batch × K × d_model × 2 bytes (total, split across EP-1 peers)

  Step 3: LOCAL EXPERT COMPUTE
    Each GPU runs GEMM for its local expert on received tokens
    GEMM: [tokens_received, d_model] × [d_model, d_ff]
    Problem: with EP=8 and K=2, each GPU receives only batch×K/EP tokens!
    At batch=32: 32 × 2 / 8 = 8 tokens per expert → tiny GEMM!

  Step 4: ALL-TO-ALL COMBINE
    Each GPU sends computed results back to originating GPUs
    Same volume as dispatch (Step 2)

  Step 5: WEIGHTED COMBINATION
    Each GPU combines results from its token's top-K experts

COMMUNICATION VOLUME (per MoE layer):
  Dispatch  = batch × seq_len × K × d_model × 2 bytes
  Combine   = same
  Total     = 2 × batch × seq_len × K × d_model × 2 bytes

  LLaMA-scale MoE (batch=32, seq=4096, K=2, d=4096, BF16):
    = 2 × 32 × 4096 × 2 × 4096 × 2 = 8.6 GB per MoE layer per step!

  For 32 MoE layers: 8.6 GB × 32 = 275 GB per forward pass
```

```
At IB HDR (50 GB/s effective): 275/50 = 5.5 seconds — untenable!
At NVLink within-node (900 GB/s): 275/900 = 0.31 seconds — acceptable
```

**EXPERT PLACEMENT STRATEGIES — COMMUNICATION COST IMPACT**



*Fig 13.4 — Expert placement strategies. Naive cross-node All-to-All over InfiniBand (50 GB/s effective) is 18× slower than within-node NVLink All-to-All (900 GB/s). The critical constraint: Expert Parallelism degree EP must fit within one NVLink domain (EP ≤ 8 for H100 DGX nodes).*

### 13.4.2 The Small-Batch Expert Problem

Even with perfect NVLink placement, MoE inference at typical batch sizes faces a serious compute efficiency problem: each expert receives only a small fraction of the total batch.

```
PER-EXPERT GEMM SIZE (Batch=32, N=8 experts, K=2):

Tokens per expert (perfectly balanced): 32 × 2 / 8 = 8 tokens

Dense FFN (batch=32):
  GEMM: [32, 4096] × [4096, 14336]
  Arithmetic Intensity: 32 × 4096 × 14336 × 2 / (4096 × 14336 × 2 + 32 × 14336 × 2)
                      ≈ 32 FLOP/byte (very memory-bound, but manageable)

MoE expert (batch=8 per expert):
  GEMM: [8, 4096] × [4096, 14336]
  Arithmetic Intensity ≈ 8 FLOP/byte  (4× worse than dense at batch=32!)

  H100 ridge point: ~295 FLOP/byte
  Expert AI = 8 << 295 → EXTREMELY memory-bandwidth limited

Implications:
  GPU compute utilization per expert: ~5% (vs ~10-15% for dense at batch=32)

  Mitigation strategy 1: "Expert batching" — aggregate requests from
    multiple concurrent users into one expert call (requires scheduling)

  Mitigation strategy 2: Increase overall batch size until per-expert
    tokens exceed ~64 for reasonable GEMM efficiency

  Target per-expert batch for efficiency: ≥ 64 tokens
  Required total batch: 64 × N / K = 64 × 8 / 2 = 256 tokens minimum!
```

## 13.5 Profiling Cookbook — Diagnosing in Production

### 🔬 SPEC DECODE: DIAGNOSING LOW ACCEPTANCE RATE

**Symptom:** Speculative decoding enabled but TPOT improvement less than 1.5×.

**Step 1 — Measure actual acceptance rate:**

```
# vLLM (0.5+): track via prometheus metrics
curl http://localhost:8000/metrics | grep speculative
# Look for: vllm:spec_decode_draft_acceptance_rate
# Healthy: ≥ 0.70 for chat, ≥ 0.85 for code
# Unhealthy: < 0.50 → spec decode likely hurting performance!

# Verify expected speedup:
alpha = 0.55; K = 5; c = 0.10
E_tokens = (1 - alpha**(K+1)) / (1 - alpha)  # = 1.87
speedup = E_tokens / (K*c + 1)               # = 1.87/1.5 = 1.25×
# If observed < formula → overhead in batch management or KV sync
```

**Decision tree:**

- α < 0.50 on creative/diverse workloads → disable spec decode, use continuous batching instead
- α < 0.65 on code tasks → wrong draft model (must be same tokenizer family)
- α ≥ 0.80 but speedup still low → K too high (try K=3), or draft overhead high (check GPU memory)
- α varies widely across requests → segment by request type; apply spec decode only to code/SQL queries

## 🔬 MOE: DIAGNOSING EXPERT LOAD IMBALANCE

**Symptom:** P99 TTFT elevated, all GPUs show 95%+ SM utilization, but throughput 30–40% below theoretical.

