Transformer Inference Cost and Measurement Foundations

Deep Dive into DeepSeek — Chapter 2

Yin Yang

Foundation Books Project

The one rule of this chapter

A cost model starts from a request shape, not from the model's total parameter count.

- Prompt tokens create prefill work and retained attention state; generated tokens repeatedly read that state and activate current-token feed-forward work.

The one rule of this chapter

A cost model starts from a request shape, not from the model's total parameter count.

- Prompt tokens create prefill work and retained attention state; generated tokens repeatedly read that state and activate current-token feed-forward work.
- Every reported count must keep its tensor extent, layer scope, byte convention, and workload boundary attached — dimensional consistency is the chapter's main invariant.

The one rule of this chapter

A cost model starts from a request shape, not from the model's total parameter count.

- Prompt tokens create prefill work and retained attention state; generated tokens repeatedly read that state and activate current-token feed-forward work.
- Every reported count must keep its tensor extent, layer scope, byte convention, and workload boundary attached — dimensional consistency is the chapter's main invariant.
- A two-request miniature makes every formula hand-checkable, before the same ledgers attach to DeepSeek-V3's released dimensions.

The one rule of this chapter

A cost model starts from a request shape, not from the model's total parameter count.

- Prompt tokens create prefill work and retained attention state; generated tokens repeatedly read that state and activate current-token feed-forward work.
- Every reported count must keep its tensor extent, layer scope, byte convention, and workload boundary attached — dimensional consistency is the chapter's main invariant.
- A two-request miniature makes every formula hand-checkable, before the same ledgers attach to DeepSeek-V3's released dimensions.

Goal: build separate ledgers for prefill, decode, KV-cache storage, expert work, and batching — then show why none of them, alone, is a latency measurement.

Roadmap

Cost model foundations

Pricing attention, cache, and feed-forward work

Batching, the V3 profile, and evaluation

Cost model foundations

A cost model needs a workload shape

Definition (Cost model)
A stated set of quantities, formulas, and assumptions used to estimate or compare arithmetic, memory traffic, stored attention state, communication, latency, or throughput.

A cost model needs a workload shape

Definition (Cost model)
A stated set of quantities, formulas, and assumptions used to estimate or compare arithmetic, memory traffic, stored attention state, communication, latency, or throughput.

Definition (Workload shape)
The request and model dimensions that determine computation before hardware details: prompt and output lengths, serving policy, hidden width, layers, head layout, cache representation, expert dimensions.

A cost model needs a workload shape

Definition (Cost model)
A stated set of quantities, formulas, and assumptions used to estimate or compare arithmetic, memory traffic, stored attention state, communication, latency, or throughput.

Definition (Workload shape)
The request and model dimensions that determine computation before hardware details: prompt and output lengths, serving policy, hidden width, layers, head layout, cache representation, expert dimensions.

DeepSeek-V3: 671B total parameters, 37B activated per token, 61 layers, width 7168, a 512-wide latent plus 64 RoPE coordinates, and 8 of 256 routed experts selected per token.

[REPORTED: DeepSeek-V3 repo, inference config, technical report]

Prefill and decode are different shapes

Request	Prompt	Read 1	Read 2	Score positions
A	6	6	7	$7 + 8 = 15$

Prefill and decode are different shapes

Request	Prompt	Read 1	Read 2	Score positions
A	6	6	7	$7 + 8 = 15$
B	3	3	4	$4 + 5 = 9$
Total	9	9	11	24

Prefill and decode are different shapes

Request	Prompt	Read 1	Read 2	Score positions
A	6	6	7	$7 + 8 = 15$
B	3	3	4	$4 + 5 = 9$
Total	9	9	11	24

- Prefill processes $S_A + S_B = 6 + 3 = 9$ prompt positions and builds the initial per-layer attention state.

Prefill and decode are different shapes

Request	Prompt	Read 1	Read 2	Score positions
A	6	6	7	$7 + 8 = 15$
B	3	3	4	$4 + 5 = 9$
Total	9	9	11	24

- Prefill processes $S_A + S_B = 6 + 3 = 9$ prompt positions and builds the initial per-layer attention state.
- Decode step t reads $S_i + t - 1$ prior positions per request; the four generated tokens do not share one attention shape.

