SAME MODEL, DIFFERENT NUMBERS: MEASURING AND REDUCING CROSS-VENDOR MISMATCH IN LANGUAGE MODELS

Anonymous authors
Paper under double-blind review

ABSTRACT

Running the same language model on different graphics processing unit (GPU) vendors can produce different logits, even when the model weights and inputs are the same. We analyze cross-vendor mismatch in two dense and two mixture-of-experts (MoE) models with five metric families, namely bitwise equality, logit differences, top-K consistency, token agreement, and task accuracy, together with expert routing. We trace one source of the mismatch to accumulation order inside vendors' matrix instructions. Upcasting to FP32 reduces the dense model's logit error by 43% at three times the runtime, yet keeping only the MLPs in BF16 retains 94% of this gain at 1.3 times the runtime, so most of the cost of full upcasting buys little. In the MoE models, FP32 and FP16 both lower the probability error but raise the logit error and change expert selection, and FP16 fails in the dense model. An output-head low-rank adapter (LoRA) does not help either, since the final hidden state does not predict the mismatch. The mismatch also carries into training. With every seed fixed, a student distilled from a teacher running on AMD answers 431 MMLU questions differently from one distilled from the same teacher on NVIDIA. Under FP32 upcasting, bitwise equality barely changes while the output distributions move most of the way to the reference, so judging cross-vendor agreement by a single measure misreads both its cost and its gains.

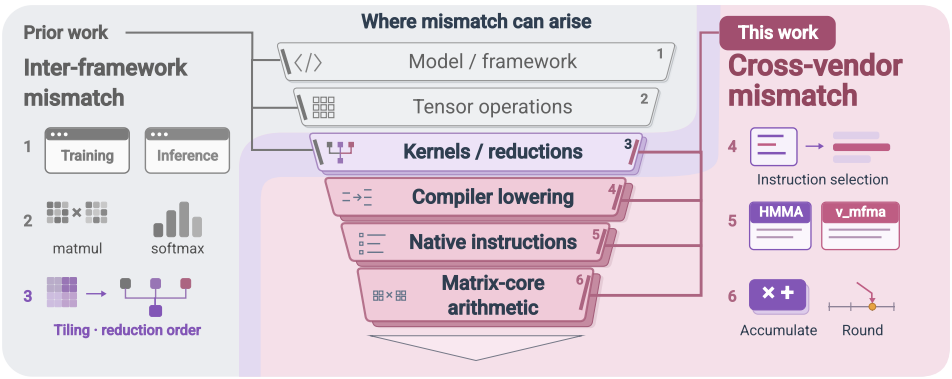


Figure 1: Cross-vendor mismatch spans the execution hierarchy. Inter-framework studies address differences in models, frameworks, and tensor-operation choices. At lower levels, kernel and hardware arithmetic can introduce additional numerical differences. The hierarchy organizes the analysis of how these differences propagate through model outputs and behavior.

1 INTRODUCTION

Large language models (LLMs) are increasingly developed and deployed across heterogeneous hardware and software environments (Chen et al., 2025). Different execution stacks implement floating-point arithmetic differently, producing different outputs across GPU vendors even when the model

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

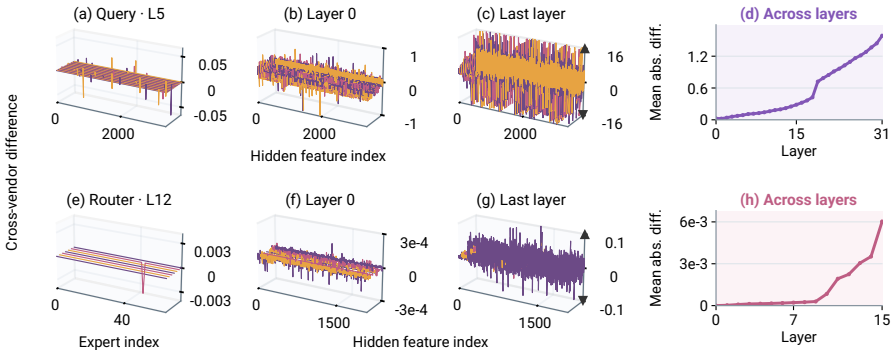


Figure 2: Cross-vendor differences in a Jais dense model (Jais Team, 2026, a–d) and OLMoE MoE model (Muennighoff et al., 2025, e–h). Components (a/e) and first blocks (b/f) receive identical inputs. Last-block outputs (c/g) and layer curves (d/h) use independent full forwards, allowing differences to propagate. The curves show the mean absolute difference across all tokens and hidden features at each block output.

weights and inputs are unchanged, which we call cross-vendor mismatch (Li et al., 2024; Zahid et al., 2024). In our measurements, fewer than one in ten output logits of an 8B model are bit-identical between an NVIDIA and an AMD GPU. This matters when a model developed on one vendor is used on another, since its output values determine the probabilities and choices used in later computation. These differences raise questions about the reliability and reproducibility of LLMs across execution environments (Chauvin et al., 2025; Yuan et al., 2025), and assessing them requires examining both the generated answers and the numerical values that produce them.

The consequences of cross-vendor mismatch depend on how these outputs are used. For example, in knowledge distillation (KD), the teacher's probability distribution supplies the student's training targets (Hinton et al., 2015). Differences in these probabilities change the targets even when the teacher selects the same token. Reinforcement learning (RL) also uses output probabilities to compute training objectives, making consistency between generation and training relevant to learning (Qi et al., 2025; Zhong et al., 2026). During generation, a small numerical difference can change the selected token when competing scores are close. In mixture-of-experts (MoE) models, router values determine expert selection and mixture weights, and a changed selection changes which parameters process the token (Ma et al., 2025). These choices affect computation and the final answer.

Reducing mismatch requires understanding where the numerical differences arise, from execution frameworks down to hardware arithmetic, as illustrated in Figure 1. Prior work mainly studies inter-framework mismatch in the upper parts of this hierarchy (Ma et al., 2025; Qi et al., 2025; Zhang et al., 2026; Zhong et al., 2026). Other cross-accelerator work validates alternative training stacks against NVIDIA-based frameworks through forward-precision and training-loss comparisons (Chen et al., 2025). Our work focuses on cross-vendor mismatch at the lower levels of the hierarchy. Here, the underlying matrix arithmetic can differ in its rounding and accumulation behavior across GPU architectures (Li et al., 2024). This is a non-trivial problem that requires deeper investigation at the machine-code level. Matching the model code and framework settings alone therefore does not ensure the same numerical behavior across vendors.

How agreement is measured decides what a study concludes, and prior work measures it incompletely. Deterministic-inference work targets bitwise-identical outputs (He & Lab, 2025; Zhang et al., 2025; Yang et al., 2026) or identical token sequences (Gond et al., 2026; Chu et al., 2026), and LayerCast (Yuan et al., 2025) judges FP32 upcasting by the spread of accuracy, response length, the first divergent token, and the top-1 probability across NVIDIA configurations only, never across vendors, with its cost reported against full FP32 rather than native BF16. These measures can miss most of what cross-vendor mismatch changes. An intervention can leave bitwise equality, generated tokens, and accuracy almost unchanged while moving the output distribution substantially toward the reference, and an analysis built on those measures alone would conclude that it does nothing. The output distribution is what distillation and reinforcement learning consume, and a cost mea-

sured against the wrong baseline hides what an intervention actually costs. Judging cross-vendor agreement therefore requires several metric families at once.

We study cross-vendor mismatch as both an analysis of the disagreement and a comparison of ways to reduce it. We connect internal values, including router values and expert selections in MoE models, to five output-level metric families, and trace the differences arise down to the matrix operations that accumulate them. We then ask where the mismatch can be reduced, by changing the arithmetic format (Qi et al., 2025; Yuan et al., 2025; Cooper et al., 2026) or by correcting only the output with a low-rank adapter (Hu et al., 2022), and whether it carries into distillation. We find that the mismatch is widespread and survives the obvious fixes, that most of the gain from higher precision comes without its cost, and that judging agreement by bitwise equality alone misses what matters.

Our contributions are as follows.

1. We characterize cross-vendor mismatch in denses and MoE models with five families of agreement measures, from bitwise equality to task accuracy, and trace where it arises by comparing values across layers and isolating accumulation paths in matrix multiplication.

2. We examine where the mismatch can be reduced, comparing precision changes throughout the forward pass, higher precision in selected modules only, and a low-rank correction at the output head, together with their cost.

3. We examine how the choice of agreement measure changes the assessment of each intervention, contrasting bitwise equality with the output distributions and choices that downstream uses consume.

4. We test whether the mismatch carries into downstream training by distilling a teacher run on each vendor and precision into the same student.

2 Cross Vendor Numerical Compatibility

2.1 Mismatch Between Vendors

Floating-point formats represent values with limited precision. Each arithmetic result is rounded to a representable value, and one unit in the last place (ULP) is the local spacing between them (Goldberg, 1991). In 16-bit brain floating point (BF16), one ULP is 0.0078125 for magnitudes in $[1, 2)$ and 0.125 in $[16, 32)$. Execution stacks can change the order of operations and the precision of intermediate results, so a small difference can cross a rounding boundary and then propagate through later layers, even with the same weights and inputs (Figure 2).

A controlled matrix-multiplication case study locates one source of these differences (Section C.2). We compare three accumulation paths for the same projection on both vendors. Vendor general matrix multiplication (GEMM) and a Triton kernel that fixes the order of reduction tiles both differ across vendors at most outputs, and the tiled kernel already differs within its first output tile, where there is no tile order left to control. Sequential scalar accumulation agrees exactly (Table 1). The difference therefore lies in how each vendor's matrix instructions accumulate within a tile (Fasi et al., 2021), which standard framework settings do not control. The tiled kernel uses the same matrix instructions as model inference, so any BF16 model on these GPUs inherits this difference.

Table 1: Mismatch in the matmul case study.

Path	Captured	Synthetic
Full projection		
Vendor GEMM	14738/15552	14444/15552
Ordered-tile	15211/15552	14960/15552
Scalar	0/15552	0/15552
First output tile		
Ordered-tile	1022/1728	398/1728
Scalar	0/1728	0/1728

Unequal / total FP32 elements.

For an input x and model weights θ, the reference (R) and target (T) executions produce next-token logits $z_v(x) = f_v(x; \theta) \in \mathbb{R}^V$ for $v \in \{R, T\}$, where V is the vocabulary size. Each execution depends on its hardware, software, kernels, precision, and batching, and the reference defines the target outputs. In mixture-of-experts (MoE) models, router values also select each token's experts.

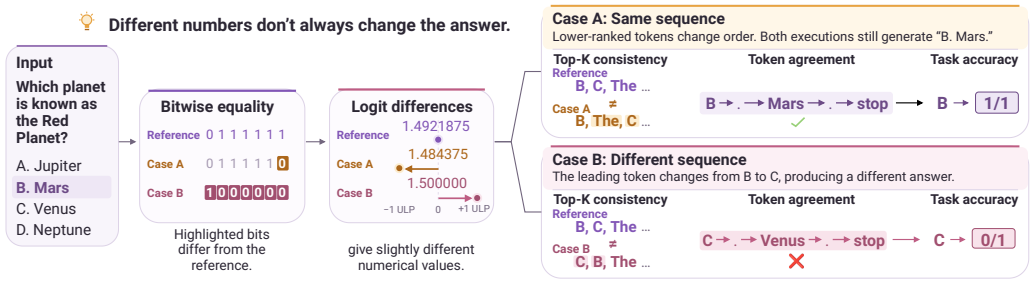


Figure 3: Two constructed examples of numerical mismatch across the five metric families. Case A changes logit values and lower rankings while preserving the generated sequence and correct answer. Case B changes the leading token and leads to an incorrect answer.

