PART III · CHAPTER 6

Inference System Architecture

From user request to streamed tokens — prefill vs. decode physics, PagedAttention, disaggregated serving, SLA engineering, and the fleet sizing formula that ties it all together.

Book: AI/ML Infrastructure from Silicon to Scale — V2 Edition | **Author:** Venkat Vinjam | **Cross-ref:** Ch 4: FlashAttention Ch 11: KV cache engineering Ch 13: Speculative decoding Ch 17: Inference observability

> *"Inference is where your model meets reality — and where you pay for every engineering mistake made during training infrastructure design. TTFT is the promise you make to the user. TPOT is the promise you keep. This chapter is about not breaking either promise."*
>
> — LLM Serving Engineering Principle

This chapter covers the complete LLM inference stack: from the physics of prefill and decode phases through production-grade serving systems (vLLM, TRT-LLM), disaggregated architectures, SLA engineering, and the fleet sizing formula. Every number is derivable from first principles; every recommendation is quantified with production references.

6.1 The Two Phases: Prefill vs. Decode

LLM inference is not a single computation — it is two fundamentally different operations that happen sequentially. Every production serving architecture is, at its core, a system for managing the tension between these two phases.

```
THE TWO PHASES OF LLM INFERENCE:

PHASE 1 — PREFILL:
  Input:   Full prompt (S tokens) processed in parallel (one forward pass)
  Output:  First token + KV cache populated for all S positions
  GEMM:    [S × d_model] × [d_model × 4d_model]  — large, balanced GEMM
  AI:      ~(S × d_model) / model_bytes  — high at large S
  Regime:  COMPUTE-BOUND for S > ~128  (Tensor Core saturated)
  Metric:  TTFT — Time To First Token

PHASE 2 — DECODE:
  Input:   One token (last generated) per autoregressive step
  Output:  Next token logits + one new KV row appended per layer
  GEMM:    [1 × d_model] × [d_model × 4d_model]  — vector-matrix (GEMV)
  AI:      ~1 FLOP/byte (load all weights for 1 FLOP per weight element)
  Regime:  MEMORY-BANDWIDTH-BOUND  (always, for any reasonable batch)
  Metric:  TPOT — Time Per Output Token

TPOT THEORETICAL MINIMUM (bandwidth floor):
  TPOT_min = model_bytes / HBM_bandwidth

  LLaMA-3 70B [SHIPPED]:
    BF16 on H100:  140 GB / 3.35 TB/s  =  41.8 ms/token  ← physics floor
    FP8  on H100:   70 GB / 3.35 TB/s  =  20.9 ms/token  ← 2× from quantization
    INT4 on H100:   35 GB / 3.35 TB/s  =  10.4 ms/token  ← 4× from quantization
    BF16 on H200:  140 GB / 4.8 TB/s   =  29.2 ms/token
    BF16 on MI300X: 140 GB / 5.3 TB/s  =  26.4 ms/token

  These are hard physics floors — no software optimization crosses them.
  The only escapes: more HBM bandwidth, quantization, or batching.
```

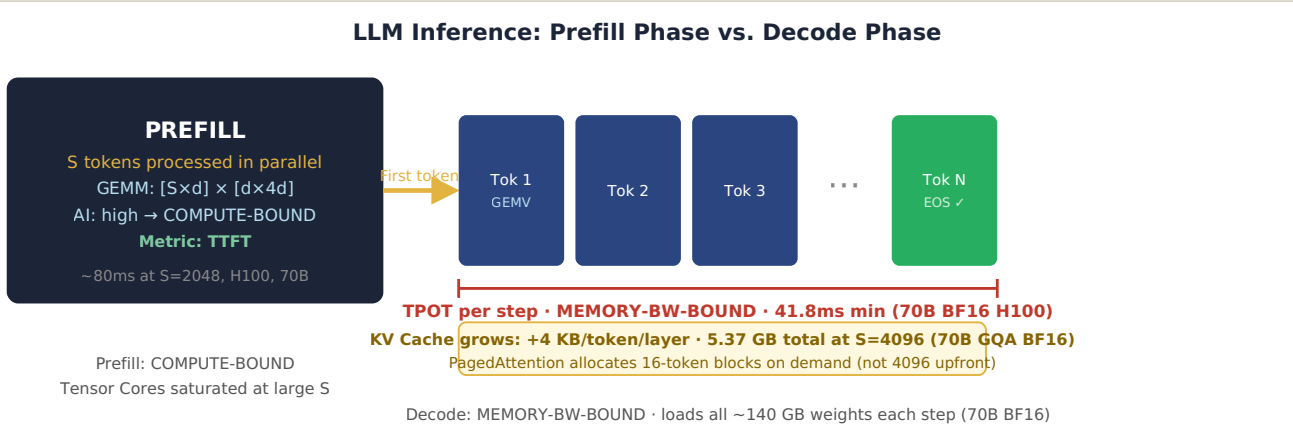


Figure 6.1 — Prefill and decode phase anatomy. Prefill (left): all S prompt tokens processed in one parallel GEMM — compute-bound. Decode (right): one token per step, loads full weight matrix per step — memory-bandwidth-bound, 41.8ms minimum per token on H100 for LLaMA-3 70B BF16. The KV cache grows by 4 KB per token per layer.

