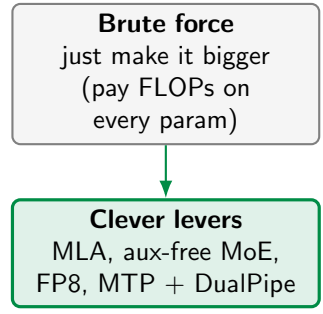
A frontier model for the price of a house

A **671B-parameter MoE** (only **37B active** per token) that stands on par with **GPT-4o** and **Claude-3.5-Sonnet** – yet its full training cost about **$5.6M** (**2.79M H800 GPU-hours**). An order of magnitude cheaper than you would expect for this tier.

And it is the base model that **DeepSeek-R1**'s reasoning (**[LLM-11]**) was trained on top of.

The trick is *not* brute scale. It is a stack of **efficiency levers** – **two carried over** from DeepSeek-V2 and re-validated, **three new** in V3.



frontier quality, a fraction of the bill

Outline

Setup: total vs active, inherited vs new

MLA: shrink the KV cache (from V2)

DeepSeekMoE + auxiliary-loss-free balancing

FP8 training at large scale

Multi-Token Prediction (MTP)

Does it work? And the R1 link

Recap

Total vs active, inherited vs new

Name the pieces first – and be honest about which ones V3 actually invented.

671B total, 37B active per token

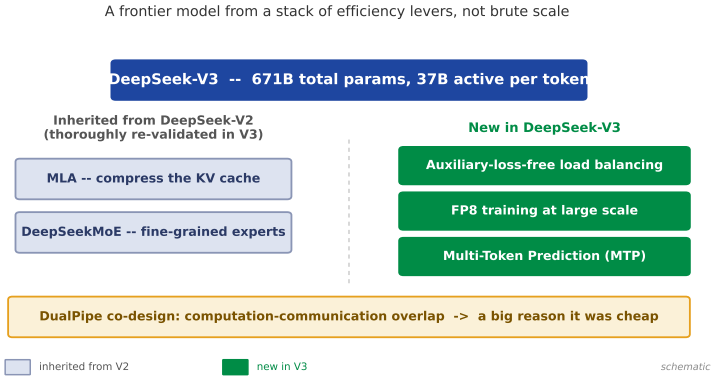
Recall the MoE distinction from **[LLM-13]**: **total** params (everything stored, sets memory) vs **active** params (what fires per token, sets compute).

DeepSeek-V3 pushes it hard:

- **671B total** / **37B active** – about **18x** more stored than fires.
- Each MoE layer: **1 shared expert** + **256 routed experts**, **top-8** routed per token.
- Pre-trained on **14.8T** tokens.

The whole design question: how do you get **frontier quality** out of those 37B active params – and train the 671B **cheaply**? V3's answer is a stack of **five efficiency levers**.

Five levers – two inherited, three new (do not conflate them)



Attribution matters. **MLA** and **DeepSeekMoE** are *inherited from DeepSeek-V2* and re-validated here – not V3 inventions. V3's own contributions are **auxiliary-loss-free balancing**, **large-scale FP8**, and **MTP**. **DualPipe** co-design is a big reason it was cheap.

MLA – shrink the KV cache

Inherited from DeepSeek-V2, thoroughly re-validated in V3.

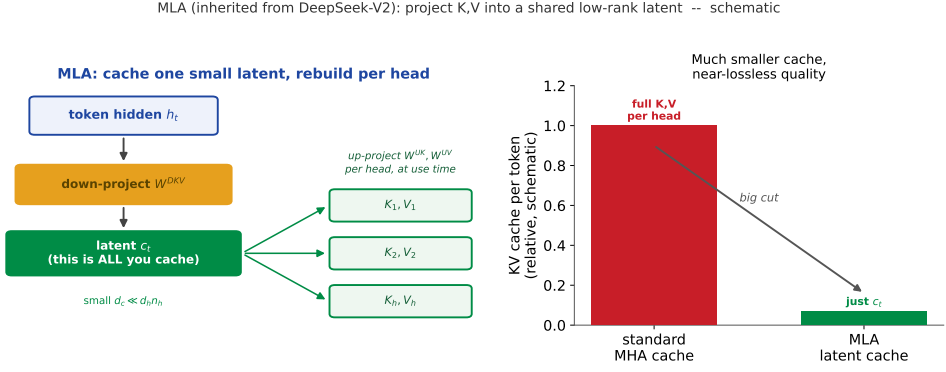
The KV-cache problem

To generate one token, attention needs the **keys and values of every past token**. So we **cache** them – and that cache is the memory wall at long context.