2.2 Evaluation Metrics

A numerical difference matters through what later computation does with it. A one-ULP change can reorder lower-ranked candidates while the generated answer stays the same, or it can overtake the leading token and change the answer (Figure 3). Numerical differences can also change learning targets, such as a teacher's probabilities, when the generated answer agrees.

We therefore define five output-level metric families. **Bitwise equality** asks whether the executions reproduce exactly the same represented values. **Logit differences** measure the size of the differences in logits and probabilities. **Top-K consistency** examines whether the same candidates remain highly ranked and whether their order is preserved. **Token agreement** compares next-token choices on the same inputs and complete sequences when each execution follows its own continuation. **Task accuracy** checks whether the resulting answers are correct. For MoE models, we also compare router values and selected experts. Appendix A gives the complete formulas and evaluation conventions.

3 Methodology

We analyze two ways to reduce cross-vendor mismatch against the same fixed NVIDIA reference, each answering one question. First, can the mismatch be mitigated by changing the arithmetic itself? We run the whole forward pass in a different floating-point format, without any training. Second, can the mismatch be represented and corrected at the model's output? We train a low-rank adapter on the output head while the rest of the computation stays unchanged. The two interventions act at opposite ends of the computation, so comparing them shows where the mismatch can be reduced.

3.1 Precision interventions

Precision changes act on every operation of the models. The case study in Section 2 shows one way intermediate values diverge, as the two vendors accumulate the same products in different orders, and Figure 4 shows why the arithmetic format matters for such order differences. Matrix instructions already accumulate BF16 products in FP32, but BF16 execution rounds every intermediate value to BF16 between operations, including the residual stream and normalization outputs. Every partial sum is rounded to the nearest representable value, so a different order changes the result when a partial sum is rounded differently. In the constructed BF16 example, the same three numbers added in two orders end one unit in the last place (ULP) apart, because one order lands exactly halfway between two representable values and the tie is broken in opposite directions. A finer format rounds each partial sum by a smaller amount, so the same order difference produces a smaller discrepancy, and in this example FP16 and FP32 give the same result for both orders. A finer format therefore makes each rounding difference smaller, but it does not remove the different accumulation orders, so it can reduce the mismatch without guaranteeing identical outputs.

FP32 upcasting. Following LayerCast (Yuan et al., 2025), the linear weights stay stored in BF16 while products, activations, partial sums, and logits use 32-bit floating point (FP32). This cannot recover bits absent from the weights, but it keeps more low-order bits in every intermediate value

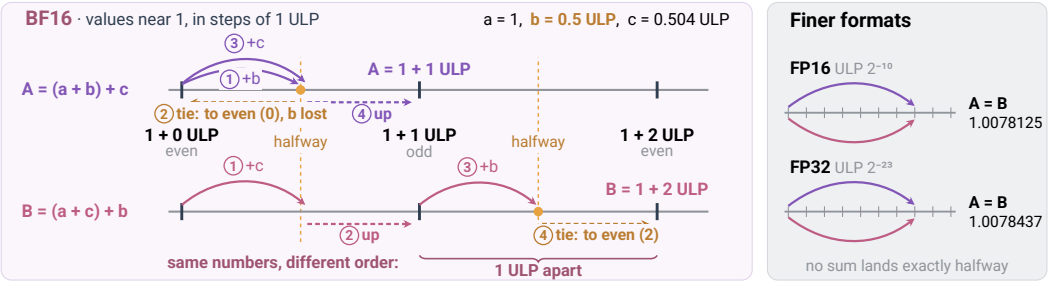


Figure 4: BF16 can gives different value due to less finer on ULP steps. BF16 values near one lie one ULP apart, sums round to the nearest, with ties to the even step. With $b = 0.5$ ULP and $c = 0.504$ ULP, in $A = (a + b) + c$ ① $1 + b$ lands halfway, ② rounds down to even so b is lost, and ③–④ c rounds it up to $1 + 1$ ULP. In $B = (a + c) + b$ the sum rounds up to $1 + 1$ ULP, then b lands halfway and rounds to the even $1 + 2$ ULP. FP16 and FP32 have finer steps, so both orders agree.

between operations, at the cost of more arithmetic and memory. We test it against the fixed NVIDIA BF16 reference.

FP16. 16-bit floating point (FP16) has 10 significand and 5 exponent bits, against 7 and 8 for BF16. Its finer grid can absorb small implementation differences at the BF16 memory footprint, as argued for training–inference mismatch in reinforcement learning (Qi et al., 2025). Its narrower range risks overflow, so we first check that the outputs are finite.

3.2 Output-head LoRA

We also test whether a cheap learned correction at the output can absorb the mismatch. The mismatch is small in magnitude, but a small difference is correctable only if it is predictable from what the head receives, so the adapter also serves as a control for whether the mismatch is visible at the output. We add a low-rank adapter $\Delta W = \frac{\alpha}{r} BA$ to the output head $W \in \mathbb{R}^{V \times d}$, with $A \in \mathbb{R}^{r \times d}$, $B \in \mathbb{R}^{V \times r}$, and rank $r \ll \min(V, d)$, and keep every base parameter frozen. For a target hidden value h_T, the adapter changes the logits by $\Delta z_T(h_T) = \frac{\alpha}{r} BAh_T$, and losses are evaluated on the corrected native logits. It can change output values, probabilities, and rankings for the same input, but not the hidden states, matrix operations, or expert routing earlier in the forward pass. During generation, a changed token can still change later computation. The adapter therefore tests how much of the mismatch can be corrected at the output alone.

Training objective. We combine four losses that target different aspects of agreement. Raw-logit MSE targets individual values, forward KL divergence the reference distribution, token-gap the leading token's separation from the alternatives, and top-5-gap the gaps that set the order and boundary of the reference top 5 (complete formulas in Section A.3). Each loss ℓ_j is divided by a fixed normalizer c_j, its initial mean over training positions, and the objective is $\mathcal{L} = \sum_{j \in \mathcal{J}} w_j \ell_j / c_j$ with $\mathcal{J} = \{\mathrm{MSE}, \mathrm{KL}, \mathrm{token}, \mathrm{top5}\}$. Varying w_j tests which aspects training emphasizes, but a weighting does not guarantee that all agreement measures improve together.

4 Experimental Design

We study Jais 8B (Jais Team, 2026) as the dense model and OLMoE (Muennighoff et al., 2025) as the MoE model. NVIDIA A100 provides the reference and AMD Instinct MI210 runs the target. NVIDIA A100 uses the Ampere architecture, and AMD Instinct MI210 uses the CDNA2 architecture, which was built as a direct competitor to Ampere. The two therefore belong to a comparable generation. The reference serves as an operational anchor rather than a claim that it is more accurate, so we report reference fidelity and task accuracy seperately.

For Jais, NVIDIA generates greedy reference responses of up to 1,024 tokens, 12.5 tokens long on average, for 1,200 training and 200 development questions, split equally between Massive Multitask

Language Understanding (MMLU) (Hendrycks et al., 2021) and ArabicMMLU (Koto et al., 2024), the natural second benchmark for the Arabic-centric Jais, where the English-only models use ARC-Challenge. AMD hidden values on the training responses train the adapters in an offline fashion. We evaluate on the development responses and on the deduplicated MMLU and ArabicMMLU test sets, which include the adapter's training questions (Table 7 excludes them). For OLMoE, NVIDIA generates 1,024-token references for 640 MMLU and ARC-Challenge questions. The base model never emits an end-of-sequence token, so every response reaches this cap, and rollout agreement is therefore not comparable between the two models. Each comparison requires us to move the full-vocabulary logits of every position from one system to the other, about 13 GB in BF16 for the 128 development responses alone, so a longer cap would make this transfer impractical. The 128 development responses evaluate every intervention, and the other 512 train the adapters, on their first 128 positions, and the distillation students (Section 5.5). Accuracy uses the complete MMLU and ARC-Challenge test sets. Intervals are paired bootstrap intervals over questions, and Appendix B gives the protocol.

5 Experiments & Results

5.1 Overall comparison

Table 2 compares all interventions across the five output families for the dense Jais model and the mixture-of-experts (MoE) OLMoE model. We use Native AMD (BF16) as the baseline. The precision interventions are FP32 upcasting, our adaptation of LayerCast (Yuan et al., 2025), and FP16. The output-head interventions are output-head LoRA with Equal and a low-rank adaptation (LoRA) trained only with forward Kullback–Leibler (KL) divergence. No intervention improves every family. On Jais, FP32 upcasting reduces the numerical mismatch substantialy, while neither output-head adapter improves on Native AMD (BF16). On OLMoE, both precision changes lower the probability error but raise the raw-logit error and match fewer top-5 lists. Output-head LoRA fails in both models, so we do not repeat it, and we test the precision interventions on a second dense model, Llama-3.1-8B, and a second MoE model, Qwen1.5-MoE. Over OLMoE's 1,024-token responses, only about a third of rollouts stay identical to the reference even without any intervention.

5.2 Can precision mitigate the mismatch?

On Jais, FP32 upcasting brings the numerical outputs much closer to the reference. It reduces raw-logit MSE by 42.8% and forward KL by 37.1% relative to Native AMD (BF16) (Figure 7a). It also matches 126 more ordered top-5 lists. The discrete choices change far less, as it matches two fewer leading tokens and one fewer rollout.

Task accuracy is essentially unchanged. FP32 upcasting answers 19 more MMLU and seven more ArabicMMLU questions correctly than Native AMD (BF16), and neither change is statistically significant. The numerical gain has a clear cost. It takes 3.15 times as long and uses 2.55 times the peak GPU memory (Table 3). On Jais, FP16 produced nonfinite outputs, so it has no numerical comparison. FP16 overflows to 65,504, and the Jais MLP activation of block 20 reaches about 1.2×10^5 in BF16. Whether FP16 is usable therefore depends on the activation of each model.

Table 3: Cost on one MI210, same workload within each model. *Relative to a native run measured alongside (1,054 s). Native timings vary by about 7% between runs. †All except MLPs in FP32 (Section 5.3)

	Time (s)	Memory (GB)
Jais		
Native AMD (BF16)	6,111	18.7
FP32 upcasting	19,245 (3.15×)	47.7 (2.55×)
Selective FP32†	(1.30×)	(2.02×)
OLMoE		
Native AMD (BF16)	1,129	14.4
FP32 upcasting	1,405 (1.24×)	28.7 (1.98×)
FP16	1,154 (1.02×)	14.4 (0.99×)
Head LoRA (KL)*	1,066 (1.01×)	15.1 (1.05×)
Head LoRA (Eq.)*	1,054 (1.00×)	15.1 (1.05×)

On OLMoE both formats are finite, and the result is mixed. Relative to Native AMD (BF16), FP32 upcasting raises raw-logit MSE by 10.7% but lowers forward KL by 3.8% and probability MSE by 8.9%. FP16 raises raw-logit MSE by 16.9% and forward KL by about 2%, and lowers probability MSE by 4.4%. When the outputs rounded to BF16 before the comparison, the KL reductions become increases (Section B.4). Both match fewer ordered top-5 lists. First-layer expert agreement falls from 99.5% to 92.9% under FP32 upcasting and 92.6% under FP16 and stays lower on all 16 layers (Figure 6).

