6. Inference & Compression

The Physics of Generation: From GQA to Disaggregated Serving

Table of contents

0.1 Overview

This chapter is a practical guide to **efficient LLM inference** and **compression**, framed the way modern MLE interviews are framed: *identify the bottleneck (compute vs memory vs network), quantify it, then pick the right system + model levers.*

We'll focus on:

- **Inference physics:** *prefill* vs *decode* (compute-bound vs memory-bound)
- **KV cache:** sizing, fragmentation, paging, quantization
- **System levers:** continuous batching, chunked prefill, prefix caching, speculative decoding, guided decoding
- **Serving architecture:** P/D disaggregation, multi-tenancy, routing
- **Compression:** quantization, pruning/sparsity, distillation, low-rank/adapters
- **Evaluation:** TTFT/TPOT/throughput + quality regression gates

> **Note**
>
> If you only remember one thing: **Prefill scales like GEMM (compute-bound). Decode scales like KV + weight traffic (memory-bound).**

0.2 Learning goals

By the end of this chapter, you should be able to:

- **Analyze the physics:** explain why **prefill** is typically compute-bound and **decode** is typically memory-/bandwidth-bound.
- **Calculate capacity:** estimate KV cache requirements under **MHA vs. MQA vs. GQA** (and how that changes max concurrency).
- **Design the stack:** choose engines (e.g., vLLM vs. TRT-LLM) and scheduling strategies (continuous batching, chunked prefill, prefix caching).
- **Optimize kernels:** explain how **FlashAttention** and **kernel fusion** reduce HBM traffic and launch overhead.
- **Apply compression:** select the right quantization strategy (weight-only vs. activation vs. KV) and predict TTFT/TPOT impacts.
- **Architect for scale:** design **disaggregated serving** and **multi-LoRA** systems for cost efficiency.

ight-only vs. activation vs. KV) and predict TTFT/TPOT impacts. - **Architect for scale:** design **disaggregated serving** and **multi-LoRA** systems for cost efficiency.

---

1 The physics of inference

1.1 Prefill vs decode (the "physics" of generation)

To understand LLM performance, internalize that "generating text" is actually **two different workloads** executed in sequence.

1.1.1 Phase 1: Prefill (the "reading" phase)

Also known as: prompt processing, initialization.

What happens: the model processes the full prompt (length (L_{ ext{prompt}})) in parallel, producing hidden states and building the initial KV cache.

- **Operation:** large **matrix–matrix** multiplies (GEMM) across many prompt tokens.
- **Compute:** high. Attention has a quadratic term in prompt length (roughly (O(L_{ ext{prompt}}^2)) for full attention), and MLP/linear layers are heavy GEMMs.
- **Memory access:** relatively efficient weight reuse: weights are loaded and reused across many tokens in the prompt (and across batch).

Arithmetic intensity: typically **high** → **compute-bound**.

Key latency metric: TTFT (time to first token), dominated by queueing + prefill.

1.1.2 Phase 2: Decode (the "writing" phase)

Also known as: autoregressive token generation.

What happens: the model generates output tokens sequentially. At step (t), it consumes the latest token and previously cached KV to produce the next token.

- **Operation:** effectively **matrix–vector** (GEMV) or small-GEMM at low batch sizes, plus **KV cache reads**.
- **Compute:** much smaller per step than prefill (you're processing ~1 token per sequence).
- **Memory access:** heavy. Each decode step must read:
  - substantial portions of the model weights (dominant at small batch; mitigated at higher batch via reuse), and
  - the growing KV history for attention for each active sequence.

Arithmetic intensity: typically **low** → **memory-bandwidth / scheduling bound**.

Key latency metric: TPOT/ITL (time per output token / inter-token latency), dominated by decode efficiency (KV traffic + kernel/scheduler overhead).

```
gantt
    title Lifecycle of a request (conceptual)
    dateFormat s
    axisFormat %s

    section Request A
    Prefill (compute-bound) :active, p1, 0, 2s
    Decode t=1 (memory-bound) :d1, after p1, 0.5s
```

```
    Decode t=2 :d2, after d1, 0.5s
    Decode t=3 :d3, after d2, 0.5s

    section Request B
    Wait in queue :crit, 0, 1s
    Prefill :p2, after p1, 1s
    Decode t=1 :d4, after p2, 0.5s
```

The roofline implication (why this dichotomy matters)

Feature	Prefill	Decode
Limiting factor	**Compute (FLOPs)**	**Memory bandwidth (GB/s) + KV capacity**
Typical hardware signature	Hot tensor cores	Tensor cores waiting on memory / scheduler
Key metric	**TTFT**	**TPOT / ITL**
Batching effect	More batch → more compute	More batch can be "cheap" until compute catches up to bandwidth
Common optimizations	FlashAttention, tensor parallel, compilation/fusions	Paged KV, KV/weight quantization, speculative decoding, scheduling

Decode batching intuition: when decode is bandwidth-bound, you can often increase the number of active sequences with only a modest TPOT penalty—until compute becomes dominant.