- The KV cache grows with **sequence length** $\times$ **layers** $\times$ **heads**.
- At long context it can **dwarf the model weights** in memory, and it is what you must hold per concurrent request when serving.
- Shrinking it directly buys **longer context** and **cheaper serving**.

This is a *memory* problem, not a compute one: the arithmetic is cheap, but you cannot afford to **store** full K and V for every head of every past token.

MLA: cache one low-rank latent, rebuild per head



Multi-head Latent Attention (from DeepSeek-V2): project K,V down to a shared **low-rank latent** c_t, cache *only* c_t, then up-project per head when needed. Big memory cut, quality stays near-lossless. The *why* matters more than the matrices here.

The two halves of attention's memory bill

FlashAttention ([LLM-15]) attacked the **attention-matrix** memory: never materialize the full $n \times n$ scores, stream them tile by tile.

MLA attacks the **KV-cache** memory: never store full per-head K,V, keep a small shared latent instead.

Two different memory costs, two different fixes:

- FA – the **scores** you compute *now*.
- MLA – the **K,V** you cached from the *past*.

Together they are why long-context attention is affordable at all.

Both are *efficiency* plays – the theme running through this stretch of the course.

Balancing experts without a quality tax

DeepSeekMoE is from V2. The auxiliary-loss-free fix is V3's own.

The balancing tension (recall [LLM-13])

MoE only works if tokens **spread across experts**. Left alone, the router **collapses** onto a few favourites and the rest go **dead**.

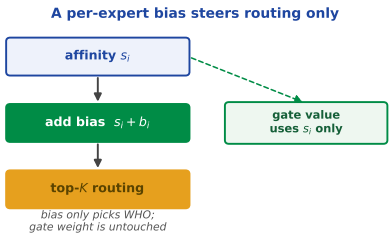
The standard fix from **[LLM-13]** is an **auxiliary balancing loss** added to the objective. But it is a real tension:

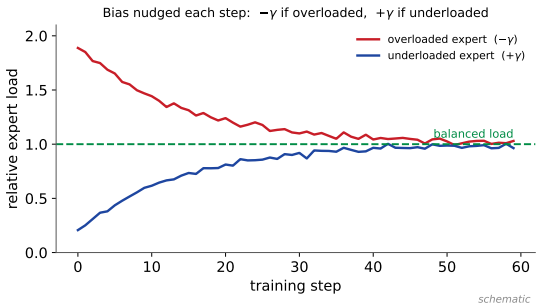
- Too weak – experts still collapse.
- Too strong – it **fights the main loss** and **costs quality**.

The aux loss pays for balance with a **gradient that is not about the task**. You are always trading model quality against load balance.

V3's fix: a routing bias, nudged online – no aux-loss tax

Auxiliary-loss-free balancing (V3): equalize experts with a routing bias, no aux-loss quality tax





Add a per-expert **bias** b_i to the affinity *for routing only* (top-K uses $s_i + b_i$; the gate weight still uses the original s_i). Each step, nudge b_i: $-\gamma$ if the expert is overloaded, $+\gamma$ if underloaded. Load evens out with **no extra gradient, no quality tax**.

Shared + fine-grained experts, plus a light safety net

DeepSeekMoE's structure (from V2):

- **Fine-grained routed experts** (256 of them) – smaller experts, more specialization per FLOP.
- **Shared expert(s)** always on – absorb the common, general computation so routed experts can specialize.

On top, V3 keeps one guardrail:

- A **light complementary sequence-wise balance loss** with a *tiny* weight – just to stop any single sequence from going extremely unbalanced.

The heavy lifting is the **bias trick**; the tiny sequence-wise loss is only a safety net, not the old quality-costing aux loss.

FP8 training

8-bit for the forward *and* backward pass – at frontier scale.

Why 8-bit training is hard

QLoRA ([LLM-12]) taught the quantization intuition: fewer bits, less memory – but a tiny dynamic range struggles with **outliers**. There it was for *frozen storage*. FP8 training pushes low precision into the **live forward and backward pass**.