Table 2: AMD against the NVIDIA reference. Top-5 is ordered top-5 agreement, and rollout the share of responses identical to the reference. Selective FP32 keeps only the MLPs in BF16. Bold is best and underline second best per model. FP16 and FP32 outputs are rounded to BF16 for bit-equal.

Method	Bit-equal (%)↑	Logit MSE↓	KL (10^{-3})↓	Top-5 (%)↑	Rollout (%)↑	MMLU (%)↑	AMMLU/ ARC (%)↑
Jais (dense)							
Native AMD (BF16)	9.26	0.2873	7.08	66.53	93.50	60.86	71.36
FP16	nonfinite outputs						
FP32 upcasting	**11.24**	**0.1644**	**4.45**	**71.58**	93.00	**61.00**	**71.40**
Selective FP32	11.07	0.1717	4.62	71.38	**94.00**	60.98	71.36
Head LoRA (KL)	9.22	0.2888	7.67	65.89	92.50	60.88	71.35
Head LoRA (Equal)	8.65	0.2957	7.21	66.37	93.50	60.87	71.34
OLMoE (MoE)							
Native AMD (BF16)	**14.17**	**0.0215**	0.194	**69.57**	34.38	47.37	54.10
FP16	12.85	0.0251	0.198	66.66	**35.16**	**47.46**	54.10
FP32 upcasting	13.79	0.0238	**0.187**	67.75	33.59	47.44	**54.69**
Head LoRA (KL)	13.03	0.0219	0.202	69.00	33.59	47.37	54.10
Head LoRA (Equal)	13.92	0.0216	0.202	69.04	33.59	47.37	54.10
Llama-3.1-8B (dense)							
Native AMD (BF16)	9.52	0.0053	0.218	80.27	**16.41**	67.04	**83.62**
FP16	9.77	0.0059	0.151	79.33	10.94	**67.06**	83.53
FP32 upcasting	**11.50**	**0.0045**	**0.130**	**81.00**	8.59	**67.06**	83.45
Qwen1.5-MoE (MoE)							
Native AMD (BF16)	**5.64**	**0.0152**	**1.66**	**66.75**	0.78	60.74	76.54
FP16	4.97	0.0199	2.30	62.22	**2.34**	**60.92**	**76.79**
FP32 upcasting	5.43	0.0177	2.01	63.89	**2.34**	60.84	76.62

Table 4: Distilling OLMoE into OLMo-1B with the teacher run on each execution. Same ans. is the share of MMLU questions answered as the NVIDIA-teacher student does, and teacher KL is the teacher's own distance from the NVIDIA teacher. Gray numbers give the accuracy change from the NVIDIA-teacher student, and bold marks the best AMD teacher (smallest change for accuracy). The other-seed student changes only the data order.

	vs. NVIDIA-teacher student			Accuracy (%)↑		Teacher
Teacher	KL (10^{-4})↓	Top-1 (%)↑	Same ans. (%)↑	MMLU	ARC	KL (10^{-4})↓
No distillation	–	–	20.77	24.79 −1.14	26.37 −0.51	–
NVIDIA, repeat	0	100.00	100.00	25.93	26.88	0
NVIDIA, other seed	73.4	99.15	82.59	25.67 −0.26	28.50 +1.62	0
Native AMD (BF16)	1.61	99.84	**96.93**	**26.12** +0.19	27.22 +0.34	1.94
AMD FP32 upcasting	23.7	99.51	94.67	25.64 −0.29	26.79 −0.09	**1.87**
AMD FP16	**1.26**	**99.87**	96.46	26.21 +0.28	**26.88** ±0.00	1.98

The two further models confirm this split (Section E.1). FP32 upcasting lowers the logit error of the dense Llama model, while both finer formats move Qwen1.5-MoE further from the reference on every logit measure. Higher precision is therefore not a safe default for matching a BF16 reference.

5.3 Selective FP32 closes the gap at a fraction of the cost

Full FP32 upcasting, as proposed by LayerCast (Yuan et al., 2025), buys its gain with 3.15 times the runtime, so we ask which part of the computation needs the higher precision. Figure 5 upcasts one module type at a time and in combination, with the full results in Table 15. Keeping only the MLPs in BF16 retains 94% of the MSE reduction and 93% of the KL reduction of full upcasting and matches almost as many top-5 lists, with accuracy unchanged, at 1.30 times the native forward time. It therefore runs 2.3 times faster than full upcasting. The MLPs are the most expensive module to upcast and the least useful, since upcasting them alone raises KL, and Jais is where full upcasting helps most, so even here most of its cost buys little.

The five metric families show why this tradeoff stays invisible to a single measure. Full upcasting raises the share of bit-equal logits only from 9.3% to 11.2%, so almost nine in ten values still differ from the reference, yet it lowers KL by 37% and matches five percentage points more ordered top-5 lists. Closing the gap therefore does not require bitwise equality. Judged by bitwise equality, full upcasting achieves little, and judged by accuracy alone, neither setting changes anything, although the output distributions move most of the way to the reference. These distributions are what distillation and other uses of probabilities consume (Section 5.5), so a system judged on one family can pay for precision it does not need.

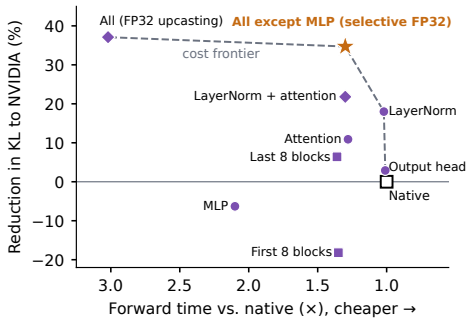


Figure 5: Selective FP32 in Jais. The dashed line joins the settings that no cheaper setting beats on KL (Table 15).

5.4 Can output-head LoRA correct the mismatch?

Output-head LoRA brings neither model closer to the reference at the common training endpoint. On Jais, output-head LoRA with Equal raises raw-logit mean squared error (MSE) and forward KL relative to Native AMD (BF16) and matches fewer bit-equal values and ordered top-5 lists, and the KL-only adapter also raises both and loses rollout agreement. OLMoE gives the same outcome. Both adapters lower their training objective, yet on the development responses they raise forward KL by about 4% (KL-only $+4.2\%$, Equal $+4.1\%$) and raw-logit MSE by 1.7% and 0.4%, and they match about 700 fewer ordered top-5 lists out of 131,072. Expert selection is identical to Native AMD (BF16), because the adapter acts after the last expert layer. The adapters are cheap, costing at most 1% more time and 5% more memory (Table 3), and task accuracy changes by only a few answers. Changing the loss weights does not help either, as all seven weighting profiles end with higher MSE and KL than Native AMD (BF16) in both models (Section D.2). The adapter fits the training responses but not unseen ones, because the mismatch it must learn behaves like rounding noise rather than a fixed map from the hidden state (Section 6).

5.5 Impact on Downstream Learning

We use knowledge distillation (KD) to test whether cross-vendor mismatch in a teacher changes the student (Hinton et al., 2015). We choose KD because the student learns directly from the teacher's probabilities, so logit differences change its training targets. The teacher is OLMoE and the student is OLMo-1B (Olmo Team, 2024), a separate model with the same tokenizer. The student minimizes forward KL to the teacher's 64 most likely tokens, renormalized. Each student learns from one teacher execution, which is NVIDIA BF16, Native AMD (BF16), AMD FP32 upcasting, or AMD FP16. All students share prompts, reference responses, training setup, starting checkpoint, and AMD hardware, so only the teacher's probabilities differ. A repeated NVIDIA-teacher student shows how much students differ without any teacher mismatch, and a student trained with another data-order seed shows ordinary training variation. Table 4 compares each student with the NVIDIA-teacher student and reports its accuracy, and Table 6 lists the training settings.

The teacher's mismatch carries into the student. Repeating the NVIDIA-teacher run reproduces the student bit for bit, so any remaining difference comes from the teacher alone. The student distilled from the Native AMD (BF16) teacher differs from the NVIDIA-teacher student by a forward KL of 1.6×10^{-4} and gives a different answer on 431 MMLU questions (3.1%) and 25 of 1,172 ARC-Challenge questions, while its accuracy changes by only 0.2 points. Upcasting the teacher to FP32 makes this worse, although its own KL to the NVIDIA teacher is the lowest of the three, as its student moves 15 times further in KL and changes 748 MMLU answers (5.3%). The FP16 teacher behaves like the native one (497 MMLU answers). A different data-order seed changes 2,445 answers of this near-chance student, so the hardware effect is smaller than ordinary training variation. The user chooses a seed, however, while the hardware behind a shared cluster or cloud endpoint is often out of their control, so fixing seeds cannot restore reproducibility.

6 DISCUSSION

What precision fixes, and what it does not. FP32 upcasting acts on the rounding of every intermediate value, but the different accumulation orders remain, so the outputs stay unequal. In OLMoE a finer format removes rounding differences between vendors but also stops rounding the way the BF16 reference does, and a changed expert choice then carries the difference into every later layer. Precision on one vendor alone cannot reach a BF16 reference, yet FP32 on both makes 98.4% of OLMoE responses identical (Section E.2).

Why output-head LoRA is not enough. The output head acts after all earlier computation, so it can only map the AMD hidden value that reaches it into a correction. Out of fold, the best linear map from that value increases the error by 0.81% (Section D.3), and every adapter and weighting ends with higher MSE and KL in both models. Computing the AMD head in FP32 and rounding its output to BF16 reproduces the native logits exactly (Section E.3), so the difference already sits in the hidden value, in a form a head correction cannot learn.

7 RELATED WORK

Cross-vendor numerical differences arise at several levels. Li et al. (2024) find rounding and accumulation differences across matrix accelerators, and Zahid et al. (2024) trace program disagreements to mathematical functions and compiler optimizations Yuan et al. (2025) study variability in LLMs across configurations and propose LayerCast, which keeps BF16 weights and computes in FP32. Another work uses FP16 computation and found that it can reduces training–inference differences in reinforcement learning (Qi et al., 2025). Those past work motivate us to do further analysis on the impact of precision towards the LLM outputs for the cross-vendor settings. To lower level, Bhola & Duraisamy (2024) define that the nondeterminism caused by randomness can be assumed of a probabilistic error model and the way to fix this is to freeze input, execution order, and rounding mode. This is similar with what we found with our experiment on scalar matmul in Table 1.