1.1.3 Interview Q&A: TTFT vs ITL

- If **TTFT** is too high: prefill is slow → add compute (faster GPU), improve kernels (FlashAttention), reduce prompt length, enable prefix caching, or shard (TP) for very large models.
- If **ITL/TPOT** is too high: decode is slow → reduce data moved (weight/KV quantization), improve KV management (paged KV), use speculative decoding, and fix scheduling/continuous batching.

1.2 Arithmetic intensity and the "compute vs bandwidth" trap

A back-of-the-envelope way to reason about bottlenecks is **arithmetic intensity**:

1.2.1 TODO: remove equation due to rendering error

- **High** (I) → compute-bound (tensor cores busy)
- **Low** (I) → memory-bound (cores waiting for HBM)

1.2.2 Why prefill tends to be compute-bound

In prefill, weights get reused across many prompt tokens in a batch, boosting (I).

1.2.3 Why decode tends to be memory-bound

In decode, at small batch sizes you do relatively little compute per token but still must read: - weights (unless cached effectively at higher batch), - and a growing KV cache for attention.

> **Note**
>
> A common real-world symptom: *high GPU "utilization" reported, but low tensor core utilization* (the GPU is busy waiting on memory or launching kernels).

---

2 Memory bottlenecks: KV cache

2.1 Attention architecture variants: MHA vs. MQA vs. GQA

You can't reason about KV cache cost without knowing how many **KV heads** your model has.

- **MHA (Multi-Head Attention):** (N_{kv-heads} = N_{q-heads}). Highest KV memory/bandwidth.
- **MQA (Multi-Query Attention):** (N_{kv-heads} = 1). Smallest KV cache, but can reduce quality for some tasks.
- **GQA (Grouped-Query Attention):** (1 < N_{kv-heads} < N_{q-heads}). Common "Goldilocks" choice (used in many modern models).

2.1.1 KV cache scaling rule

Holding everything else fixed, KV cache size (and decode KV bandwidth) scales **linearly** with (N_{kv-heads}). Therefore:

2.1.2 TODO: remove equation due to rendering error

Example: if (N_{q-heads}=64) and (N_{kv-heads}=8), then KV cache is about (64/8 = 8×) smaller than MHA.

> Tip
>
> **Interview move:** If TPOT improves after switching to GQA, say: **less KV read per step → less bandwidth pressure → higher concurrency at the same latency**.

2.2 What the KV cache is

For attention at time (t), the model needs keys/values for **all prior tokens** (1..t). Recomputing is too slow, so we cache K/V per layer.

2.3 The math: estimating KV cache size

A standard interview back-of-the-envelope question:

[KV bytes per token ; ; 2 ×N_{layers} ×N_{kv-heads} ×D_{head} ×P_{bytes}]

Total KV footprint for a sequence length (L) and concurrency (B):

[KV bytes total ; ; B ×L ×KV bytes per token]

Where: - the **2** is for K and V, - (N_{kv-heads}) is **KV heads** (important: with GQA/MQA this can be *much smaller* than attention heads), - (P_{bytes}) is bytes per element (e.g., 2 for FP16/BF16; 1 for FP8/INT8).

> Tip
>
> **Don't forget GQA:** KV cache size depends on **KV heads**, not attention heads.

2.3.1 Worked example (generic, interview-style)

Assume: - (N_{layers}=80), - (N_{kv-heads}=8), - (D_{head}=128), - FP16 → (P_{bytes}=2).

KV bytes/token:

[2 ×80 ×8 ×128 ×2 = 327{,}680;bytes 0.31;MB/token]

At (L=8{,}192), KV per sequence (2.5) GB.

Implication: long context + high concurrency is primarily a *memory capacity* problem.

2.4 PagedAttention: the OS metaphor

Naively allocating a contiguous KV tensor for max length wastes memory (fragmentation). **PagedAttention** treats KV like virtual memory:

1. Divide KV into fixed-size blocks (e.g., 16 tokens per block).
2. Allocate blocks on demand.
3. Keep a "page table" mapping sequence positions to blocks.

```
flowchart LR
  A[Sequence tokens] --> B[KV pages: blocks of 16 tokens]
  B --> C[Non-contiguous allocation]
  C --> D[Lower fragmentation → higher concurrency]
```

Why it matters - Near-zero fragmentation increases effective capacity. - Enables **preemption** and **continuous batching**.