6.2 Continuous / In-Flight Batching

```
STATIC BATCHING — TWO FAILURE MODES:

FAILURE 1 — PADDING WASTE:
  Batch = 4 requests, max output = 200 tokens:
    Request A finishes at token 20  → padded 180 steps (90% waste)
    Request B finishes at token 50  → padded 150 steps (75% waste)
    Request C finishes at token 150 → padded  50 steps (25% waste)
    Request D finishes at token 200 → no padding
  Compute wasted: (180+150+50) / (4×200) = 47.5% of GPU cycles wasted on padding

FAILURE 2 — HEAD-OF-LINE BLOCKING:
  Request E arrives at t=10ms. D still running at t=200ms.
  E must wait 190ms in queue. E's TTFT = 190ms queue + prefill time.
  Typical GPU utilization under static batching: 40–55%.

CONTINUOUS BATCHING (iteration-level scheduling, Orca 2022):
  At every decode step:
    1. Execute one decode step for all active requests
    2. Check: any request finished (EOS or max_tokens)?
    3. If yes: evict that slot, immediately admit next request from queue
    4. No padding — each request at its exact current sequence length

  Throughput gain vs static batching: 10–30× improvement [empirically measured]
  This is the single most impactful inference optimization. If your serving stack
  is not using continuous batching, start here before any kernel optimization.
```

6.3 vLLM and PagedAttention

```
PRE-VLLM KV CACHE ALLOCATION:
  Must reserve contiguous memory for max_sequence_length upfront.

  LLaMA-3 70B, max_seq=4096, BF16, H100 with TP=2 (35 GB weights per GPU):
    Per-request KV = 2 × 80 × 8 × 128 × 4096 × 2 = 5.37 GB reserved upfront
    Available for KV: 35 GB  →  35 / 5.37 = 6 max concurrent requests
    Avg actual seq len = 512 tokens  →  KV used per request = 0.67 GB
    Fragmentation: (5.37 - 0.67) / 5.37 = 87.5% of KV memory wasted!

PAGEDATTENTION (Kwon et al., 2023):
  Inspiration: OS virtual memory paging
  Block size: 16 tokens per block (configurable: 8, 32)
  Allocation: on-demand — one block as each 16 tokens arrive

  Memory efficiency:
    Fragmentation:      <5% (vs 87.5% naive)
    Concurrent requests: ~6–10× more per GPU vs naive allocation

  BLOCK SHARING (prefix caching):
    Requests sharing system prompt share physical KV blocks (reference-counted).
    System prompt 2048 tokens: computed ONCE, pointer-shared across all requests.
    TTFT improvement: 50–80% for system-prompt-heavy workloads.

  PREEMPTION (when KV pool full):
    Swap low-priority KV to host DRAM: ~1–2s via PCIe (128 GB/s)
    Prevention: target <80% KV utilization always — monitor vllm:gpu_kv_cache_usage_perc
```