Controlling execution can reduce mismatch. Batch-invariant kernels keep a request's reduction order independent of its batch (He & Lab, 2025), fixed reduction trees preserve accumulation order across tensor-parallel sizes (Zhang et al., 2025), and Yang et al. (2026) reconstruct a vendor library's reduction order to reproduce its tensor-core results bit for bit, which makes runs repeatable within one vendor but not identical across vendors. These approaches require control over every kernel on both sides, whereas we instead measure how far unmodified executions diverge and what model-level and precision-level interventions recover. Note that also the past work investigating on the same-vendor, not different vendors like our work. Routing replay stabilizes MoE training (Ma et al., 2025) which motivates us to investigate the impact on the MoE models. LLM-42 verifies tokens with fixed-size batches (Gond et al., 2026) and MarginGate uses token margins to reduce verification (Chu et al., 2026). Such sequence agreement leaves open how closely the probabilities agree, which we measure alongside generation with our metrics and correct with LoRA (Hu et al., 2022).

8 CONCLUSIONS

We provide insight on this cross-vendor mismatch across the whole stack, from the accumulation order inside matrix instructions to expert routing, generated answers, and distilled students. The obvious fixes fall short. FP32 upcasting helps the dense model at three times the runtime, and keeping only the MLPs in BF16 retains almost all of this gain at 1.3 times the runtime, but precision gives the MoE models a mixed result, and output-head LoRA cannot reach the computation where the differences arise. A teacher on a different vendor changes 431 of a student's MMLU answers even when training is exactly reproducible, making hardware a hidden variable in distillation and in comparisons across machines. Cross-vendor agreement should therefore be tested like any other model property, and our five families of measures provide the means, since each of these findings appears in some families and not in others. The results also show where the solution lies. Making the accumulation order explicit and shared across vendors in kernels and matrix instructions would remove a major source of the mismatch.

REPRODUCIBILITY STATEMENT

All four models are public checkpoints, used at fixed revisions with byte-identical weights, configuration, and tokenizer files on the two systems (Appendix B). All prompts come from the public MMLU, ArabicMMLU, and ARC-Challenge benchmarks. Appendix A defines every metric, and Appendix B gives the reference capture, the adapter and distillation training settings, and the precision settings. Intervals use 10,000 paired bootstrap draws over whole questions with a fixed seed. The kernel case study lists its tile sizes, instructions, and complete procedures in Section C.2. We will release the code for reference capture, comparison, adapter training, and distillation.

AI USE STATEMENT

We did not use generative AI tools for problem formulation, solution ideas, the paper story, variable selection, research topic discovery, finding related work, or deciding the paper's structure and content. The authors wrote the paper. Generative AI tools were used to help with experiments and code, to enhance figures, to suggest a title, and to revise the authors' text, including rewording and condensing passages. The authors reviewed and approved every change. LLM-generated code was verified and tested for correctness by the main authors. We take responsibility for the final content of this work, including text, claims, or artifacts produced with the aid of generative AI.

REFERENCES

Sahil Bhola and Karthik Duraisamy. Deterministic and probabilistic rounding error analysis for mixed-precision arithmetic on modern computing units, 2024. URL https://arxiv.org/abs/2411.18747.

Timothée Chauvin, Erwan Le Merrer, François Taïani, and Gilles Trédan. Log probability tracking of LLM apis. *CoRR*, abs/2512.03816, 2025. doi: 10.48550/ARXIV.2512.03816. URL https://doi.org/10.48550/arXiv.2512.03816.

Feilong Chen, Yijiang Liu, Yi Huang, Hao Wang, Miren Tian, Ya-Qi Yu, Minghui Liao, and Jihao Wu. Mindvl: Towards efficient and effective training of multimodal large language models on ascend npus. *CoRR*, abs/2509.11662, 2025. doi: 10.48550/ARXIV.2509.11662. URL https://doi.org/10.48550/arXiv.2509.11662.

Kexin Chu, Yang Zhou, and Wei Zhang. Margingate: Sparse margin-triggered verification for batch-invariant LLM inference. *CoRR*, abs/2605.30218, 2026. doi: 10.48550/ARXIV.2605.30218. URL https://doi.org/10.48550/arXiv.2605.30218.

Peter Clark, Isaac Cowhey, Oren Etzioni, Tushar Khot, Ashish Sabharwal, Carissa Schoenick, and Oyvind Tafjord. Think you have solved question answering? try arc, the AI2 reasoning challenge. *CoRR*, abs/1803.05457, 2018. URL http://arxiv.org/abs/1803.05457.

Liam Cooper, Shinnung Jeong, Hyeran Jeon, Jeffrey Young, and Hyesoon Kim. Accelerating the mitigation of llm inference nondeterminism across gpu architectures, 2026. URL https://arxiv.org/abs/2609.25624.

Massimiliano Fasi, Nicholas J. Higham, Mantas Mikaitis, and Srikara Pranesh. Numerical behavior of NVIDIA tensor cores. *PeerJ Comput. Sci.*, 7:e330, 2021. doi: 10.7717/PEERJ-CS.330. URL https://doi.org/10.7717/peerj-cs.330.

David Goldberg. "what every computer scientist should know about floating-point arithmetic". *ACM Comput. Surv.*, 23(3):413, 1991.

Raja Gond, Aditya K. Kamath, Ramachandran Ramjee, and Ashish Panwar. LLM-42: enabling determinism in LLM inference with verified speculation. *CoRR*, abs/2601.17768, 2026. doi: 10.48550/ARXIV.2601.17768. URL https://doi.org/10.48550/arXiv.2601.17768.

Horace He and Thinking Machines Lab. Defeating nondeterminism in llm inference. *Thinking Machines Lab: Connectionism*, 2025. https://thinkingmachines.ai/blog/defeating-nondeterminism-in-llm-inference/.

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021. URL https://openreview.net/forum?id=d7KBjmI3GmQ.

Geoffrey E. Hinton, Oriol Vinyals, and Jeffrey Dean. Distilling the knowledge in a neural network. *CoRR*, abs/1503.02531, 2015. URL http://arxiv.org/abs/1503.02531.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net, 2022. URL https://openreview.net/forum?id=nZeVKeeFYf9.

Jais Team. Jais 2: A family of arabic-centric open large language models. *CoRR*, abs/2608.13580, 2026. doi: 10.48550/ARXIV.2608.13580. URL https://doi.org/10.48550/arXiv.2608.13580.

Fajri Koto, Haonan Li, Sara Shatnawi, Jad Doughman, Abdelrahman Sadallah, Aisha Alraeesi, Khalid Almubarak, Zaid Alyafeai, Neha Sengupta, Shady Shehata, Nizar Habash, Preslav Nakov, and Timothy Baldwin. ArabicMMLU: Assessing massive multitask language understanding in Arabic. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Findings of the Association for Computational Linguistics: ACL 2024*, pp. 5622–5640, Bangkok, Thailand, August

2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.findings-acl.334. URL https://aclanthology.org/2024.findings-acl.334/.

Xinyi Li, Ang Li, Bo Fang, Katarzyna Swirydowicz, Ignacio Laguna, and Ganesh Gopalakrishnan. FTTN: feature-targeted testing for numerical properties of NVIDIA & AMD matrix accelerators. In *24th IEEE International Symposium on Cluster, Cloud and Internet Computing, CCGrid 2024, Philadelphia, PA, USA, May 6-9, 2024*, pp. 39–46. IEEE, 2024. doi: 10.1109/CCGRID59990.2024.00014. URL https://doi.org/10.1109/CCGrid59990.2024.00014.

Wenhan Ma, Hailin Zhang, Liang Zhao, Yifan Song, Yudong Wang, Zhifang Sui, and Fuli Luo. Stabilizing moe reinforcement learning by aligning training and inference routers. *CoRR*, abs/2510.11370, 2025. doi: 10.48550/ARXIV.2510.11370. URL https://doi.org/10.48550/arXiv.2510.11370.

Niklas Muennighoff, Luca Soldaini, Dirk Groeneveld, Kyle Lo, Jacob Morrison, Sewon Min, Weijia Shi, Evan Pete Walsh, Oyvind Tafjord, Nathan Lambert, Yuling Gu, Shane Arora, Akshita Bhagia, Dustin Schwenk, David Wadden, Alexander Wettig, Binyuan Hui, Tim Dettmers, Douwe Kiela, Ali Farhadi, and et al. Olmoe: Open mixture-of-experts language models. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net, 2025. URL https://openreview.net/forum?id=xXTkbTBmqq.

Olmo Team. Olmo: Accelerating the science of language models. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, pp. 15789–15809. Association for Computational Linguistics, 2024. doi: 10.18653/V1/2024.ACL-LONG.841. URL https://doi.org/10.18653/v1/2024.acl-long.841.

Penghui Qi, Zichen Liu, Xiangxin Zhou, Tianyu Pang, Chao Du, Wee Sun Lee, and Min Lin. Defeating the training-inference mismatch via FP16. *CoRR*, abs/2510.26788, 2025. doi: 10.48550/ARXIV.2510.26788. URL https://doi.org/10.48550/arXiv.2510.26788.

Qwen Team. Qwen1.5-MoE: Matching 7B model performance with 1/3 activated parameters. https://qwenlm.github.io/blog/qwen-moe/, 2024.

Llama Team. The llama 3 herd of models. *CoRR*, abs/2407.21783, 2024. doi: 10.48550/ARXIV.2407.21783. URL https://doi.org/10.48550/arXiv.2407.21783.

Ziteng Yang, Nicholas J. Riasanovsky, Warren Deng, and Vivek Sarkar. Taming bitwise behavior in gpu kernels with tensor core: Black-box reconstruction, compiler enforcement, and static verification, 2026. URL https://arxiv.org/abs/2609.11356.

Jiayi Yuan, Hao Li, Xinheng Ding, Wenya Xie, Yu-Jhe Li, Wentian Zhao, Kun Wan, Jing Shi, Xia Hu, and Zirui Liu. Understanding and mitigating numerical sources of nondeterminism in LLM inference. In Danielle Belgrave, Cheng Zhang, Laura N. Montoya, Hsuan-Tien Lin, Razvan Pascanu, Piotr Koniusz, Marzyeh Ghassemi, Nancy Chen, Iván Vladimir Meza Ruíz, and Arturo Loaiza-Bonilla (eds.), *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2025, NeurIPS 2025, San Diego, CA, USA, December 2-7, 2025 / Mexico City, Mexico, November 30 - December 5, 2025*, 2025. URL http://papers.nips.cc/paper_files/paper/2025/hash/f80094a824ba5912d4a2de169c404a40-Abstract-Conference.html.

Anwar Hossain Zahid, Ignacio Laguna, and Wei Le. Testing GPU numerics: Finding numerical differences between NVIDIA and AMD gpus. In *SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis, Atlanta, GA, USA, November 17-22, 2024*, pp. 547–557. IEEE, 2024. doi: 10.1109/SCW63240.2024.00077. URL https://doi.org/10.1109/SCW63240.2024.00077.

Yaxiang Zhang, Yingru Li, Jiacai Liu, Jiawei Xu, Ziniu Li, Qian Liu, and Haoyuan Li. Beyond precision: Training-inference mismatch is an optimization problem and simple LR scheduling fixes it. *CoRR*, abs/2602.01826, 2026. doi: 10.48550/ARXIV.2602.01826. URL https://doi.org/10.48550/arXiv.2602.01826.

