samf.sh/talks/2026/08/03

Pre-Training LLMs on a Supercomputer

The most model per GPU-hour: what works, what breaks, and what it costs.

Sam Foreman, Argonne National Laboratory

2026-08-03 · ATPESC 2026

Roadmap

One goal, two halves: **get the most model per GPU-hour.** Part 1 spends the compute well; Part 2 keeps a crash from wasting it.

Part 1: getting a run off the ground *(now, ~30 min)*

- Data preparation
- Python environments + Lustre at scale
- Parallelism: enough to launch

Part 2: keeping it alive to the end *(after the break, ~60 min)*

- Critical batch size + second-order optimizers
- Failures, shared filesystems, checkpointing
- Fault-tolerant training + automatic restarts

Who this is for

One goal runs through all of it: **the best model you can get per GPU-hour.** Every choice ahead trades compute for model quality; the job is to spend it well.

- You know HPC. You may be new to training LLMs at scale.
- This is **lessons-learned**, not a survey: do-this-not-that from real runs on Aurora / Polaris.
- Builds on today's earlier talks: **Jane** just did LLM basics (tokenization, training objectives); **Bethany** covered the parallelisms; **Nathan**, profiling. We assume those and focus on **making it survive at full-machine scale.**

The stack

Three repos, one job each. No `cuda` in user code.

- 🍋 saforem2/`ezpz` · launch, anywhere
- 🧠 saforem2/`torchtitan@ezpz` · train (XPU fork: FSDP · TP · PP · EP · MoE)
- 📚 zhenghh04/`blendcorpus` · data (weighted blend + shard)

> **Hardware agnostic** x
> Write it once on Aurora's XPUs, run it anywhere.

Data prep is the first wall

Not a download. A distributed-systems problem, before the first GPU step.

- **Curate · blend · shard** 2T+ tokens, many corpora
- **Reproducible** every rank, every run
- **Survivable** flaky FS · node failures · noisy neighbors

> **The three steps, in order** x
> **tokenize** (text → shards) → **blend** (weighted mix) → **pin** (make it reproducible)

From raw corpus to tokens

Raw text → **tokenized, sharded binaries** before you can blend.

- **Pin the tokenizer** (SentencePiece, vocab 32k) → or every shard is stale
- **Tokenize in parallel** → EOD per doc, fixed-size `.bin` shards + file list
- **Decide doc packing** up front → it changes the loss
- **Bottleneck is the filesystem**, not the tokenizer

> **Tokenize once, reuse forever** x
> Re-tokenizing 2T tokens is a whole-allocation tax.

Blending data, efficiently

Sample **every batch** to a fixed mixture. Do not concatenate.

- **Concatenation forgets:** order becomes a hyperparameter
- **Domain weights** honored per batch, deterministically
- BlendCorpus : aggregate → sample → index