```
# DCGM metrics for expert utilization:
dcgmi dmon -e 1001,1002,1006,1007 -d 1000 -c 10
# 1001: SM_ACTIVE, 1002: SM_OCCUPANCY, 1006: DRAM_ACTIVE, 1007: NVLink BW

# vLLM expert balance metrics:
curl http://localhost:8000/metrics | grep moe_expert
# vllm:moe_expert_load{expert_id="0"} 0.125  ← balanced (1/8)
# vllm:moe_expert_load{expert_id="4"} 0.450  ← HOT EXPERT!

# Python diagnostic script:
expert_loads = [0.10, 0.09, 0.11, 0.08, 0.45, 0.07, 0.05, 0.05]
mean_load = sum(expert_loads) / len(expert_loads)   # = 0.125
max_load  = max(expert_loads)                       # = 0.45
imbalance_ratio = max_load / mean_load              # = 3.6 → ALERT!
# Threshold: alert if imbalance_ratio > 2.0
```

**Root causes and fixes:**

- *Training imbalance:* auxiliary balance loss weight too low (λ < 0.01) → retrain or use DeepSeek-V3's auxiliary-loss-free method
- *Workload mismatch:* production traffic contains domain(s) that training underrepresented → add capacity factor (1.5×) for hot experts

• *EP placement:* "hot" expert on a GPU with NVLink congestion from TP traffic → re-assign expert placement based on topology profiling

## 13.6 Combining Speculative Decoding and MoE

Deploying spec decode on MoE targets (e.g., Mixtral 8×7B) introduces two complications: draft model selection and verification cost. The verification forward pass on the target now includes All-to-All routing for all K draft tokens simultaneously — but since they are batched as a single prefill, this is generally efficient.

| Target Model | Recommended Draft | Why | Expected Speedup |
|---|---|---|---|
| Mixtral 8×7B (47B total) | Mistral-7B (dense) | **Same tokenizer; 7B is 10× cheaper** | **~2.1× TPOT** |
| Mixtral 8×22B (141B total) | Mixtral 8×7B or Mistral-7B | 7B: same vocab; 8×7B: better α | 1.8–2.3× |
| DeepSeek-V3 (671B total) | DeepSeek-V2-Lite (16B) | Same vocabulary and architecture family | ~2.0× |
| Any MoE | **Unrelated model** | **Different tokenizer → corrupted output** | **FAILS** |

**KV CACHE WITH MOE — NO SPECIAL HANDLING NEEDED**

Attention layers in MoE models are identical to dense transformers — the MoE structure only affects the FFN sublayers. KV cache allocation, paged attention, and prefix caching work exactly the same as for dense models. Both draft and target KV caches have standard shape: [layers, heads, seq_len, head_dim]. No MoE-specific KV logic required.

## 13.7 Disaggregated Prefill-Decode — The Next Frontier

Speculative decoding optimizes decode throughput within a single GPU pool. A complementary technique — *disaggregated prefill-decode* — attacks TTFT and TPOT simultaneously by recognizing that prefill and decode have fundamentally different hardware requirements and should run on separate, specialized GPU pools.

```
THE PREFILL/DECODE HARDWARE MISMATCH:

PREFILL (prompt processing):
  Computes attention over the full prompt: O(S²) FLOPs
  Processes all S tokens simultaneously: large GEMM
  Regime: COMPUTE-BOUND (high arithmetic intensity)
  Optimal hardware: maximize FLOPS (high SM count, high FLOPS/dollar)
  H100 utilization during prefill: 60–80%

DECODE (token generation):
  Single token per step, batch-limited
  Arithmetic Intensity: ~1 FLOP/byte (deeply memory-bound)
  Regime: MEMORY-BANDWIDTH-BOUND
  Optimal hardware: maximize HBM capacity and bandwidth
  H100 utilization during decode: 0.3–5%

THE PROBLEM WITH COLOCATED P/D:
  Running both on the same GPU pool = hardware utilization compromise
  GPUs are sized for compute (because prefill needs it)
  But 80–95% of serving wall time is decode (where compute is wasted)

  Result: GPU fleet sized by prefill compute, underutilized for decode

DISAGGREGATED ARCHITECTURE (NVIDIA Dynamo / DistServe):

  Prefill pool (P-nodes):  fewer, compute-optimized GPUs
                            H100 SXM5 or A100 (high FLOPS/dollar)
                            Handles all prompt processing

  Decode pool (D-nodes):   more, HBM-optimized GPUs
                            H200 (higher HBM BW) or custom (large HBM)
                            Handles all token generation

  KV Transfer:             after prefill, KV cache transferred P → D
                            over NVLink/InfiniBand
                            Volume: 2 × n_layers × n_heads × S × d_head × 2 bytes
                            LLaMA-3 70B, S=2048: ~8 GB per request (significant!)