Prefill and decode are different shapes

Request	Prompt	Read 1	Read 2	Score positions
A	6	6	7	$7 + 8 = 15$
B	3	3	4	$4 + 5 = 9$
Total	9	9	11	24

- Prefill processes $S_A + S_B = 6 + 3 = 9$ prompt positions and builds the initial per-layer attention state.
- Decode step t reads $S_i + t - 1$ prior positions per request; the four generated tokens do not share one attention shape.

Time to first token counts prefill plus queueing; inter-token latency counts the gap between decode steps. Every claim must say which one it is counting.

FLOPs, bytes, and the roofline floor

$$T_{\text{step}} \;\geq\; \max\left(\frac{F}{P_{\text{eff}}},\; \frac{Q_{\text{mem}}}{B_{\text{eff}}}\right)$$

T-sub-step is at least the larger of: work F over attainable arithmetic rate P-sub-eff, and moved bytes Q-sub-mem over attainable bandwidth B-sub-eff.

- Operational intensity $I = F/Q_{\text{mem}}$ (multiply-add pairs per byte) says which side of the hardware balance a workload sits on.

FLOPs, bytes, and the roofline floor

$$T_{\text{step}} \;\geq\; \max\left(\frac{F}{P_{\text{eff}}},\; \frac{Q_{\text{mem}}}{B_{\text{eff}}}\right)$$

T-sub-step is at least the larger of: work F over attainable arithmetic rate P-sub-eff, and moved bytes Q-sub-mem over attainable bandwidth B-sub-eff.

- Operational intensity $I = F/Q_{\text{mem}}$ (multiply-add pairs per byte) says which side of the hardware balance a workload sits on.
- Worked check: 12 GFLOPs over 3 GB gives $I = 4$; on a 120 TFLOP/s, 2 TB/s accelerator the arithmetic floor is 0.1 ms and the traffic floor is 1.5 ms — bandwidth-bound.

FLOPs, bytes, and the roofline floor

$$T_{\text{step}} \;\geq\; \max\left(\frac{F}{P_{\text{eff}}},\; \frac{Q_{\text{mem}}}{B_{\text{eff}}}\right)$$

T-sub-step is at least the larger of: work F over attainable arithmetic rate P-sub-eff, and moved bytes Q-sub-mem over attainable bandwidth B-sub-eff.

- Operational intensity $I = F/Q_{\text{mem}}$ (multiply-add pairs per byte) says which side of the hardware balance a workload sits on.
- Worked check: 12 GFLOPs over 3 GB gives $I = 4$; on a 120 TFLOP/s, 2 TB/s accelerator the arithmetic floor is 0.1 ms and the traffic floor is 1.5 ms — bandwidth-bound.

The floor identifies pressure, not the observed 1.5 ms: launch, synchronization, and imperfect overlap still push a real kernel above it.

Pricing attention, cache, and feed-forward work

Full-context attention: square, triangular, decode

Score terms per layer, $H_Q d_h$ per comparison:

$$\underbrace{H_Q S^2 d_h}_{\text{square}}$$

Full-context attention: square, triangular, decode

Score terms per layer, $H_Q d_h$ per comparison:

$$\underbrace{H_Q S^2 d_h}_{\text{square}} \qquad \underbrace{H_Q \frac{S(S+1)}{2} d_h}_{\text{causal triangle}}$$

Full-context attention: square, triangular, decode

Score terms per layer, $H_Q d_h$ per comparison:

$$\underbrace{H_Q S^2 d_h}_{\text{square}} \qquad \underbrace{H_Q \frac{S(S+1)}{2} d_h}_{\text{causal triangle}} \qquad \underbrace{H_Q d_h (S_i + t)}_{\text{one decode token}}$$

Full-context attention: square, triangular, decode

Score terms per layer, $H_Q d_h$ per comparison:

$$\underbrace{H_Q S^2 d_h}_{\text{square}} \qquad \underbrace{H_Q \frac{S(S+1)}{2} d_h}_{\text{causal triangle}} \qquad \underbrace{H_Q d_h (S_i + t)}_{\text{one decode token}}$$

- Running example ($H_Q = 2, d_h = 4, S_A = 6, S_B = 3$): square $= 360$, triangular $= 216$, two-step decode total $= 192$.