2.5 KV cache quantization

KV cache can be quantized (FP16 → FP8/INT8/INT4) to: - increase max concurrency, - reduce memory bandwidth in decode.

Tradeoff: quality regressions often show up in: - long-context retrieval, - "needle in haystack" style tasks, - tool-use correctness when evidence is mid-context.

---

3 Kernel and attention optimizations

3.1 FlashAttention

FlashAttention improves attention speed by reducing HBM traffic and fusing operations. Use it when attention becomes dominant (long context, high throughput settings).

Practical tips - Validate kernel compatibility: MHA/GQA/MQA, RoPE, sliding window. - Re-check numerics when changing precision (BF16/FP16/FP8).

3.2 Kernel fusion and fused ops

Even when an operation is "small," launching many GPU kernels can be expensive (CPU GPU coordination, scheduling, synchronization). **Kernel fusion** combines multiple steps into fewer launches to reduce overhead and keep data on-chip longer.

Common fusions around attention blocks:

- scale + mask + softmax (+ dropout)
- bias + activation + residual
- fused layernorm, fused rotary embeddings (implementation-dependent)

Why it matters - Prefill: improves throughput by reducing launch overhead and memory traffic. - Decode: reduces per-token overhead where kernels are tiny and launch costs dominate.

> ℹ Note
>
> Kernel fusion complements FlashAttention: **FlashAttention reduces HBM traffic inside attention; fusion reduces overhead around it.**

3.3 Page attention vs flash attention

They solve different problems:

- **FlashAttention:** faster attention compute (bandwidth reduction within attention).
- **PagedAttention:** smarter KV memory management (capacity + scheduling + fragmentation).

Interview pattern: propose both when context is long **and** concurrency is high.

---

4 System optimization: batching & scheduling

4.1 Continuous batching (in-flight batching)

Static batching waits for the batch to finish; continuous batching inserts new requests as slots open.

- Boosts throughput (tokens/sec/GPU).
- Can improve p95/p99 by reducing head-of-line blocking if paired with admission control.

```
flowchart TB
  Q[Request queue] --> S[Scheduler]
  subgraph GPU_Batch
```

```
    A1[Req A active]
    B1[Req B finishes] -->|evict| C1[Req C admitted]
    D1[Req D active]
  end
  S --> C1
```

4.1.1 Admission control (why it matters)

Without admission control, you can over-admit, blow KV capacity, and destroy tail latency.

Common policies: - cap active sequences by KV budget, - prioritize short requests (SRPT-like heuristics), - preempt low-priority or very long decode tails.

4.2 Chunked prefill (solving the convoy effect)

A huge prompt (RAG with tens of thousands of tokens) can "freeze" the batch if prefill is done atomically.

Chunked prefill: 1. Prefill chunk 1 for long request. 2. Decode steps for short requests. 3. Prefill chunk 2, etc.

Benefit: smoother ITL and lower p99.

4.3 Prefix caching (prompt caching)

If many requests share a prefix (system prompt, policy text, long instructions), caching KV for that prefix avoids recomputation.

Practical tips - Normalize prompts for cache hits (templating consistency). - Split stable prefix vs volatile suffix. - Track prefix-cache hit ratio and saved prefill tokens.

4.4 Speculative decoding (trade compute for bandwidth)

Decode is often memory-bound. Speculative decoding uses: - a small **draft model** to propose (K) tokens, - a big **target model** to verify those tokens in one pass.

Win condition: high acceptance rate and draft much cheaper than target.

```
flowchart LR
  A[Prompt + KV] --> B[Draft proposes K tokens]
  B --> C[Target verifies in 1 pass]
```

```
  C -->|accept m<=K| D[Advance by m tokens]
  C -->|reject| E[Fallback decode]
```

4.5 Test-time scaling (reliability without retraining)

Test-time scaling spends compute **during inference** to increase reliability, without changing the model weights.

> Tip
>
> **ELI5:** *Test-time scaling is like thinking twice: generate several attempts, check them, and pick the best.*

4.5.1 Common patterns to mention

- best-of-N + verifier,
- self-consistency voting,
- critique → revise loops,
- search (Tree-of-Thought, MCTS) with pruning.

Spending compute *at inference* to improve reliability.