PRODUCTION RESULTS (Meta DistServe, NVIDIA Dynamo):
  TTFT improvement:   ~3× (prefill no longer blocked by decode queue)
  TPOT improvement:   ~1.5× (decode pool optimized for bandwidth)
  Overall throughput: ~1.35× at 2:1 decode-to-prefill node ratio
  Cost reduction:     ~27% at scale (right-sizing each pool independently)

WHEN TO INVEST: Justification threshold ~3× request volume above baseline
  At <500 requests/min: colocated P/D simpler and sufficient
  At >2000 requests/min with SLA constraints: disaggregation justified
```

## 13.8 Worked Example: Deploying Speculative Decoding for a Code Assistant

**WORKED EXAMPLE — PRINCIPAL LEVEL**

**Scenario:** You are deploying a LLaMA-3 70B code assistant on 2× H100 SXM5 (160 GB total HBM). Current throughput: 1,200 tok/s at batch=16, TPOT=85ms. SLA requires TPOT ≤ 40ms. Your traffic is 70% code completion, 30% chat.

**Step 1: Establish baseline arithmetic intensity**

```
70B model, BF16: 140 GB loaded per decode step
TPOT = 140 GB / 3.35 TB/s ≈ 41.8 ms (memory-bound at batch=1)
At batch=16: 16 × 140 GB... wait, batch doesn't increase memory load for weights
  but DOES increase arithmetic intensity:
  AI = 2 × 70B × batch / (140 GB × batch) ≈ 1 FLOP/byte still at batch=16

Actually: at batch=16, each token still requires the same 140 GB weight load
but you get 16 tokens processed simultaneously:
  TPOT at batch=16 ≈ 41.8 ms / 1 = ~42 ms per batch iteration
  Throughput = 16 tokens / 42 ms = 381 tok/s ← low utilization

Measured TPOT=85ms > 42ms theoretical → 2× memory overhead from KV cache
  at batch=16, seq=2048, 80 layers: KV size = 2 × 80 × 8 × 2048 × 128 × 2B ≈ 84 GB
  Total HBM access per step: 140 GB (weights) + 84 GB (KV) = 224 GB
  Expected TPOT = 224 / 3,350 ≈ 67 ms ← close to observed 85ms (overhead explained)
```

**Step 2: Configure speculative decoding**

```
Draft model: LLaMA-3 8B (same tokenizer family) [SHIPPED]
  Memory: 8B × 2 bytes = 16 GB additional HBM
  Total: 140 GB (70B) + 16 GB (8B) = 156 GB of 160 GB available ✓ (tight!)

Expected acceptance rates (from production data):
  Code completion (70%): α = 0.88
  Chat (30%):            α = 0.72
  Blended α = 0.70 × 0.88 + 0.30 × 0.72 = 0.616 + 0.216 = 0.832

Speedup calculation (K=5, c = 16/140 = 0.114):
  E[tokens] = (1 - 0.832^6) / (1 - 0.832) = (1 - 0.334) / 0.168 = 3.96
  T_iter = K × c + 1 = 5 × 0.114 + 1 = 1.57
  Speedup = 3.96 / 1.57 = 2.52×

New TPOT = 85ms / 2.52 = 33.7ms ← MEETS SLA of 40ms ✓

New throughput = 16 tokens / 33.7ms × 2.52 = ~1,200 tok/s (but we get more tokens/iter!)
Actually: throughput ≈ original × speedup = 1200 × 2.52 = ~3,024 tok/s
```

**Step 3: Memory safety check and production configuration**

```
HBM budget: 160 GB total
  - 70B model (BF16):      140 GB
  - 8B draft model (BF16):  16 GB
  - Available for KV cache:   4 GB  ← PROBLEM! Only 4 GB for KV cache

  Fix: Use FP8 quantization for 70B model:
    70B × 1 byte = 70 GB (FP8)
    Available for KV + draft: 160 - 70 - 16 = 74 GB for KV cache
    At batch=16, seq=2048: ~84 GB KV needed → still tight