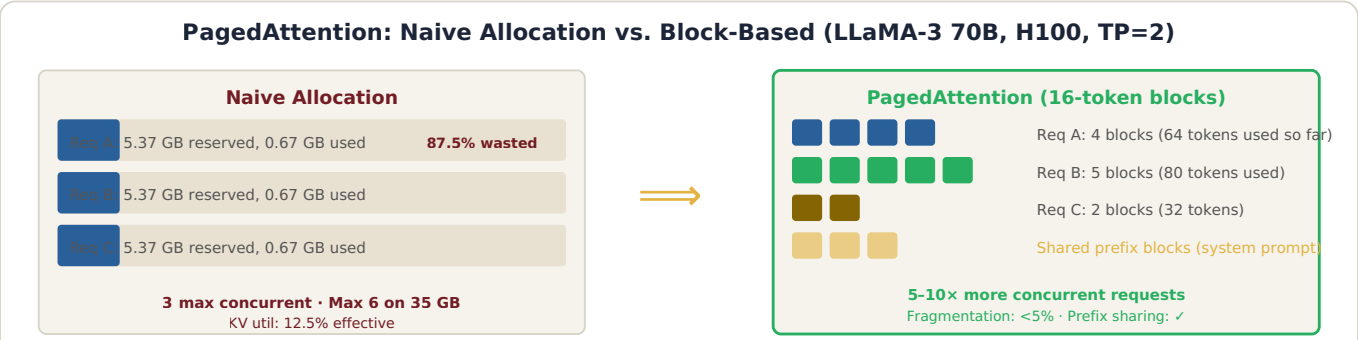


Figure 6.2 — PagedAttention vs naive KV allocation. Naive allocation reserves 5.37 GB per request upfront (87.5% wasted at short sequences). PagedAttention allocates 16-token blocks on demand, reducing fragmentation to <5% and enabling 5–10× more concurrent requests on the same HBM. Shared prefix blocks enable multiple requests to reuse the same system prompt KV without duplication.

6.4 Disaggregated Prefill-Decode Architecture

```
CO-LOCATED INTERFERENCE (monolithic serving):

In co-located prefill+decode:
  Long prefill (2048 tokens, H100, 70B): occupies GPU for ~80ms
  During that 80ms: ALL decode requests on those GPUs STALL

  Measured effect [REPRESENTATIVE]:
    P50 TPOT:  25ms   (when no prefill interference)
    P99 TPOT: 180ms   (dominated by prefill spikes)
    P99/P50 ratio: 7.2× ← primary diagnostic signal

DISAGGREGATED ARCHITECTURE:
  Prefill cluster:  compute-optimized (H100) · large batch prefill
  Decode cluster:   bandwidth-optimized (H200 or MI300X) · stable TPOT
  KV transfer:      RDMA / NVLink-C2C after prefill completes

KV TRANSFER BUDGET (2048-token prompt, INT8 KV, LLaMA-3 70B):
  Transfer size = 2 × 80 × 8 × 128 × 2048 × 1 byte = 2.68 GB
  Over 2× IB NDR (100 GB/s total): 2.68 / 100 = 26.8 ms
  Over NVLink-C2C (GB200, 10 TB/s): 2.68 / 10,000 = 0.27 ms (!!)

WHEN TO DISAGGREGATE:
  P99/P50 TTFT ratio > 5× while queue time P99 < 200ms
  Fleet size > ~100 GPU-equivalents (engineering overhead is real)
  Cost savings: 20–35% per token at scale

NIXL (NVIDIA Inference Transfer Library, 2025):
  Abstracts RDMA / NVLink-C2C / PCIe for KV transfer
  Achieves 85–95% of theoretical link bandwidth in production
```

6.5 TensorRT-LLM: Production Compilation

```
TRT-LLM OPTIMIZATION PIPELINE:

  HuggingFace checkpoint
    → Quantization (FP8/INT4 via modelopt / AMMO)
    → TensorRT compilation:
         Kernel fusion (attention + norm + residual into single kernel)
         CUDA Graph recording (fixed decode step pipeline)
         Layer-by-layer optimization for {GPU, precision, batch, seq_len}
    → Engine file (.plan) — locked to {GPU, precision, max_batch, max_seq}
    → Triton Inference Server integration

PERFORMANCE vs PyTorch EAGER [REPRESENTATIVE]:
  Kernel fusion:      30–50% latency reduction
  CUDA Graphs:        20–40% decode overhead reduction
  FP8 quantization:   ~2× throughput
  Combined:           2–3× total throughput improvement

OPERATIONAL COSTS:
  Build time:         5–30 min per engine (offline, not at serving time)
  Portability:        Engine NOT portable across {GPU, precision, max_batch, max_seq}
  Fleet implication:  Mixed H100/H200 fleet requires separate engine builds per GPU type
```