1. **Best-of-N:** sample N candidates, score with a verifier, take the best.
2. **Sequential revision:** draft → critique → fix.
3. **Search:** Tree-of-Thought / MCTS-style exploration (most useful for agents and tool chains).

```
# Pseudocode: test-time scaling (best-of-N)
cands = model.generate(prompt, n=16, temp=0.7)
scores = verifier.score(prompt, cands)
best_response = cands[argmax(scores)]
```

> Tip
>
> **Interview framing**
> Test-time scaling is an "inference lever" when retraining is expensive or slow. The tradeoff is latency/cost.

4.6 Guided decoding and constrained generation

Constrained decoding enforces: - JSON schema correctness, - tool argument validity, - grammar constraints.

Tradeoffs: - constraint checking overhead, - can reduce diversity (sometimes desirable for tools).

---

5 Production patterns: disaggregated serving

5.1 Why disaggregate prefill and decode?

Prefill and decode want different "hardware personalities":

- Prefill: compute-heavy (benefits from high tensor-core throughput).
- Decode: bandwidth + memory heavy (KV traffic; long tails).

Colocating both can create interference and tail-latency spikes.

5.2 Prefill/Decode (P/D) split

Pattern 1. Prefill fleet runs prompts, builds KV. 2. Transfer KV (and state) over fast interconnect. 3. Decode fleet continues autoregressive generation.

```
flowchart LR
  U[User request] --> P[Prefill workers]
  P -->|KV + state| X[Transfer]
  X --> D[Decode workers]
  D --> U2[Stream tokens]
```

Design questions to cover - KV transfer cost (bytes = KV size): when is it worth it? - network fabric (NVLink / InfiniBand / TCP): what limits you? - failure handling: retries, partial streams, idempotency - observability: TTFT split across fleets, queueing per tier

---

5.3 Multi-LoRA serving (the "Bento" pattern)

Serving many fine-tuned variants (per customer, per feature, per locale) naively requires one GPU (or replica set) per model. **Multi-LoRA serving** keeps a single frozen base model resident and dynamically applies lightweight adapter deltas per request.

5.3.1 Mental model

- **Base weights:** shared, always loaded
- **Adapters (LoRA):** small, swappable deltas (often <1–2% of base params)
- **Scheduler:** groups requests by (base, adapter) to batch efficiently

```
flowchart LR
  R[Requests w/ adapter_id] --> S[Router/Scheduler]
  S -->|batch by adapter| G1[GPU batch: adapter A]
  S -->|batch by adapter| G2[GPU batch: adapter B]
```

```
  G1 --> O[Responses]
  G2 --> O
```

5.3.2 Practical engineering points (interview-grade)

- **Batching constraint:** mixing many adapters in the same decode batch can add overhead; most systems batch by adapter_id.
- **Caching:** prefix caching can be per-(base, adapter) depending on implementation.
- **Hot set vs cold set:** keep popular adapters in GPU memory; page less-used adapters (or load on demand).
- **Isolation:** ensure correct adapter routing to avoid "tenant bleed."

```
# Pseudocode: adapter-aware batching (conceptual)
while True:
    reqs = dequeue_ready()
    groups = groupby(reqs, key=lambda r: r.adapter_id)
    for adapter_id, batch in groups.items():
        activate_adapter(adapter_id)         # swap/merge/apply LoRA
        run_decode_or_prefill(batch)
```

> Tip
>
> **Interview one-liner:** "Multi-LoRA turns N fine-tuned models into **one shared base + N small deltas**, maximizing GPU utilization and lowering cost-per-request."

6 Compression: shrinking the model

6.1 Quantization

6.1.1 Taxonomy

Type	Target	What changes	Best for
Weight-only (INT8/INT4)	Model size + bandwidth	store weights low-bit; dequantize for compute	memory-bound decode, edge/CPU
Activation quant (INT8/FP8)	Compute throughput	matmuls in lower precision	compute-bound prefill, large batches
KV cache quant	Memory capacity + bandwidth	K/V stored low precision	long context, high concurrency

6.1.2 The outlier problem (why naive quant fails)

LLMs have outlier channels / activation spikes. Naive quantization clips them and can crater quality.

Mitigation strategies to mention: - outlier-aware weight quant (e.g., AWQ-style), - activation smoothing (SmoothQuant-style), - selective higher precision for outlier blocks.

Practical tips - Evaluate long-context retrieval and tool-call correctness after quant. - Re-tune decoding params if distribution shifts. - Calibrate on production-like prompts (length, language, tools).