  Better fix: FP8 weights + BF16 activations + INT8 KV cache:
    70B FP8 weights = 70 GB
    8B BF16 draft   = 16 GB
    KV INT8         = 42 GB (half of BF16)
    Total: 128 GB ✓ — 32 GB headroom for OS and overheads

vLLM production config:
  --model meta-llama/Llama-3-70b-instruct
  --speculative-model meta-llama/Llama-3-8b-instruct
  --num-speculative-tokens 5
  --quantization fp8
  --kv-cache-dtype fp8_e5m2
  --tensor-parallel-size 2
  --gpu-memory-utilization 0.92
```

**Business impact:** 2.52× throughput improvement means serving 2.52× the request volume on the same hardware. At $3/GPU-hr × 2 GPUs × 720 hr/month = $4,320/month infrastructure cost. Without spec decode, doubling capacity requires 2 additional GPUs: $4,320 additional spend. Spec decode eliminates that cost — saving **$51,840/year** on a single 2-GPU deployment.

## 13.9 Expert Load Distribution — Visualizing Imbalance

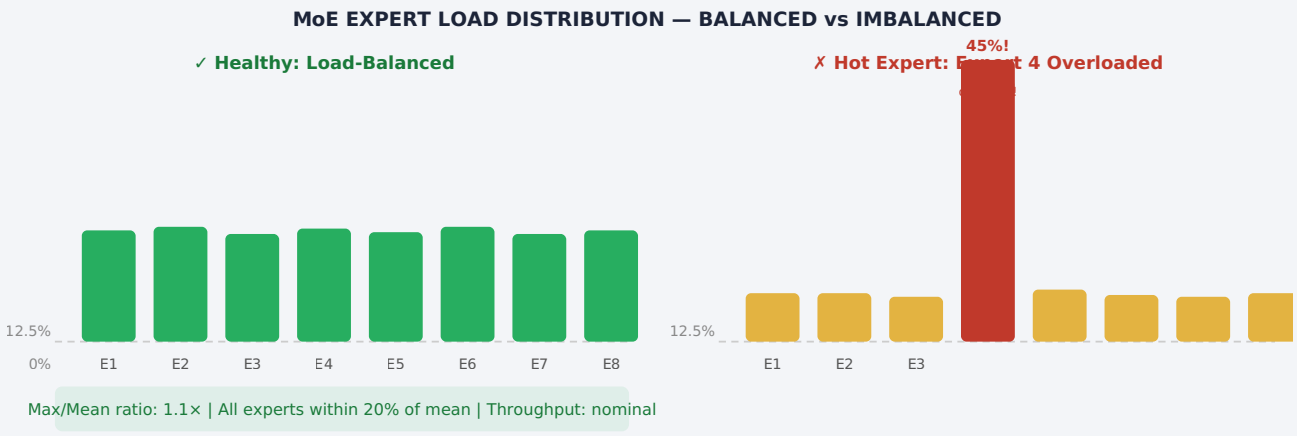



*Fig 13.5 — Expert load distribution in balanced vs. hot-expert MoE deployments. The insidious aspect of expert imbalance: all 8 GPUs may show 95%+ SM utilization while throughput is 34% below theoretical. Expert 4's tokens overflow its capacity factor buffer and are dropped silently. Monitor `vllm:moe_expert_load` continuously.*

### Key Takeaways

**CORE CONCEPTS**

Speculative decoding is exact — not an approximation. The accept/reject criterion (min(1, p_i/q_i) with residual resampling) guarantees output is statistically identical to standard autoregressive decoding. It is safe to deploy without quality regression testing.

Speedup is most sensitive to acceptance rate α. Increasing K from 5→8 at α=0.80 yields only +4% speedup improvement. Increasing α from 0.80→0.90 at K=5 yields +21%. Invest in better draft models (EAGLE, same-family smaller model), not just larger K.

MoE decouples quality from compute. Quality scales with total parameters; compute cost scales with active parameters (K/N fraction). DeepSeek-V3 achieves 88.5% MMLU (frontier quality) at ~37B active params per token — comparable compute to LLaMA-3 40B dense, but with 671B total knowledge capacity.

Expert load imbalance is a silent performance killer. A 3.6× imbalance (one expert receiving 45% of traffic) causes 34% throughput degradation while all GPU utilization metrics appear healthy. Only expert-level routing metrics expose this failure mode.

**COST & PRODUCTIVITY IMPACT**

Speculative decoding at α=0.83, K=5 delivers 2.52× TPOT reduction on a mixed code/chat workload — eliminating the need for hardware scaling to meet a 40ms TPOT SLA. On a 2×H100 deployment at $3/GPU-hr, that saves ~$52K/year in additional hardware cost.

- Expert Parallelism within a single NVLink domain (EP=8, within-node) achieves 18× lower All-to-All latency than cross-node InfiniBand placement. This is not a marginal improvement — it is the difference between a viable and unviable MoE deployment.
- Disaggregated prefill-decode reduces infrastructure cost by ~27% at scale by right-sizing compute-optimized and HBM-optimized GPU pools independently. Justified at >2000 requests/minute with tight TTFT SLAs.

**PRINCIPAL-LEVEL INSIGHT**

- The combined optimization stack (FP8 + speculative decoding + disaggregated P/D + MoE architecture) delivers 5–10× throughput improvement over a BF16 dense baseline — but these techniques interact. Speculative decoding on MoE targets is slightly less efficient due to routing overhead in the verification pass; expect 85–90% of the speedup you'd get on a comparable dense target.
- Draft model selection is constrained by tokenizer compatibility, not model architecture. Before any spec decode deployment, verify identical vocabularies. Any tokenizer mismatch produces corrupted output that may be subtle (slightly wrong probability distributions) rather than obviously wrong (garbage tokens), making it dangerous to test only on simple prompts.