6.6 SLA Engineering: TTFT, TPOT, and Fleet Sizing

Metric	Definition	Chat SLA	Code Completion	Batch API
TTFT	Time from request to first output token	<200ms	<50ms	<5s
TPOT	Time per output token during decode	<30ms	<20ms	<100ms
E2EL	TTFT + TPOT × (output_tokens − 1)	<10s	<3s	<300s

```
FLEET SIZING FORMULA:

Step 1: Determine peak traffic
  peak_tok/s = peak_rps × avg_output_tokens
  Example: 500 req/s × 200 tok = 100,000 tok/s

Step 2: Per-GPU decode capacity at TPOT SLA
  tok_per_GPU = (1000ms / TPOT_target_ms) × batch_size_at_SLA
  Example: (1000/25) × 32 = 1,280 tok/s per GPU

Step 3: Raw GPU count
  raw_GPUs = peak_tok/s / tok_per_GPU = 100,000 / 1,280 = 78 GPUs

Step 4: Apply utilization target and burst headroom
  total_GPUs = raw_GPUs / target_util × burst_headroom
             = 78 / 0.75 × 1.20 = 125 H100s

Step 5: Validate KV memory constraint
  KV_per_request = 5.37 GB  (70B, 4K ctx, BF16)
  KV_budget_per_GPU = (total_HBM - weights - runtime) = 35 GB
  max_concurrent = 35 / 5.37 = 6  → verify batch_size_at_SLA ≤ 6

PRODUCTION INVARIANTS (never violate):
  GPU utilization target:    ≤ 80%  (headroom for traffic spikes)
  KV cache utilization:      ≤ 80%  (headroom to prevent preemption cascade)
  P99/P50 TTFT alarm:        > 5×   (prefill-decode interference indicator)
```

6.7 Speculative Decoding in Production

```
SPECULATIVE DECODING MECHANISM:
  1. Draft model (7B) generates K=5 candidate tokens quickly (~2ms)
  2. Target model (70B) verifies all 5 in ONE forward pass (same cost as 1 token)
  3. Accept longest matching prefix via rejection sampling

ACCEPTANCE RATE MATH (α = per-token acceptance rate):
  Expected accepted tokens = (1 − α^(K+1)) / (1 − α)
  At α=0.80, K=5: E[tokens] = (1 − 0.8⁶) / 0.2 = (1−0.262)/0.2 = 3.69 tokens
  Effective speedup: 3.69× decode throughput at batch=1

WHEN IT HELPS:
  ✓ Batch=1 or small batch (single user sessions, code generation)
  ✓ Predictable text (code, structured output, templates)
  ✓ Acceptance rate α > 0.70  (monitor in production via metrics)

WHEN IT HURTS:
  ✗ Large batch (draft verification overhead not amortized)
  ✗ Creative / diverse generation (α typically 0.3–0.5)
  ✗ Already bandwidth-bottlenecked at high batch (adds compute cost)
```

6.8 Prefix Caching and KV Reuse

```
PREFIX CACHING ECONOMICS:
  System prompt = 4,096 tokens. No caching at 10,000 req/s:
    Wasted prefill: 4096 × 10,000 = 40.96 billion tokens/s repeated computation
    TTFT overhead per request: ~160ms (2048-token prefill on H100 70B)

  With RadixAttention / prefix caching (vLLM ≥0.4, SGLang):
    System prompt KV computed ONCE → stored in block pool → pointer-shared
    TTFT improvement: 50–80% for system-prompt-heavy workloads

IMPORTANT IMPLEMENTATION DETAIL:
  Cache key = EXACT input token IDs only.
  Temperature ≠ 0 does NOT prevent cache hits (sampling is post-logits).
  Temperature changes output sequences but does NOT change KV computation.
  A request with temperature=0.7 hits the cache identically to temperature=0.
```

6.9 Distributed Inference: TP and PP

Strategy	What	Comm	BW Required	Constraint
Tensor Parallelism (TP)	Split weight matrices col/row	AllReduce per layer	**NVLink mandatory**	NVSwitch domain (≤8 GPUs H100)
Pipeline Parallelism (PP)	Split layers across stages	P2P activations	**IB acceptable**	Pipeline bubble = (PP−1)/(PP+M−1)
Data Parallelism	Replicate model, split requests	None at inference	N/A	Model must fit per replica

> **TP over InfiniBand is unusable for inference.**
>
> TP=8, 70B model, 80 layers, 0.5 GB AllReduce per layer: NVLink (900 GB/s): 0.5/900 × 1,000ms = 0.56ms × 80 = 44ms overhead. IB NDR (50 GB/s): 0.5/50 × 1,000ms = 10ms × 80 = 800ms overhead. At TPOT_min=41.8ms on H100, IB-based TP adds 19× the base TPOT in overhead. TP must stay within the NVSwitch domain.

6.10 Inference Observability — The 5 Metrics That Matter

Metric	Alert Threshold	Root Cause	Action
`vllm:request_queue_time_seconds P99`	**>200ms**	Insufficient GPU capacity	Add GPUs or reduce max_seq
`vllm:gpu_kv_cache_usage_perc`	**>80%**	KV budget exhaustion → preemption	Reduce concurrent requests or quantize KV
`vllm:time_to_first_token P99/P50 ratio`	**>5×**	Prefill-decode interference	Chunked prefill or disaggregation
`DCGM_FI_DEV_GPU_UTIL`	**<70% sustained peak**	Scheduling or admission issue	Tune batch admission, check CUDA Graphs
`vllm:generation_throughput_tokens_per_s`	**<60% theoretical**	Low batch efficiency or preemption	Tune max_batch, prefix caching, KV quant