Ziyang Zhang, Xinheng Ding, Jiayi Yuan, Rixin Liu, Huizi Mao, Jiarong Xing, and Zirui Liu. Deterministic inference across tensor parallel sizes that eliminates training-inference mismatch. *CoRR*, abs/2511.17826, 2025. doi: 10.48550/ARXIV.2511.17826. URL https://doi.org/10.48550/arXiv.2511.17826.

Tianle Zhong, Neiwen Ling, Yifan Pi, Zijun Wei, Tianshu Yu, Geoffrey Fox, Peng Wu, and Xiao Yu. Diagnosing training inference mismatch in LLM reinforcement learning. *CoRR*, abs/2605.14220, 2026. doi: 10.48550/ARXIV.2605.14220. URL https://doi.org/10.48550/arXiv.2605.14220.

A METRIC DEFINITIONS

A.1 OUTPUT COMPARISONS

The main text explains the role of each metric family. Here we define how the measurements are calculated. Let $\mathcal{X} = (x_1, \ldots, x_N)$ be the evaluated reference inputs, with each position counted separately. For reference execution R and target execution T, write $z_v(x_i) \in \mathbb{R}^V$ for the logits over a vocabulary of size V and $z_{v,ij} = z_{v,j}(x_i)$ for token index j, with $v \in \{R, T\}$. Let $\mathbf{1}[\cdot]$ be one when its condition holds and zero otherwise. All comparisons use the native outputs, and adaptation replaces z_T with the corrected target output. Agreement counts are divided by their stated denominators to obtain fractions, or multiplied by 100 after division to obtain percentages. For mixture-of-experts (MoE) models, we also define the comparison of router values and selected experts below. Figure 3 shows why several families are needed, using two constructed cases for the same question. Candidate C's logit changes by one ULP in each case. In Case A, lower-ranked candidates exchange positions while B remains the leading token, and both executions answer correctly. In Case B, C overtakes B and the answer becomes incorrect. Bitwise equality flags both cases, while the other families separate them.

Bitwise equality. Let $b_{v,ij}$ be the bit representation of the native 16-bit brain floating-point (BF16) value $z_{v,ij}$. The equal-element count is

$$C_{\text{bit}} = \sum_{i=1}^{N} \sum_{j=1}^{V} \mathbf{1}[b_{T,ij} = b_{R,ij}]. \tag{1}$$

Its fraction is $F_{\text{bit}} = C_{\text{bit}}/(NV)$, and full-tensor equality is $B_{\text{all}} = \mathbf{1}[C_{\text{bit}} = NV]$. This compares bit patterns, so positive and negative zero are distinct. Full-tensor equality requires every compared element to match. This native BF16 measure is not applicable when an execution returns logits in another format, as in the LayerCast comparison.

Logit differences. Raw-logit mean squared error (MSE) measures the average squared difference between vocabulary values,

$$\text{MSE}_z = \frac{1}{NV} \sum_{i=1}^{N} \sum_{j=1}^{V} (z_{T,ij} - z_{R,ij})^2. \tag{2}$$

At temperature one, the corresponding probabilities are $p_{v,ij} = \exp(z_{v,ij}) / \sum_{a=1}^{V} \exp(z_{v,ia})$. Probability MSE compares these distributions coordinate by coordinate,

$$\text{MSE}_p = \frac{1}{NV} \sum_{i=1}^{N} \sum_{j=1}^{V} (p_{T,ij} - p_{R,ij})^2. \tag{3}$$

Forward Kullback–Leibler (KL) divergence compares the target distribution with the reference distribution that supplies the training signal,

$$\text{KL}_{R \to T} = \frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{V} p_{R,ij} \big(\log p_{R,ij} - \log p_{T,ij} \big). \tag{4}$$

The KL sum over the vocabulary is averaged over inputs. MSE also divides by the vocabulary size. We use natural logarithms and compute the probabilities through log-softmax. These quantities respond differently to the same correction. For example, a shared shift $z_T = z_R + c\mathbf{1}$ gives raw-logit MSE of c^2 and leaves both probability measures at zero.

Top-K consistency. Let $g_v(x_i) = \arg\max_j z_{v,ij}$ be the leading token. Let $\pi_v^K(x_i)$ be the ordered tuple returned by the execution's top-K selection operation, and let $S_v^K(x_i)$ contain its token indices.

We compare the leading token, the complete ordered list, and its membership as

$$C_1 = \sum_{i=1}^{N} \mathbf{1}[g_T(x_i) = g_R(x_i)],$$
$$C_{\mathrm{ord},K} = \sum_{i=1}^{N} \mathbf{1}[\pi_T^K(x_i) = \pi_R^K(x_i)], \tag{5}$$
$$C_{\mathrm{set},K} = \sum_{i=1}^{N} \mathbf{1}[S_T^K(x_i) = S_R^K(x_i)].$$

All three counts have denominator N, and the ordered and set comparisons use $K = 5$. Set agreement requires all five members to match. Leading-token selection uses argmax, while the two top-5 comparisons use the native top-K operation. These operations retain their recorded tie handling, as specified in Section B.1.

Token agreement. Same Greedy Token compares the next token at each reference input, giving $C_{\mathrm{greedy,fixed}} = C_1$, the same count as Top-1. To compare generated sequences, let q_m be initial prompt m among M prompts, and let $y_{v,m,t}$ be execution v's token at step t. Before stopping, each execution uses its own inputs, $y_{v,m,t} = g_v(q_m \| y_{v,m,<t})$, where $\|$ denotes concatenation. For rollout agreement in the main table, generation stops at the first end-of-sequence (EOS) token or the response cap $H = 1{,}024$. Let $s_{v,m} = (y_{v,m,1}, \ldots, y_{v,m,L_{v,m}})$ be the resulting token sequence, where $L_{v,m} \leq H$ and an emitted EOS token is retained. We add no padding or synthetic EOS token. The matching-sequence count is

$$C_{\mathrm{greedy},H} = \sum_{m=1}^{M} \mathbf{1}[s_{T,m} = s_{R,m}]. \tag{6}$$

Its denominator is M. Equality requires both sequence lengths and all generated tokens to match. Capped responses are compared over the observed horizon, so this measure does not require both sequences to end in EOS. For comparison, agreement at every valid reference input is $\sum_{m=1}^{M} \prod_{t=1}^{L_{R,m}} \mathbf{1}[g_T(x_{m,t}^R) = g_R(x_{m,t}^R)]$, where $x_{m,t}^R$ follows the saved reference continuation. This comparison uses the reference inputs, whereas equation 6 follows each execution's generated continuation. The reference-input comparison ends at the reference's first EOS token or cap. Each rollout ends at its own first EOS token or cap.

Task accuracy. For a benchmark $\mathcal{B} = \{(q_m, a_m)\}_{m=1}^{M_\mathcal{B}}$, let a_m be the correct answer and $\widehat{a}_v(q_m)$ the answer parsed from execution v's generated response. The correct-answer count is

$$C_{\mathrm{correct},v}^{\mathcal{B}} = \sum_{m=1}^{M_\mathcal{B}} \mathbf{1}[\widehat{a}_v(q_m) = a_m]. \tag{7}$$

The accuracy is $\mathrm{Acc}_v^{\mathcal{B}} = C_{\mathrm{correct},v}^{\mathcal{B}} / M_\mathcal{B}$. A failed parse counts as incorrect. We report each benchmark separately for the unadapted and adapted target, counting questions equally. This measures whether the resulting answers remain correct after the numerical correction.

A.2 Expert routing

For MoE models, let $r_v^{i\ell} \in \mathbb{R}^E$ be the router logits at evaluated position i and layer ℓ, with N_r positions, L layers, and E experts. Write $s_v^{i\ell} = \mathrm{softmax}(r_v^{i\ell})$ at temperature one. Router-logit MSE and router forward KL are

$$\mathrm{MSE}_{\mathrm{router}} = \frac{1}{N_\mathrm{r} L E} \sum_{i=1}^{N_\mathrm{r}} \sum_{\ell=0}^{L-1} \sum_{e=1}^{E} (r_{T,e}^{i\ell} - r_{R,e}^{i\ell})^2, \tag{8}$$

$$\mathrm{KL}_{\mathrm{router}} = \frac{1}{N_\mathrm{r} L} \sum_{i=1}^{N_\mathrm{r}} \sum_{\ell=0}^{L-1} \sum_{e=1}^{E} s_{R,e}^{i\ell} (\log s_{R,e}^{i\ell} - \log s_{T,e}^{i\ell}). \tag{9}$$

Let $a_v^{i\ell}$ be the ordered tuple of K_r expert indices actually selected by the router, and let $S_v^{i\ell}$ be its set of members. The selection counts are

$$C_{\mathrm{route,ord}} = \sum_{i=1}^{N_\mathrm{r}} \sum_{\ell=0}^{L-1} \mathbf{1}[a_T^{i\ell} = a_R^{i\ell}],$$
$$C_{\mathrm{route,set}} = \sum_{i=1}^{N_\mathrm{r}} \sum_{\ell=0}^{L-1} \mathbf{1}[S_T^{i\ell} = S_R^{i\ell}], \tag{10}$$
$$C_{\mathrm{route,all}} = \sum_{i=1}^{N_\mathrm{r}} \prod_{\ell=0}^{L-1} \mathbf{1}[a_T^{i\ell} = a_R^{i\ell}].$$

The first two counts use denominator $N_\mathrm{r}L$, while agreement across all layers uses N_r and requires the ordered selections to match at every layer for one position. We use the router's actual selections, retaining its native tie behavior. For the reference, the probability gap $s_{R,(K_\mathrm{r})}^{i\ell} - s_{R,(K_\mathrm{r}+1)}^{i\ell}$ between the last selected rank and the next rank describes how close those choices are, where parentheses denote descending probability rank. We report its mean and minimum across position-layer pairs. Router values and expert choices describe the mismatch inside the model, alongside the output-level measures.

A.3 Objective-loss definitions

Table 5 gives the four loss formulas. For reference, let $\delta = \hat{z}_T - z_R$ be the remaining logit error, $p_R = \mathrm{softmax}(z_R)$, and $\hat{p}_T = \mathrm{softmax}(\hat{z}_T)$ at temperature one. Let $q_1, \ldots, q_5$ be the five highest-scoring reference tokens, ordered by decreasing score with ties broken by ascending token index, and let $Q_5 = \{q_1, \ldots, q_5\}$. This appendix records the normalization used for the weighting experiments.

For N questions with m_i valid positions for question i, the fixed normalizer for component j is its initial mean over valid training positions, computed from the unadapted target and reference outputs. In full notation, $c_j = (\sum_i m_i)^{-1} \sum_i \sum_{t=1}^{m_i} \ell_j(z_{T,it}^{(0)}, z_{R,it})$. The same c_j values are used for every weighting profile.

Table 5: Individual loss formulas for one input, before normalization and weighting.