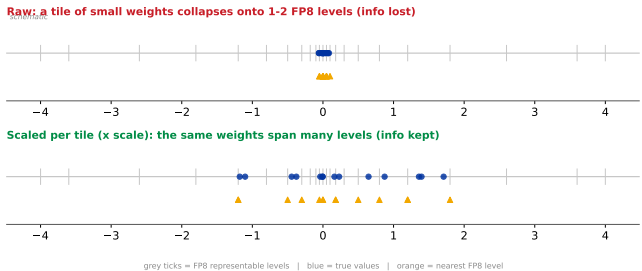
- FP8's dynamic range is **tiny**; activation and gradient **outliers overflow**.
- Naive FP8 training **diverges** – errors compound across every layer and step.

Quantizing a *frozen* weight for storage is one thing. Training *in* 8-bit – where every rounding error feeds the next gradient – is a much harder problem.

FP8 in one picture: use the range you have

FP8 has **very few** representable values over a **small** range.

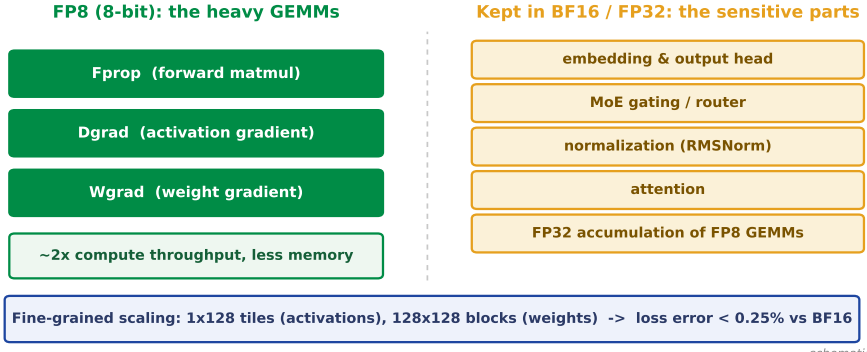
- A tile of tiny weights **collapses** onto 1-2 levels – most of the precision is wasted, information lost.
- Multiply the tile by a **per-tile scale** first: the same values now **spread across many levels**. Divide it back out afterwards.



That is **fine-grained (per-tile / per-block) scaling**: choose a scale for each small block so it fills the FP8 range – the fix that keeps 8-bit *training* from losing the small values or overflowing on the big ones.

V3's recipe: 8-bit where it is safe, precision where it matters

FP8 training (V3): 8-bit for the big matmuls, higher precision where it matters



Fine-grained scaling (per-tile 1×128 activations, per-block 128×128 weights) keeps outliers in range; **FP32 accumulation** fixes FP8's $\sim$14-bit accumulate; sensitive parts stay in **BF16/FP32**. Result: $\sim$**2x** throughput, less memory, **loss error** $< 0.25\%$ vs BF16.

Multi-Token Prediction

A denser learning signal – and a speed-up at inference time.

Predict-first: does predicting extra tokens help the main model?

Standard language-model training predicts only the **next** token at each position.

What if we also asked the model to predict the token *after* that, and the one after that?

Predict-first: does predicting extra tokens help the main model?

Standard language-model training predicts only the **next** token at each position.

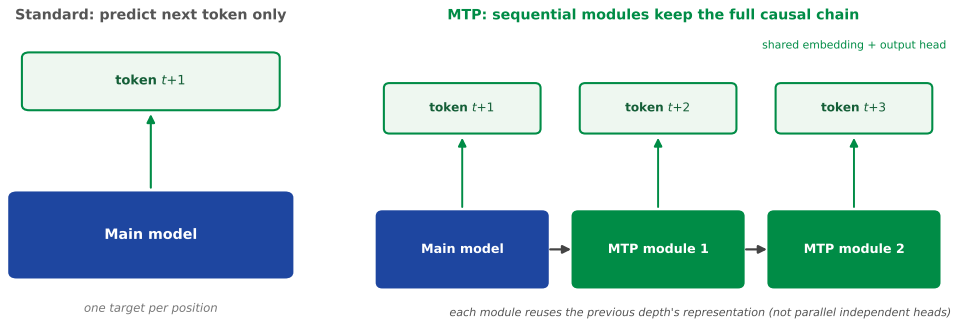
What if we also asked the model to predict the token *after* that, and the one after that?

It helps. A prediction target at *every* future depth is a **denser training signal** per position – better sample efficiency – and it forces the model to **pre-plan** its representations.