Full-context attention: square, triangular, decode

Score terms per layer, $H_Q d_h$ per comparison:

$$\underbrace{H_Q S^2 d_h}_{\text{square}} \qquad \underbrace{H_Q \frac{S(S+1)}{2} d_h}_{\text{causal triangle}} \qquad \underbrace{H_Q d_h (S_i + t)}_{\text{one decode token}}$$

- Running example ($H_Q = 2, d_h = 4, S_A = 6, S_B = 3$): square $= 360$, triangular $= 216$, two-step decode total $= 192$.
- Decode is linear in current context length per new token — but that length keeps growing with the request.

Full-context attention: square, triangular, decode

Score terms per layer, $H_Q d_h$ per comparison:

$$\underbrace{H_Q S^2 d_h}_{\text{square}} \qquad \underbrace{H_Q \frac{S(S+1)}{2} d_h}_{\text{causal triangle}} \qquad \underbrace{H_Q d_h (S_i + t)}_{\text{one decode token}}$$

- Running example ($H_Q = 2, d_h = 4, S_A = 6, S_B = 3$): square $= 360$, triangular $= 216$, two-step decode total $= 192$.
- Decode is linear in current context length per new token — but that length keeps growing with the request.

Square storage, causal visibility, and realized kernel work are three different counts. Silently swapping between them creates a factor-of-two illusion at large S.

The KV cache is a cost object, not a detail

$$C_{\text{cache}} = L\Big(\sum_i S_i\Big) H_{\text{KV}}(d_k + d_v)\, b_{\text{elt}}$$

C-sub-cache equals layers, times total stored positions, times KV heads, times the sum of d_k and d_v, times bytes per element.

Toy representation	Bytes after prefill	After two decode steps
MHA, $H_{\text{KV}} = 2$	576	832

The KV cache is a cost object, not a detail

$$C_{\text{cache}} = L\Big(\sum_i S_i\Big) H_{\text{KV}}(d_k + d_v)\, b_{\text{elt}}$$

C-sub-cache equals layers, times total stored positions, times KV heads, times the sum of d_k and d_v, times bytes per element.

Toy representation	Bytes after prefill	After two decode steps
MHA, $H_{\text{KV}} = 2$	576	832
MQA-style, $H_{\text{KV}} = 1$	288	416

The KV cache is a cost object, not a detail

$$C_{\text{cache}} = L\Big(\sum_i S_i\Big) H_{\text{KV}}(d_k + d_v)\, b_{\text{elt}}$$

C-sub-cache equals layers, times total stored positions, times KV heads, times the sum of d_k and d_v, times bytes per element.

Toy representation	Bytes after prefill	After two decode steps
MHA, $H_{\text{KV}} = 2$	576	832
MQA-style, $H_{\text{KV}} = 1$	288	416
Simplified MLA latent, $d_c = 3$	108	156

The KV cache is a cost object, not a detail

$$C_{\text{cache}} = L\Big(\sum_i S_i\Big) H_{\text{KV}}(d_k + d_v)\, b_{\text{elt}}$$

C-sub-cache equals layers, times total stored positions, times KV heads, times the sum of d_k and d_v, times bytes per element.

Toy representation	Bytes after prefill	After two decode steps
MHA, $H_{\text{KV}} = 2$	576	832
MQA-style, $H_{\text{KV}} = 1$	288	416
Simplified MLA latent, $d_c = 3$	108	156

One invariant produces all six numbers: layer count times stored positions times representation width times scalar width.

The V3 scalar-count ratio

- Materialized dense K/V: 128 heads $\times(192 + 128) = 40{,}960$ scalars per token per layer.

The V3 scalar-count ratio

- Materialized dense K/V: 128 heads $\times(192 + 128) = 40{,}960$ scalars per token per layer.
- Absorbed MLA path: $512 + 64 = 576$ scalars per token per layer.

The V3 scalar-count ratio

- Materialized dense K/V: 128 heads $\times(192 + 128) = 40{,}960$ scalars per token per layer.
- Absorbed MLA path: $512 + 64 = 576$ scalars per token per layer.
- Ratio: $40{,}960/576 \approx 71.1$. [REPORTED: DeepSeek-V3 inference config]

The V3 scalar-count ratio

- Materialized dense K/V: 128 heads $\times(192 + 128) = 40{,}960$ scalars per token per layer.
- Absorbed MLA path: $512 + 64 = 576$ scalars per token per layer.
- Ratio: $40{,}960/576 \approx 71.1$. [REPORTED: DeepSeek-V3 inference config]

At 128K stored positions, 61 layers, BF16: ≈ 610 GiB materialized versus ≈ 8.58 GiB for V3 MLA. [DERIVED: cache-growth ledger]

The V3 scalar-count ratio

- Materialized dense K/V: 128 heads $\times(192 + 128) = 40{,}960$ scalars per token per layer.
- Absorbed MLA path: $512 + 64 = 576$ scalars per token per layer.
- Ratio: $40{,}960/576 \approx 71.1$. [REPORTED: DeepSeek-V3 inference config]

At 128K stored positions, 61 layers, BF16: ≈ 610 GiB materialized versus ≈ 8.58 GiB for V3 MLA. [DERIVED: cache-growth ledger]

The $71.1\times$ gap describes stored representation, not speed: dense decode still reads every prior position either way.