Loss	Formula
Raw-logit MSE	$\ell_{\mathrm{MSE}} = \frac{1}{V}\sum_{j=1}^{V} \delta_j^2$
Forward KL	$\ell_{\mathrm{KL}} = \sum_{j=1}^{V} p_{R,j} \log \frac{p_{R,j}}{\hat{p}_{T,j}}$
Token-gap	$\ell_{\mathrm{token}} = \frac{1}{V-1}\sum_{j \neq q_1} (\delta_{q_1} - \delta_j)^2$
Top-5-gap	$\ell_{\mathrm{top5}} = \frac{1}{5}\left[\sum_{i=1}^{4} (\delta_{q_i} - \delta_{q_{i+1}})^2 + \frac{1}{V-5}\sum_{j \notin Q_5} (\delta_{q_5} - \delta_j)^2\right]$

B Experimental details

B.1 Setup

Both models run with identical weights, tokenizer, and input tokens on the NVIDIA A100 and the AMD MI210, in BF16 with eager attention on one device, batch size one, no key-value cache during evaluation, and with mixed precision, tensor-float-32 arithmetic, and deterministic modes disabled. Both systems use PyTorch 2.7.1 and Transformers 5.13.0, with CUDA 12.8 on the NVIDIA A100 and ROCm 6.3.3 on the AMD MI210. Jais is a dense model with 32 blocks and a vocabulary of 150,272 tokens, and OLMoE has 16 blocks that each select 8 of 64 experts, with a vocabulary of

Table 6: Training hyperparameters. The normalizers c_j are the mean unadapted losses over each model's training positions. *First 128 tokens of each response.

	Jais head LoRA	OLMoE head LoRA	Distillation
Trained weights	head, $r = 32$, $\alpha = 64$	head, $r = 32$, $\alpha = 64$	all student weights
Optimizer	AdamW, 10^{-4}, $(0.9, 0.999)$	AdamW, 10^{-4}, $(0.9, 0.999)$	AdamW, 10^{-5}, $(0.9, 0.95)$
Weight decay	0	0	0
Batch size	1	1	8
Training responses	1,200	512*	512
Epochs	2	2	2
Schedule	constant	constant	cosine, 8 warmup
Gradient clipping	none	none	1.0
c_{MSE} / c_{KL}	0.266 / 0.00814	0.00763 / 0.000638	–
c_{token} / c_{top5}	2.83 / 0.846	0.0181 / 0.0157	–

Table 7: Jais task accuracy without the questions used for adapter training or development (13,182 MMLU and 13,602 ArabicMMLU questions). Parser failures count as incorrect.

Method	MMLU (%) ↑	ArabicMMLU (%) ↑
Native AMD (BF16)	61.05	71.56
Head LoRA (KL)	61.06	71.56
Head LoRA weighting profiles		
Equal	61.05	71.55
MSE-heavy	61.07	71.56
KL-heavy	61.06	71.55
Token-heavy	61.08	71.56
Top-5-heavy	61.12	71.53
Numerical-heavy	61.04	71.55
Decision-heavy	61.09	71.56

50,304 tokens. Questions come from the public MMLU, ArabicMMLU, and ARC-Challenge (Clark et al., 2018) partitions, with duplicates removed and development questions chosen before a disjoint training set. Jais is a chat model, so each prompt asks for one answer letter and each prefix is evaluated by its own forward pass. OLMoE is a base model whose responses all reach the 1,024-token cap, so each response is evaluated in one teacher-forced forward that records the logits and router outputs at every position. Repeating a run on the same vendor reproduces every logit bit for bit, as the audited repeat of all 200 Jais development responses on AMD and the repeated NVIDIA-teacher student show, so the cross-vendor differences do not come from run-to-run variation.

B.2 Output-head LoRA training

We place the adapter at the output head because it performed best among the placements we tried, which also included full-model LoRA and adapters on the final down-projection and the final MLP. Table 6 lists the hyperparameters. Each proposed update is tried at step fractions from 1 down to 1/64 and kept only if it lowers the objective, and each response is weighted by its number of valid positions so that every token counts equally. The normalizers c_j of Section 3.2 are computed once from the unadapted target and shared by every weighting profile. OLMoE adapters use the same recipe on the first 128 positions of each training response.

B.3 Distillation targets

The student learns from each teacher's 64 most likely tokens, renormalized. Against the NVIDIA teacher, these targets share on average 62.1 of 64 tokens for the Native AMD (BF16) teacher, 62.0 for the FP32 teacher, and 61.9 for the FP16 teacher, with a total-variation distance of about 0.002 in each case. The FP32 teacher's training targets are therefore as close to the NVIDIA targets as the native ones, so the selection of the top 64 tokens does not explain why its student moves further.

B.4 Precision and cost

FP32 upcasting follows LayerCast (Yuan et al., 2025) and keeps the linear weights in BF16 while computing in FP32, and FP16 casts the checkpoint to FP16 for all computation. Runtime is wall-clock time including model loading, and memory is peak allocated GPU memory, both measured over the complete benchmark and development workload on one MI210. Jais FP16 produced non-finite outputs, so we report FP16 only for OLMoE. FP32 upcasting produces FP32 logits while the reference logits are BF16, so part of any measured difference could come from the reference's own rounding. Rounding the Jais FP32 logits to BF16 before the comparison leaves the gain almost unchanged, with raw-logit MSE -42.5% and forward KL -33.7% relative to Native AMD (BF16). For OLMoE on 32 development responses, rounding leaves the rise in raw-logit MSE unchanged ($+13.5\%$ for FP32 upcasting and $+20.7\%$ for FP16) and turns the small KL reductions into increases ($+5.3\%$ and $+11.5\%$). Once both sides are compared in BF16, the finer formats therefore move OLMoE further from the reference on both measures.

C Extra Details of Where the Mismatch Arises

C.1 Expert routing by layer in OLMoE

Figure 6 gives the router agreement of Section 5.2 layer by layer, comparing the eight experts that AMD and the reference select at each layer and position under the same input. With Native AMD (BF16), agreement starts at 99.5% in the first layer and falls with depth to 82–85% in the last three layers. Both finer formats start lower, at 92.9% (FP32 upcasting) and 92.6% (FP16), and stay below Native AMD (BF16) at every layer, so the loss begins at the first router and persists through the model.

We also ask whether these disagreements are tie-breaking artifacts. On the first 32 development responses, we recompute the AMD router values and measure the reference gap between its 8th and 9th experts. Only 2.8% of first-layer positions have a gap of at most one BF16 step, yet 65% of the Native AMD (BF16) disagreements fall there, so native disagreements are mostly near-ties broken differently. Under FP32 upcasting and FP16, only 20% of the much more frequent first-layer disagreements fall at such near-ties. The finer formats therefore change the router inputs by more than tie-breaking, which matches their larger logit error.

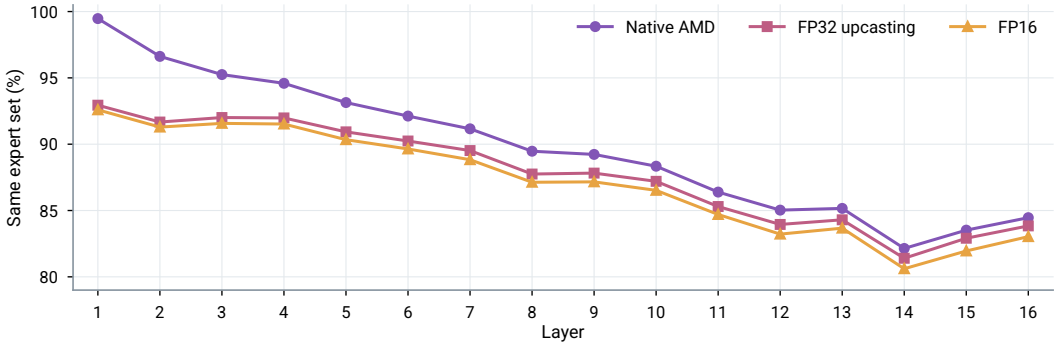


Figure 6: Share of the 131,072 OLMoE development positions at which AMD selects the same eight experts as the NVIDIA reference, by layer. Native AMD (BF16) matches the reference almost exactly at the first layer, while FP32 upcasting and FP16 lose 6.5 and 6.9 percentage points there, and remain below Native AMD (BF16) at every layer.

C.2 Kernel and matrix-instruction case study

This case study supports the accumulation-order account in Section 3.1. It uses a separate, smaller workload, so it illustrates the mechanism rather than establishing the cause of the model differences in Section 5.

Workload. We take the layer-0 query projection of SmolLM2-135M-Instruct ($M = 27$, $N = 576$, $K = 576$) with BF16 inputs and weights and FP32 accumulation and output, on the A100 and the MI210. Each comparison uses the same captured inputs and weights, and is repeated with synthetic inputs. Table 1 in Section 2 counts unequal FP32 outputs among 15,552 for three implementations, namely vendor general matrix multiplication (GEMM), a shared Triton kernel with a fixed tile order ($32 \times 64 \times 32$ tiles, four warps, two stages, no autotuning, no split reduction), and a scalar Triton kernel that accumulates sequentially over K. On AMD, the FP32-output GEMM request falls back from hipBLASLt to another library path.

Results. All three implementations are repeatable within each vendor. Vendor GEMM and the ordered-tile kernel differ across vendors, by at most 3.43×10^{-5} and 4.58×10^{-5}. The scalar kernel agrees exactly on both inputs. Fixing the outer tile order is therefore not enough, and the remaining difference lies inside the matrix instructions that compute each tile. The compiled MI210 ordered-tile kernel contains the matrix fused multiply-add (MFMA) instruction `v_mfma_f32_32x32x8bf16_1k` (32 static occurrences) and reproduces all 15,552 saved outputs bit for bit. The first-tile rows of Table 1 evaluate only the first 27×64 output tile over the reduction range $[0, 32)$, as separate one-stage kernels, and show that differences already appear there. Algorithm 1 gives the complete procedures.

Algorithm 1 Complete matrix-multiplication procedures

Given BF16 matrices $X \in \mathbb{R}^{M \times K}$ and $W \in \mathbb{R}^{N \times K}$, each path returns an FP32 output $Y \in \mathbb{R}^{M \times N}$. The explicit kernels use FP32 accumulators. For ordered-tile matmul, $B_M = 32$, $B_N = 64$, $B_K = 32$, and $T = \lceil K/B_K \rceil$. Partition X, W into zero-padded tiles $X_{it} \in \mathbb{R}^{B_M \times B_K}$ and $W_{jt} \in \mathbb{R}^{B_N \times B_K}$. Output writes retain only indices inside Y.

(a) Vendor GEMM

$Y \leftarrow \mathrm{VendorGEMM}(X, W^\top)$
return Y

The library selects its own implementation and reduction order.