Chapter 6 Quick Reference

Metric / Formula	Value (LLaMA-3 70B, H100)	Interpretation
TPOT_min (BF16, batch=1)	**41.8 ms/token**	Hard bandwidth floor — no software crosses it.
TPOT_min (INT4, batch=1)	10.4 ms/token	4× improvement from quantization alone.
KV per request (4K ctx, BF16)	5.37 GB	Max ~6 concurrent requests per H100 GPU pair.
Continuous batching gain	10–30× vs static	Largest single inference optimization available.
PagedAttention fragmentation	<5% (vs 87.5% naive)	5–10× more concurrent requests on same HBM.
TRT-LLM vs PyTorch eager	2–3× throughput	Via fusion + CUDA Graphs + FP8.
Spec decoding speedup (α=0.8, K=5)	3.69× at batch=1	Works only for predictable text at small batch.
Prefix cache TTFT improvement	50–80%	For system-prompt-heavy workloads. Zero hardware cost.
TP over NVLink vs IB	44ms vs 800ms overhead	TP must use NVLink — IB-based TP is 18× slower.
Max KV utilization (target)	≤ 80%	Production invariant — above 80% triggers preemption cascade.

Chapter 6 Review Questions

0. **LLaMA-3 70B on H100 SXM5 (TP=2, 35 GB weights per GPU, BF16). Calculate TPOT_min at batch=1. Then calculate the minimum number of H100 pairs to achieve TPOT <25ms at 1,000 concurrent 4096-token requests, using 75% utilization target and 20% burst headroom.**
   Show all steps. Then repeat the calculation with INT8 KV quantization and check whether the KV memory constraint changes the answer.

0. **A vLLM deployment shows: P99 TTFT=1.8s, P50 TTFT=120ms, queue time P99=30ms, KV cache util=62%, GPU util=78%. What is the root cause?**
   The P99/P50 TTFT ratio is 15×. Queue time is low. KV utilization is not critical. GPU utilization is healthy. Walk through your diagnostic reasoning and name the exact fix.

0. **PagedAttention uses block_size=16 tokens. Explain the internal fragmentation vs. metadata overhead tradeoff for block sizes of 8, 16, and 32 tokens. What is the optimal block size for a workload with Poisson-distributed output lengths (mean=200, stddev=150)?**

0. **A disaggregated serving system processes 20 req/s, average prompt 2048 tokens, INT8 KV. Calculate the KV transfer bandwidth required. Is 2× IB NDR (100 GB/s total) sufficient? What is the per-request transfer latency?**
   Now calculate the same for NVLink-C2C (10 TB/s, GB200 NVL72). What new bottleneck emerges in the GB200 case that was negligible with IB?

0. **Speculative decoding: K=5 candidates, α=0.75 acceptance rate. Calculate expected accepted tokens per verify step and effective decode speedup. At what α does speculative decoding break even (no benefit vs standard decode)?**
   At batch=64 with a target verify time of 50ms per step and draft time of 5ms per step for K=5 tokens, is speculative decoding beneficial? Show the calculation.

0. **Design a production inference architecture for LLaMA-3 70B serving 10K req/s, 500-token average input, 300-token average output, P99 TPOT <25ms, P99 TTFT <200ms.**
   Evaluate: (a) Monolithic serving, H100 TP=2, BF16. (b) Disaggregated, H100 prefill + MI300X decode, FP8 weights + INT8 KV. Show all math, state all assumptions, compare 3-year TCO, and recommend one architecture with quantitative justification.

Chapter Cross-References

Ch 4 FlashAttention — §6.1 explains why standard attention is memory-bound. Ch 4 provides the full derivation of FA's HBM traffic reduction (34× per head at S=4096) and shows why FAv3 achieves 87% HBM bandwidth utilization vs 24% for standard attention.

Ch 11 KV cache engineering — §6.3 introduces PagedAttention at the system level. Ch 11 extends this with prefix caching (RadixAttention), KV cache quantization to INT8/FP8, KV offloading strategies, and the complete eviction policy framework.

Ch 13 Speculative decoding — §6.7 covers the acceptance rate math and production guidance. Ch 13 goes deeper: Medusa multi-head prediction, lookahead decoding, and draft model selection criteria with quality-throughput tradeoff quantification.

Ch 17 Observability — the five metrics in §6.10 are configured as Prometheus scrape targets from vLLM and DCGM. Ch 17 covers the full Prometheus/Grafana dashboard design, alert rule construction, and the complete dcgm-exporter metric list.