From concept to code: two cache buffers, not one

from **DeepSeek-V3/inference/model.py**

```
if attn_impl == "naive":
    self.register_buffer("k_cache", torch.zeros(..., self.qk_head_dim), ...)
    self.register_buffer("v_cache", torch.zeros(..., self.v_head_dim), ...)
else:
    self.register_buffer("kv_cache", torch.zeros(..., self.kv_lora_rank), ...)
    self.register_buffer("pe_cache", torch.zeros(..., self.qk_rope_head_dim), ...)
```

From concept to code: two cache buffers, not one

from **DeepSeek-V3/inference/model.py**

```
if attn_impl == "naive":
    self.register_buffer("k_cache", torch.zeros(..., self.qk_head_dim), ...)
    self.register_buffer("v_cache", torch.zeros(..., self.v_head_dim), ...)
else:
    self.register_buffer("kv_cache", torch.zeros(..., self.kv_lora_rank), ...)
    self.register_buffer("pe_cache", torch.zeros(..., self.qk_rope_head_dim), ...)
```

kv_cache is width kv_lora_rank= 512; pe_cache is width qk_rope_head_dim= 64. Neither trailing dimension mentions the head count — the entire 71.1× claim, written as a tensor shape. [INSPECTED: DeepSeek-V3 inference/model.py]

Dense MLP cost scales with hidden width squared

$$\underbrace{2d(rd) = 2rd^2}_{\text{two-projection}}$$

Two times d times r-d, or three times d times r-d — r-d is the intermediate width.

Dense MLP cost scales with hidden width squared

$$\underbrace{2d(rd) = 2rd^2}_{\text{two-projection}} \qquad \underbrace{3d(rd) = 3rd^2}_{\text{gated, three-projection}}$$

Two times d times r-d, or three times d times r-d — r-d is the intermediate width.

Dense MLP cost scales with hidden width squared

$$\underbrace{2d(rd) = 2rd^2}_{\text{two-projection}} \qquad \underbrace{3d(rd) = 3rd^2}_{\text{gated, three-projection}}$$

Two times d times r-d, or three times d times r-d — r-d is the intermediate width.

For $d = 8, r = 4$: two-projection $= 512$, three-projection $= 768$ multiply-add pairs per token per layer.

Dense MLP cost scales with hidden width squared

$$\underbrace{2d(rd) = 2rd^2}_{\text{two-projection}} \qquad \underbrace{3d(rd) = 3rd^2}_{\text{gated, three-projection}}$$

Two times d times r-d, or three times d times r-d — r-d is the intermediate width.

For $d = 8, r = 4$: two-projection $= 512$, three-projection $= 768$ multiply-add pairs per token per layer.

A dense feed-forward layer spends its full parameter budget on every token. Sparse activation can only reduce per-token arithmetic by changing *which* weights are used.

Total parameters versus active parameters

- DeepSeek-V3: 671B total, 37B activated per token (main-model scope); the distributed weight package adds 14B of MTP weights for a 685B artifact total. [REPORTED: DeepSeek-V3 repo]

Total parameters versus active parameters

- DeepSeek-V3: 671B total, 37B activated per token (main-model scope); the distributed weight package adds 14B of MTP weights for a 685B artifact total. [REPORTED: DeepSeek-V3 repo]
- 256 routed experts, top-8 activation, 1 shared expert.

Total parameters versus active parameters

- DeepSeek-V3: 671B total, 37B activated per token (main-model scope); the distributed weight package adds 14B of MTP weights for a 685B artifact total. [REPORTED: DeepSeek-V3 repo]
- 256 routed experts, top-8 activation, 1 shared expert.
- Expert-only stored-to-active matrix ratio: $257/9 \approx 28.6$ — a different ratio from $671/37 \approx 18.1$, because the whole-model ratio also carries attention, embeddings, and dense layers.