(b) Ordered-tile matmul

for all output tiles (i, j) **do**
  $C \leftarrow 0_{B_M \times B_N}$ in FP32.
  for $t = 1, \ldots, T$ **do**
    Load $A \leftarrow X_{it}$, $B \leftarrow W_{jt}$.
    $C \leftarrow \mathrm{dot}(A, B^\top, C)$
  end for
  Write $Y_{ij} \leftarrow C$.
end for
return Y

(c) Scalar matmul

for all output elements (i, j) **do**
  Initialize $c = 0$ in FP32.
  for $k = 1, \ldots, K$ **do**
    $x \leftarrow \mathrm{FP32}(X_{ik})$
    $w \leftarrow \mathrm{FP32}(W_{jk})$
    $c \leftarrow c + xw$
  end for
  Write $Y_{ij} \leftarrow c$.
end for
return Y

Instruction probes. Table 8 lists the instructions in two controlled probes of a logical $32 \times 32 \times 16$ operation. On the MI210, two nonzero products of 2^{-24} added to an FP32 accumulator of 1 give $1 + 2^{-23}$ when placed at reduction positions 0 and 1, but 1 when placed at positions 0 and 8, one ULP apart. Where a contribution sits within the instruction sequence thus changes the result. On the A100, computing the same operation with width-eight or width-sixteen instructions gives identical outputs on all 28 registered examples and across 100 repetitions, so reduction width alone does not change the A100 result here.

Table 8: Matrix instructions in the controlled probes, for the complete logical $32 \times 32 \times 16$ operation. Both PTX source names end in `.row.col.f32.bf16.bf16.f32`.

Path	Source instruction	Machine instruction	Count
A100, $K = 16$	`mma.sync.aligned.m16n8k16`	`HMMA.16816.F32.BF16`	8
A100, $K = 8 + 8$	`mma.sync.aligned.m16n8k8`	`HMMA.1688.F32.BF16`	16
MI210, $K = 8 + 8$	`v_mfma_f32_32x32x8bf16_1k`	`v_mfma_f32_32x32x8bf16_1k`	2

D ADDITIONAL INTERVENTION RESULTS

Figure 7 summarizes every intervention against Native AMD (BF16) and every distilled student against the NVIDIA-teacher student.

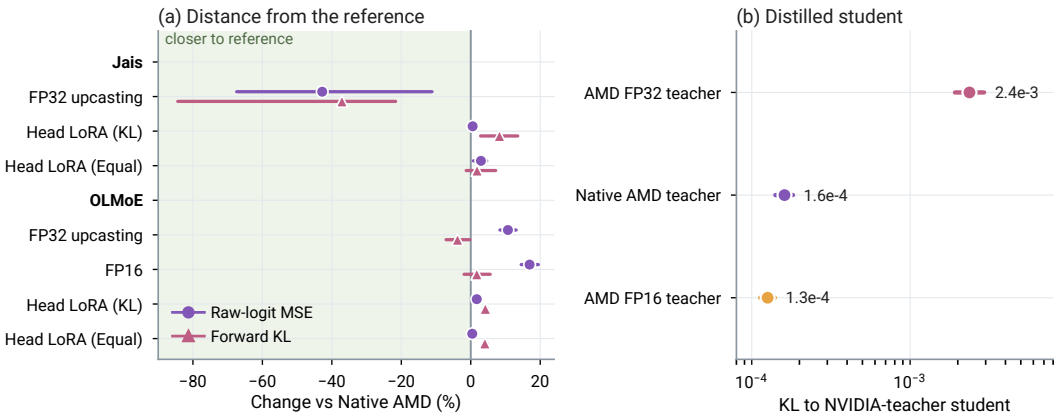


Figure 7: (a) Change in raw-logit MSE and forward KL relative to Native AMD (BF16). Values left of zero are closer to the NVIDIA reference, and Jais FP16 is omitted because it produces nonfinite outputs. Only FP32 upcasting on Jais moves toward the reference on both measures, and every other intervention moves away on at least one. (b) Forward KL of each distilled student from the NVIDIA-teacher student.

Table 9: Complete output comparison for Table 2. Choices use 2,498 positions and 200 rollouts for Jais and 131,072 positions and 128 rollouts of 1,024 tokens for OLMoE. The greedy next token on the reference input equals Top-1 and is not repeated. Jais accuracy parses generated answers on 13,890 MMLU and 14,303 ArabicMMLU questions, including the adapter's training questions (Table 7 excludes them), and OLMoE accuracy scores answer-letter logits on the full MMLU and ARC-Challenge test sets. – marks non-BF16 outputs.

	Numerical values				Choices (%) ↑				Accuracy (%) ↑	
Method	Bit-eq. (%) ↑	Logit MSE ↓	Prob. MSE (10^{-7}) ↓	KL (10^{-3}) ↓	Top-1	Ord. top-5	Top-5 set	Rollout	MMLU	AMMLU/ ARC
Jais										
Native AMD (BF16)	9.26	0.2873	0.279	7.08	98.76	66.53	83.91	93.50	60.86	71.36
FP16				nonfinite outputs						
FP32 upcasting	–	0.1644	0.186	4.45	98.68	71.58	86.55	93.00	61.00	71.40
Head LoRA (KL)	9.22	0.2888	0.298	7.67	98.52	65.89	83.23	92.50	60.88	71.35
Head LoRA (Equal)	8.65	0.2957	0.278	7.21	98.64	66.37	83.23	93.50	60.87	71.34
OLMoE										
Native AMD (BF16)	14.17	0.0215	0.00833	0.194	99.85	69.57	85.10	34.38	47.37	54.10
FP16	–	0.0251	0.00796	0.198	99.84	66.66	83.42	35.16	47.46	54.10
FP32 upcasting	–	0.0238	0.00759	0.187	99.84	67.75	84.04	33.59	47.44	54.69
Head LoRA (KL)	13.03	0.0219	0.00863	0.202	99.85	69.00	84.87	33.59	47.37	54.10
Head LoRA (Equal)	13.92	0.0216	0.00866	0.202	99.85	69.04	84.83	33.59	47.37	54.10

D.1 Complete output comparison

Table 9 extends Table 2 with probability mean squared error (MSE), top-1 agreement, and top-5 set agreement.

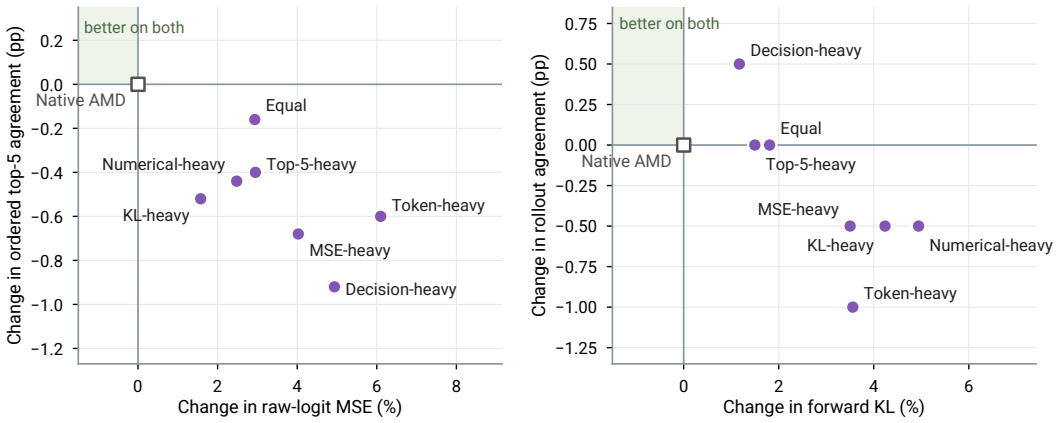


Figure 8: Effect of loss weighting on Jais at the fixed training endpoint, relative to Native AMD (BF16) (open square). The left panel plots the change in raw-logit mean squared error (MSE) against change in ordered top-5 agreement over 2,498 positions. The right panel plots the change in forward Kullback–Leibler (KL) divergence against change in rollout agreement over 200 responses, where one response is 0.5 percentage points. The shaded region would improve both measures. No weighting profile reaches it in the left panel, and only Decision-heavy gains a response on the right while still increasing KL.

D.2 Loss weighting

A single weighting may emphasize the wrong aspect of agreement, so we train seven profiles that change only the loss coefficients. Table 10 lists the seven weighting profiles, Figure 8 places their numerical and choice agreement together, and Table 11 gives all seven profiles on the measures of the main table. No profile lowers raw-logit MSE or forward KL below Native AMD (BF16). KL-heavy has the lowest MSE among them, Numerical-heavy lowers MSE relative to Equal while raising KL, and Decision-heavy gains one rollout while losing ordered top-5 agreement. A profile's name therefore describes what training emphasizes rather than what improves. On OLMoE, every profile raises both MSE ($+0.25$ to $+0.65\%$) and KL ($+2.8$ to $+5.3\%$), with all paired intervals above zero, and matches 486 to 1,211 fewer ordered top-5 lists. MSE-heavy gains a rollout (45 of 128, against 44) while its numerical error grows. Every OLMoE profile ends above Native AMD (BF16) on MSE and KL, leaves expert selection unchanged, and moves benchmark counts by at most one answer.

Table 10: Loss weights after normalization.

Profile	MSE	KL	Token	Top-5
Equal	0.25	0.25	0.25	0.25
MSE-heavy	0.70	0.10	0.10	0.10
KL-heavy	0.10	0.70	0.10	0.10
Token-heavy	0.10	0.10	0.70	0.10
Top-5-heavy	0.10	0.10	0.10	0.70
Numerical-heavy	0.40	0.40	0.10	0.10
Decision-heavy	0.10	0.10	0.40	0.40

D.3 Capacity of an output-head correction

Before training any adapter, we ask how much of the Jais mismatch a shared linear correction at the output head could represent at all. Let $H \in \mathbb{R}^{N\times d}$ hold the AMD hidden values entering the head and $E \in \mathbb{R}^{N\times V}$ the NVIDIA-minus-AMD logit residuals at the same $N = 19{,}316$ training positions, with $d = 3{,}328$ and $V = 150{,}272$. A rank-r head correction has the form HW with $\mathrm{rank}(W) \le r$. Table 12 gives the fraction of $\|E\|_F^2$ that the best such map removes. Even an ideal rank-32 map removes 11.9%, and using every hidden direction removes only 26.1%. These are in-sample fits, and a regression on pure noise with all 3,328 directions would already fit about 17% (d/N), so the out-of-fold result below is the relevant one.

Table 12: Share of squared Jais logit mismatch a linear head map can remove, same inputs.

Map	Removed
Rank 32	11.87%
Any rank (all directions)	26.09%
Outside any linear map	73.91%

With $G = H^\top H = U\,\mathrm{diag}(\lambda)U^\top$ and $R = H^\top E$, we form $K = \mathrm{diag}(\lambda)^{-1/2}U^\top R$ over the retained directions and $T = KK^\top$. The top r eigenvalues of T divided by $\|E\|_F^2$ give the rank-

Table 11: Loss-weighting comparison at the fixed training endpoint. Positions, rollouts, and accuracy sets are as in Table 9, and the last column is ArabicMMLU for Jais and ARC-Challenge for OLMoE. Jais accuracy includes the adapter's training questions (Table 7 excludes them).