- Per-expert batch size is the hidden bottleneck in MoE inference efficiency. For production MoE serving at EP=8, you need total batch ≥ 256 tokens to achieve reasonable arithmetic intensity per expert (≥ 64 tokens each). Below this threshold, each expert GEMM operates at AI ≈ 8 FLOP/byte — 37× below H100's ridge point — wasting 97% of compute capacity.

## Review Questions

**FOUNDATIONAL** Build the mental model — no math required

1. Speculative decoding claims to produce output "statistically identical" to standard autoregressive decoding. Explain in plain English why this is true — what happens at each position when the draft token is accepted vs. rejected? What would go wrong if you always accepted draft tokens unconditionally?
2. A dense 70B model and a MoE model with 8 experts and K=2 active per token both produce the same output hidden state dimension. Explain how the MoE model can have better quality than the dense model at the same compute cost per token, and why it still requires more GPU memory to serve.
3. You are told that "all GPUs in our MoE cluster show 96% SM utilization but throughput is 30% below expected." Without looking at any other metrics, name the most likely root cause and describe which single monitoring metric would confirm or deny your hypothesis.

**INTERMEDIATE** Apply and calculate — production context

1. A legal document summarization service uses LLaMA-3 70B. Production measurement shows acceptance rate $\alpha=0.55$ with $K=5$ and draft cost ratio $c=0.10$. (a) Calculate the theoretical speedup. (b) Is speculative decoding worth deploying at this

acceptance rate? (c) The team proposes switching to K=8 to "get more tokens per iteration" — calculate the speedup for K=8 at α=0.55 and explain the result.

2. You are deploying Mixtral 8×7B with EP=8 (one expert per GPU) across a 2-node server with 8 GPUs per node and NVLink 4.0 (900 GB/s). Batch=64 requests, seq_len=1024, d_model=4096, K=2, BF16. (a) Calculate the All-to-All communication volume per MoE layer per forward pass. (b) At what fraction of token dispatch would cross-node traffic begin if experts were distributed uniformly across all 16 GPUs instead of 8?

3. A Mixtral 8×7B deployment monitoring shows Expert 3 consistently receives 38% of tokens while others share the remaining 62% evenly. The model was trained with auxiliary balance loss (λ=0.01). Capacity factor = 1.25. Calculate the token overflow rate at Expert 3 for a batch of 512 tokens (K=2) and explain the impact on output quality.

**PRINCIPAL LEVEL** Design, decide, and defend — architecture reviews

1. **Spec Decode + MoE Architecture Design:** Your team is deploying DeepSeek-V3 (671B total, 37B active, 64 routed + 2 shared experts, top-6 routing from routed). You have a budget for 12× H100 SXM5 (80 GB each = 960 GB total). (a) What is the minimum viable spec decode configuration? Identify the draft model, K value, and justify the choice with memory math. (b) What EP configuration and node topology do you recommend for the 64 routed experts across 12 GPUs? Show the All-to-All communication volume per layer. (c) What monitoring setup would you implement on Day 1 to detect expert imbalance before it impacts SLAs?

2. **Disaggregated P/D Sizing:** A production serving system handles 3,000 requests/minute with average prompt length 1,500 tokens and average output length 800 tokens. Current setup: 32× H100 colocated. TTFT P99 = 800ms (SLA = 300ms). You are proposing disaggregated prefill-decode. (a) Estimate the ratio of prefill compute to decode compute. (b) How would you size the P-pool vs D-pool? (c) What is the KV transfer volume per second and can it be served over InfiniBand HDR (200 Gbps)? Justify your architecture with numbers.