6.2 Pruning and sparsity

6.2.1 Types

- **Unstructured pruning:** hard to accelerate without specialized kernels.
- **Structured pruning:** block/N:M sparsity can yield real speedups on supported hardware.
- **Architectural sparsity:** MoE routing is "sparsity by design."

Interview hooks - why structured sparsity is preferred for real latency wins, - why pruning can introduce non-linear "quality cliffs."

6.3 Knowledge distillation

6.3.1 Forms

- **Response distillation:** student learns teacher outputs (SFT on traces).
- **Logit distillation:** KL to teacher logits (needs teacher access).
- **On-policy distillation:** student samples, teacher guides (reduces distillation distribution mismatch).

6.3.2 When it wins

- cheaper model at similar behavior,
- stabilize post-RL policies,
- compress tool-use / reasoning traces into smaller students.

6.4 Low-rank factorization and adapters

- low-rank factorization of weights,
- adapter-based PEFT (LoRA/DoRA-style),
- multi-adapter serving and routing considerations.

7 Framework landscape

TODO: RL. vllm, sglang, triton server

7.1 Training framework (where the checkpoint comes from)

Topics to include: - FSDP/ZeRO tradeoffs, activation checkpointing - tensor/pipeline parallelism, microbatching - mixed precision and numerics checks - how training-time decisions affect inference (e.g., GQA/MQA, context length)

7.2 Inference framework (where tokens come from)

What to highlight in interviews: - KV paging + continuous batching support - prefix caching + chunked prefill support - speculative decoding integration - quantization support (weights and KV) - guided decoding support for tools

---

8 Evaluation & metrics

8.1 Core metrics

1. **TTFT**: time to first token (queue + prefill)
2. **TPOT / ITL**: time per output token / inter-token latency (decode efficiency)
3. **Throughput**: tokens/sec/GPU (utilization)
4. **Cost**: $/1M tokens (hardware + ops)

8.2 Quality regression

- **Perplexity (PPL):** good for sanity checks across quantization/caching changes (within same tokenizer/eval setup).
- **Needle-in-a-haystack variants:** stress long-context retention (especially after KV quantization).
- **Tool-use correctness:** JSON validity + end-to-end tool success rate.
- **Prompt continuation similarity:** ROUGE-L / BLEU / BERTScore (use cautiously).

---

9 Capstone: inference decision matrix

Constraint	Strategy	Why
Latency sensitive (chat)	continuous batching + speculative decoding	reduce TPOT while keeping utilization
Throughput sensitive (batch jobs)	large batch + activation quant/FP8	saturate compute, amortize overhead
Long context (RAG/docs)	**GQA/MQA** + paged KV + chunked prefill + KV quant	reduce KV footprint + prevent OOM + reduce head-of-line
Massive scale (>10k r/s)	P/D disaggregation	scale compute (prefill) vs bandwidth (decode) independently

---

10 Appendix: interview drills

10.1 Drill 1: batch size vs latency

Q: Why does increasing batch size improve throughput but hurt latency?

A (excellent): - Throughput improves because you amortize weight loads and kernel launch overhead across more tokens/requests (higher arithmetic intensity). - Latency can worsen because each request waits for larger batch prefill/step completion, and queueing increases if you chase max utilization.

10.2 Drill 2: OOM on long prompts

Q: Your model is OOM'ing on long prompts. What do you do?

A (excellent): 1. Paged KV / block allocation to reduce fragmentation. 2. KV quantization (FP16 $\rightarrow$ FP8 halves KV bytes). 3. Admission control (cap active sequences by KV budget). 4. If a single request exceeds memory: tensor parallel / offload / disaggregation.

10.3 Drill 3: "why is decode slow?"

Q: Decode TPOT got worse after a change. What do you check?

A (excellent): - KV cache size and precision (did L or concurrency grow? did KV quant disable?) - batching/scheduler (are we at small batch? poor packing? preemption?) - kernel path (did attention kernel disable? did GQA/MQA mismatch?) - sampling overhead (guided decoding constraints, tool validators)

10.4 Drill 4: GQA vs MHA (KV cache impact)

Q: How does **Grouped-Query Attention (GQA)** improve inference vs standard **MHA**?

A (excellent): - GQA reduces the number of **KV heads** (N_{kv}) relative to query heads (N_q), so KV cache size scales down by roughly (N_q/N_{kv}). - That cuts **decode bandwidth** (less KV to read per step) and increases max concurrency before OOM. - It's a "Goldilocks" tradeoff: smaller KV than MHA, usually better quality than MQA.