The extra targets are a training aid. At inference you can keep just the main next-token model – or reuse the extra heads to go *faster* (next slide).

MTP: sequential modules that keep the full causal chain

Multi-Token Prediction (V3): denser signal per position, and it enables speculative decoding -- schematic

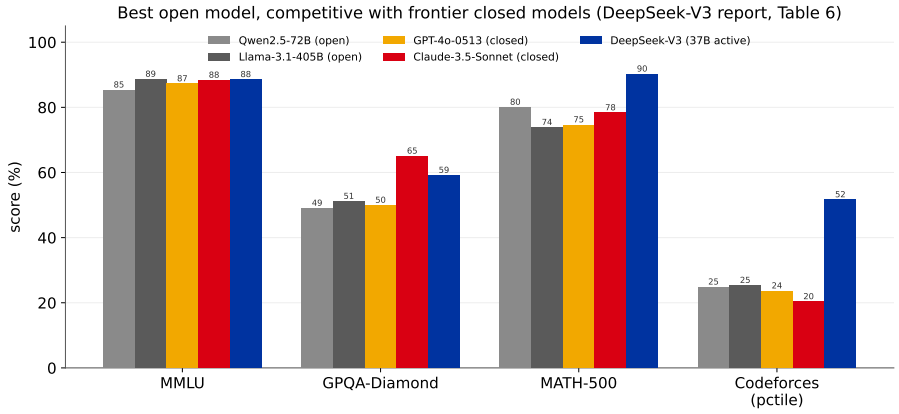


V3 adds **sequential** lightweight modules that each reuse the previous depth's representation – keeping the **full causal chain** (distinct from Gloeckle et al.'s *parallel independent heads*). Bonus: those modules enable **speculative decoding** – the 2nd-token prediction is accepted **85–90%** of the time, so inference gets faster.

Does it work?

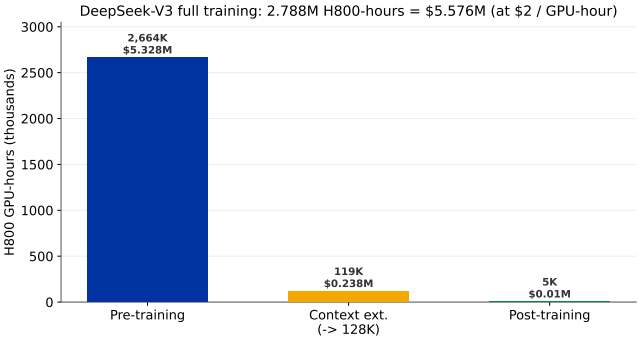
Best open model, competitive with frontier closed – for a fraction of the cost.

Benchmarks: best open, competitive with closed



With **37B active** params, V3 is the **strongest open** model and holds its own against **GPT-4o** and **Claude-3.5-Sonnet** – *dominant* on math (MATH-500, AIME) and competition code (Codeforces), a step behind Claude on GPQA. It beats Qwen2.5-72B and Llama-3.1-405B across the board.

The cost, and what it made possible



180K GPU-hours per trillion tokens -- ~3.7 days/T on a 2048-GPU H800 cluster. (Table 1; $2/GPU-hour assumption.)

Full training: **2.788M H800-hours**, **$5.576M**; pre-training alone **2.664M**. Frontier quality without a frontier budget.

[LLM-11] R1 took *this* base model and added reasoning RL. The efficiency here is exactly what made that next step **affordable**.

Recap

- **Frontier MoE, cheaply: 671B total / 37B active**, 14.8T tokens, full training **2.788M H800-hours (≈$5.6M)** – efficiency, not scale.
- **Inherited from DeepSeek-V2 (re-validated): MLA** (cache a low-rank KV latent) and **DeepSeekMoE** (fine-grained + shared experts).
- **New in V3: auxiliary-loss-free balancing** (a routing bias nudged online, no quality tax), **large-scale FP8 training** (loss error $< 0.25\%$), and **MTP** (denser signal + speculative decoding). **DualPipe** co-design made it cheap.
- **Results:** best open model, competitive with GPT-4o / Claude-3.5; the base **[LLM-11] R1** was built on.

Next: *[LLM-11] R1* adds reasoning RL on top of this base; *[LLM-13] MoE* and *[LLM-12] QLoRA* are the roots of its balancing and quantization ideas. The frontier is now **efficient training**, not just bigger.