Method	Bit-eq. (%) ↑	Logit MSE ↓	KL (10^{-3}) ↓	Ord. top-5 (%) ↑	Rollout (%) ↑	MMLU (%) ↑	AMMLU/ ARC (%) ↑
Jais							
Native AMD (BF16)	9.26	0.2873	7.08	66.53	93.50	60.86	71.36
Equal	8.65	0.2957	7.21	66.37	93.50	60.87	71.34
MSE-heavy	8.72	0.2988	7.33	65.85	93.00	60.89	71.36
KL-heavy	8.95	0.2918	7.38	66.01	93.00	60.89	71.34
Token-heavy	8.07	0.3048	7.33	65.93	92.50	60.89	71.36
Top-5-heavy	8.65	0.2957	7.19	66.13	93.50	60.96	71.32
Numerical-heavy	8.75	0.2944	7.43	66.09	93.00	60.86	71.34
Decision-heavy	8.37	0.3014	7.16	65.61	94.00	60.91	71.35
OLMoE							
Native AMD (BF16)	14.17	0.02149	0.194	69.57	34.38	47.37	54.10
Equal	13.92	0.02158	0.202	69.04	33.59	47.37	54.10
MSE-heavy	14.02	0.02154	0.200	69.20	35.16	47.37	54.10
KL-heavy	13.77	0.02163	0.202	69.08	32.81	47.37	54.10
Token-heavy	13.96	0.02157	0.203	68.96	32.81	47.37	54.10
Top-5-heavy	13.88	0.02160	0.205	68.65	32.81	47.37	54.10
Numerical-heavy	13.93	0.02158	0.201	69.14	32.81	47.37	54.10
Decision-heavy	13.92	0.02159	0.204	68.86	34.38	47.37	54.10

r fraction, and its trace gives the all-direction fraction. All sums use FP64, positions have equal weight, and no intercept is used. The cutoff $\tau = \max(N, d)\,\epsilon_{64}\,\lambda_{\max}(G)$ keeps 3,327 of 3,328 directions, and keeping all of them changes both fractions by less than 0.01 percentage points.

These are upper bounds on the same inputs rather than results of a trained adapter. On new prompts the representable part does not transfer either, as regularized rank-32 maps fitted on disjoint folds increase the pooled out-of-fold squared error, by 0.81% in the best setting. The analysis does not establish the rank of E itself, since a residual can have concentrated structure that the hidden value still cannot predict.

E Additional experiments

E.1 Two further models

Output-head LoRA fails in both Jais and OLMoE, so these experiments focus on the precision interventions. We repeat the OLMoE protocol on a second dense model, Llama-3.1-8B-Instruct (Team, 2024), and a second MoE model, Qwen1.5-MoE-A2.7B-Chat (Qwen Team, 2024), which routes each token to four of 60 experts. Both use the same 128 development responses with the 1,024-token cap, the NVIDIA A100 BF16 reference, and the AMD MI210 target with byte-identical weights. Logit measures use 32 of these responses at every position, and rollouts use all 128. Table 13 shows that both models reproduce the pattern of the main study. The MoE model again has the larger mismatch, as only one of its 128 responses matches the reference and the four routed experts agree at every layer for only 16.9% of positions. FP32 upcasting lowers the logit error of Llama but makes fewer of its responses identical, and in Qwen1.5-MoE both finer formats move every logit and router measure away from the reference, as they do in OLMoE. Accuracy stays within a few dozen answers across formats and vendors in both models, so the change in behavior again remains invisible to accuracy.

E.2 Matched precision on both vendors

The precision interventions of Section 5.2 change the format on AMD alone, so the target no longer rounds the way the BF16 reference does. We therefore also run the NVIDIA reference forward in

Table 13: Two further models on AMD against the NVIDIA BF16 reference. Router is the share of positions and layers where the four routed experts match. Accuracy is on the full MMLU and ARC-Challenge test sets. – marks non-BF16 outputs, a dense model, or the reference itself.

Method	Bit-equal (%)↑	Logit MSE↓	KL (10^{-3})↓	Top-5 (%)↑	Router (%)↑	Rollout (%)↑	MMLU (%)	ARC (%)
Llama-3.1-8B-Instruct (dense)								
NVIDIA reference	–	–	–	–	–	–	66.93	83.45
Native AMD (BF16)	**9.52**	0.00533	0.218	80.27	–	**16.41**	67.04	83.62
FP32 upcasting	–	**0.00449**	**0.130**	**81.00**	–	8.59	67.06	83.45
FP16	–	0.00586	0.151	79.33	–	10.94	67.06	83.53
Qwen1.5-MoE-A2.7B-Chat (MoE)								
NVIDIA reference	–	–	–	–	–	–	60.48	76.71
Native AMD (BF16)	**5.64**	**0.01519**	**1.66**	**66.75**	**87.93**	0.78	60.74	76.54
FP32 upcasting	–	0.01772	2.01	63.89	85.65	**2.34**	60.84	76.62
FP16	–	0.01991	2.30	62.22	84.77	**2.34**	60.92	76.79

FP32 upcasting and in FP16 over the same OLMoE reference responses and compare each AMD format with the NVIDIA run of the same format. Table 14 shows the result on 32 development responses, with rollouts over all 128. With FP32 on both vendors, the outputs agree almost exactly, and 98.4% of the responses become identical. FP16 on both vendors lies between this and the BF16 pair. The mismatch in the main study is therefore largely a property of BF16 execution, and a shared higher-precision format removes most of it at the cost measured in Table 3.

Table 14: OLMoE with the same or a different format on each vendor. Router is the share of positions and layers where the eight selected experts match.

AMD	NVIDIA	Logit MSE (10^{-3})↓	KL (10^{-4})↓	Top-5 (%)↑	Router (%)↑	Rollout (%)↑
BF16	BF16	18.9	2.17	70.69	90.40	34.38
FP16	BF16	22.6	2.15	68.14	87.87	35.16
FP32 upcasting	BF16	21.4	2.06	69.03	88.38	33.59
FP16	FP16	14.9	0.882	78.40	93.25	60.16
FP32 upcasting	FP32 upcasting	**0.086**	**0.0019**	**99.85**	**99.97**	**98.44**

E.3 Selective FP32 in Jais

FP32 upcasting of the whole Jais forward pass gives the largest numerical gain in the main study but triples the runtime. We therefore compute only chosen modules in FP32, casting the BF16 weights losslessly, and evaluate on the same development responses and reference. Table 15 and Figure 9 show that the gain depends strongly on where the higher precision acts. When each module rounds its output back to BF16, only LayerNorm and attention lower both MSE and KL, the output head changes almost nothing, and the MLPs raise KL while being the most expensive module type to upcast. Combining LayerNorm and attention keeps more than half of the full KL reduction, and adding the output head or the last eight blocks adds little, because the residual stream between the modules is still rounded to BF16.

We therefore keep everything in FP32 except the MLPs, including the residual stream, and report this setting as selective FP32 in Table 2. It recovers more than 90% of the full reduction in raw-logit MSE and forward KL, matches almost as many ordered top-5 lists, and keeps the most responses identical to the reference, at 1.30 times the forward time of Native AMD (BF16) against 3.02 times for full upcasting. Its MMLU and ArabicMMLU accuracy stays within 16 answers of Native AMD (BF16). The residual stream therefore carries most of the rounding that full upcasting removes, while the MLPs can stay in BF16. For settings whose output head runs in FP32, Table 15 counts bit-equal logits after rounding them to BF16. The FP32 output head alone then reproduces the Native AMD (BF16) logits exactly, so the head's own arithmetic adds nothing to the mismatch.

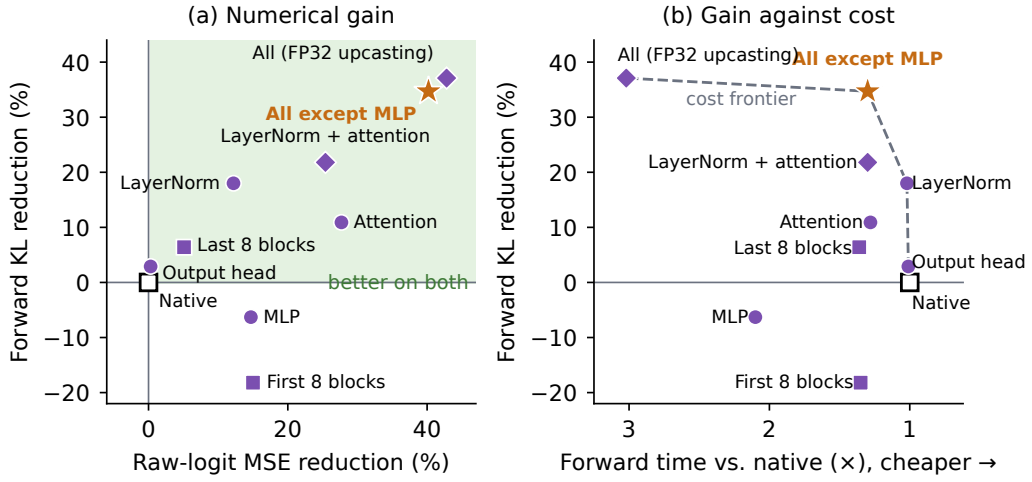


Figure 9: Selective FP32 settings on Jais, relative to Native AMD (BF16) (open square). (a) Reduction in raw-logit MSE and forward KL, where the shaded region improves both. (b) KL reduction against forward time, with cheaper settings to the right. The dashed line joins the settings that no cheaper setting beats on KL. Keeping everything except the MLPs in FP32 (star) lies on this frontier near full upcasting at 1.30 times the forward time.

Table 15: Jais with FP32 computation in selected modules

FP32 modules	Bit-equal (%)↑	Logit MSE↓	KL (10^{-3})↓	Top-5 (%)↑	Rollout (%)↑	Time (×)↓
Native AMD (BF16)	9.26	0.2873	7.08	66.53	93.50	**1.00**
Single module type						
Output head	9.26	0.2865	6.87	66.21	93.50	1.01
LayerNorm	9.20	0.2522	5.81	66.89	93.50	1.02
Attention	10.00	0.2077	6.31	68.73	91.50	1.28
MLP	9.31	0.2449	7.52	66.57	91.00	2.10
Depth slice						
First 8 blocks	9.40	0.2441	8.37	67.25	91.00	1.35
Last 8 blocks	9.59	0.2726	6.62	67.45	93.50	1.36
Combination						
LayerNorm + attention	10.01	0.2142	5.54	67.73	90.50	1.30
LayerNorm + attention + head	10.01	0.2135	5.46	67.77	92.00	1.31
LayerNorm + attention + last 8	10.12	0.2096	5.80	68.94	92.00	1.93
All except MLP (selective FP32)	11.07	0.1717	4.62	71.38	**94.00**	1.30
All (FP32 upcasting)	**11.24**	**0.1644**	**4.45**	**71.58**	93.00	3.02

F LIMITATIONS

We compare one pair of accelerators, the NVIDIA A100 and the AMD MI210, under one software stack with PyTorch 2.7.1, CUDA 12.8, and ROCm 6.3.3. Other GPU generations, vendor libraries, attention kernels, and batch sizes change the kernels that run and can change the size of the mismatch. The exact magnitudes we report therefore apply to this configuration, while the mechanism and its propagation through the model do not depend on it.

We study four open models of up to 14 billion parameters on multiple-choice benchmarks. Longer free-form generation, larger models, and other task types may accumulate more mismatch than we observe. We also treat the NVIDIA execution as an operational reference and do not compare either vendor with a higher-precision ground truth.