Total parameters versus active parameters

- DeepSeek-V3: 671B total, 37B activated per token (main-model scope); the distributed weight package adds 14B of MTP weights for a 685B artifact total. [REPORTED: DeepSeek-V3 repo]
- 256 routed experts, top-8 activation, 1 shared expert.
- Expert-only stored-to-active matrix ratio: $257/9 \approx 28.6$ — a different ratio from $671/37 \approx 18.1$, because the whole-model ratio also carries attention, embeddings, and dense layers.

"671B parameters are used per token" is a misreading. A much larger set of weights *exists* than the subset any one token's forward path touches.

Same weights, two roofline regimes

- V3 expert: $3 \cdot 7168 \cdot 2048 = 44{,}040{,}192$ weight scalars, one FP8 byte each ≈ 42 MiB.

Same weights, two roofline regimes

- V3 expert: $3 \cdot 7168 \cdot 2048 = 44{,}040{,}192$ weight scalars, one FP8 byte each ≈ 42 MiB.
- Balanced $S = 8192$ prefill, top-8 of 256: $m = 256$ rows per expert bucket $\rightarrow$ floor $22.55\,\mu$s, *arithmetic*-limited.

Same weights, two roofline regimes

- V3 expert: $3 \cdot 7168 \cdot 2048 = 44{,}040{,}192$ weight scalars, one FP8 byte each ≈ 42 MiB.
- Balanced $S = 8192$ prefill, top-8 of 256: $m = 256$ rows per expert bucket $\rightarrow$ floor $22.55\,\mu$s, *arithmetic*-limited.
- Single-request decode: $m = 1$ row $\rightarrow$ floor $14.68\,\mu$s, the *same* weights now *traffic*-limited.

Same weights, two roofline regimes

- V3 expert: $3 \cdot 7168 \cdot 2048 = 44{,}040{,}192$ weight scalars, one FP8 byte each ≈ 42 MiB.
- Balanced $S = 8192$ prefill, top-8 of 256: $m = 256$ rows per expert bucket $\rightarrow$ floor $22.55\,\mu$s, *arithmetic*-limited.
- Single-request decode: $m = 1$ row $\rightarrow$ floor $14.68\,\mu$s, the *same* weights now *traffic*-limited.

Weight reuse, not weight identity, decides which side of the roofline a bucket lands on — a boundary calculation, not a V3 latency measurement.

Batching, the V3 profile, and evaluation

Ragged batches and the throughput/latency tradeoff

- Request A has 6 prompt tokens, B has 3: padding both to length 6 wastes $\Delta_{\text{tokens}} = 3$ token states and $\Delta_{\text{scores}} = 27$ score entries.

Ragged batches and the throughput/latency tradeoff

- Request A has 6 prompt tokens, B has 3: padding both to length 6 wastes $\Delta_{\text{tokens}} = 3$ token states and $\Delta_{\text{scores}} = 27$ score entries.
- Continuous batching changes the active request multiset across decode steps; a new arrival can share a step with ongoing decodes.

Ragged batches and the throughput/latency tradeoff

- Request A has 6 prompt tokens, B has 3: padding both to length 6 wastes $\Delta_{\text{tokens}} = 3$ token states and $\Delta_{\text{scores}} = 27$ score entries.
- Continuous batching changes the active request multiset across decode steps; a new arrival can share a step with ongoing decodes.
- Waiting to batch two prompts can raise prompt tokens/second while delaying the shorter request's time to first token.

Ragged batches and the throughput/latency tradeoff

- Request A has 6 prompt tokens, B has 3: padding both to length 6 wastes $\Delta_{\text{tokens}} = 3$ token states and $\Delta_{\text{scores}} = 27$ score entries.
- Continuous batching changes the active request multiset across decode steps; a new arrival can share a step with ongoing decodes.
- Waiting to batch two prompts can raise prompt tokens/second while delaying the shorter request's time to first token.

Throughput and latency are not opposites, and neither is interpretable without its denominator: tokens, requests, seconds, bytes, device time, or dollars.

The combined V3 resource profile

Surface	Mechanism effect	Remaining pressure
Decode attention	MLA cuts the cache coefficient $\approx 71.1\times$	every step still reads a growing prefix

The combined V3 resource profile

Surface	Mechanism effect	Remaining pressure
Decode attention	MLA cuts the cache coefficient $\approx 71.1\times$	every step still reads a growing prefix
Prefill experts	top-8 routing reuses selected matrices	routing balance sets bucket size

The combined V3 resource profile

Surface	Mechanism effect	Remaining pressure
Decode attention	MLA cuts the cache coefficient $\approx 71.1\times$	every step still reads a growing prefix
Prefill experts	top-8 routing reuses selected matrices	routing balance sets bucket size
Decode experts	only selected + shared experts run	small buckets, little weight reuse

The combined V3 resource profile

Surface	Mechanism effect	Remaining pressure
Decode attention	MLA cuts the cache coefficient $\approx 71.1\times$	every step still reads a growing prefix
Prefill experts	top-8 routing reuses selected matrices	routing balance sets bucket size
Decode experts	only selected + shared experts run	small buckets, little weight reuse

MLA lowers stored-state coefficient; MoE lowers the fraction of expert weights any one token uses. Neither fixes production latency without runtime, hardware, and scheduler.

Evaluation is a protocol, not a number

$$(D, \mathrm{split}, \tau, m, a, S, A, R, \rho)$$

Dataset, split, prompt template, model, sampling policy, scorer, aggregation rule, result schema, provenance.

- Pass@1 $= \frac{1}{n}\sum z_i$; without-replacement pass@$k = 1 - \binom{n-c}{k} / \binom{n}{k}$.

Evaluation is a protocol, not a number

$$(D, \text{split}, \tau, m, a, S, A, R, \rho)$$

Dataset, split, prompt template, model, sampling policy, scorer, aggregation rule, result schema, provenance.

- Pass@1 $= \frac{1}{n}\sum z_i$; without-replacement pass@$k = 1 - \binom{n-c}{k} / \binom{n}{k}$.
- Worked check: $n = 20, c = 5$ gives pass@1$= 0.25$, pass@4≈ 0.7183 — not comparable without matching sampling policy and denominator.

Evaluation is a protocol, not a number

$$(D, \mathrm{split}, \tau, m, a, S, A, R, \rho)$$

Dataset, split, prompt template, model, sampling policy, scorer, aggregation rule, result schema, provenance.

- Pass@1 $= \frac{1}{n}\sum z_i$; without-replacement pass@$k = 1 - \binom{n-c}{k} / \binom{n}{k}$.
- Worked check: $n = 20, c = 5$ gives pass@1$= 0.25$, pass@4≈ 0.7183 — not comparable without matching sampling policy and denominator.

A benchmark score, timing, or verdict is a protocol outcome. The same discipline that priced FLOPs and bytes now prices evaluation claims.

Concept-to-code map for the chapter

- **two cache buffers: latent + RoPE** → `DeepSeek-V3/inference/model.py`

Concept-to-code map for the chapter

- **two cache buffers: latent + RoPE** → `DeepSeek-V3/inference/model.py`
- **released MLA, routing, and layer widths** → `DeepSeek-V3/inference/configs/config_671B.json`

Concept-to-code map for the chapter

- **two cache buffers: latent + RoPE** → `DeepSeek-V3/inference/model.py`
- **released MLA, routing, and layer widths** → `DeepSeek-V3/inference/configs/config_671B.json`
- **routed/shared expert dispatch** → `DeepSeek-V3/inference/model.py`

Concept-to-code map for the chapter

- **two cache buffers: latent + RoPE** → `DeepSeek-V3/inference/model.py`
- **released MLA, routing, and layer widths** → `DeepSeek-V3/inference/configs/config_671B.json`
- **routed/shared expert dispatch** → `DeepSeek-V3/inference/model.py`
- **hand-checkable KV-cache ledger** → `code/chapter_inference_cost_model/kv_cache_ledger_checkpoint.py`

Concept-to-code map for the chapter

- **two cache buffers: latent + RoPE** → `DeepSeek-V3/inference/model.py`
- **released MLA, routing, and layer widths** → `DeepSeek-V3/inference/configs/config_671B.json`
- **routed/shared expert dispatch** → `DeepSeek-V3/inference/model.py`
- **hand-checkable KV-cache ledger** → `code/chapter_inference_cost_model/kv_cache_ledger_checkpoint.py`

Name the workload shape, keep every ledger's units attached, then measure.

Next: Chapter 3 studies DeepSeekMoE's routing and balancing in depth; Chapter 4 studies MLA's cache representation in depth — both chapters modify the ledgers built here.