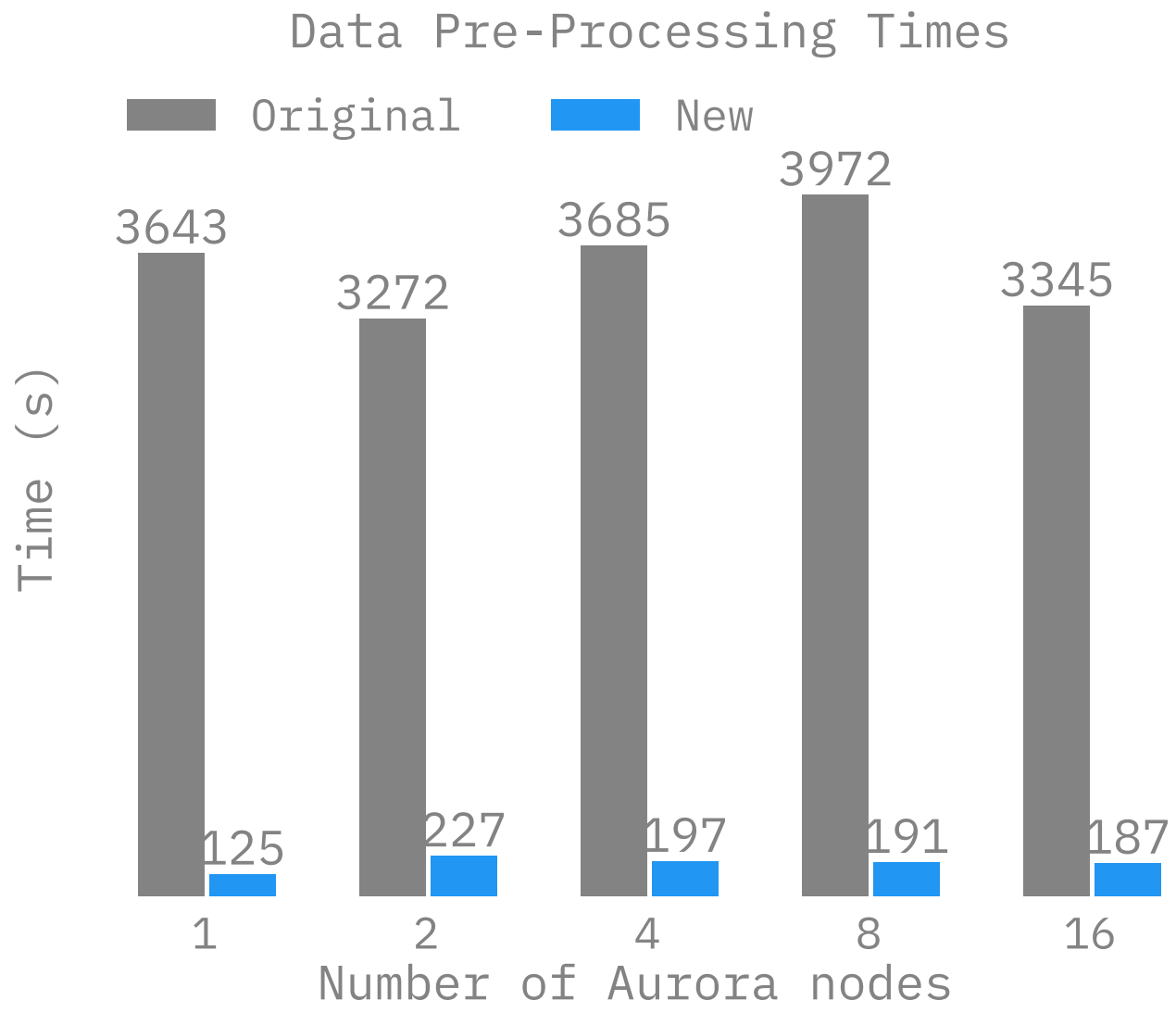


Fig: *Index-build time for 2T tokens: serial ~1 hr vs distributed ~2 min (**30x**).*

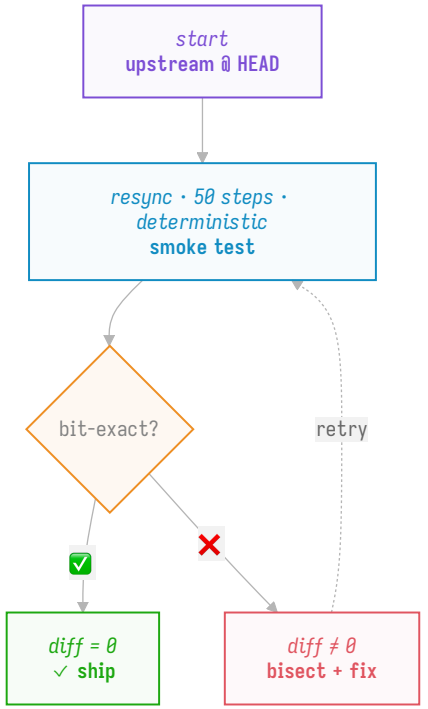
Reproducibility and the fork tax

Same seed + shards + weights → same batch, every rank, every run.

Universal:
- **data churn:** re-scraped shards break determinism
- **pin everything:** SHAs · tokenizer · shard lists · weights

If you fork upstream (we fork `torchtitan` for XPU):
- **upstream churn:** 46 syncs in 7 weeks, any can shift RNG

Every **fork sync** runs a deterministic smoke test first:



Bit-exact = **loss + grad-norm + peak memory** match baseline.

A Python environment that works

Don't build from scratch. **Layer on the site module:** it already has torch, MPI, and the GPU stack built for this machine.

```
# 1. the site's prebuilt module (torch / MPI / GPU)
module load frameworks
# 2. a venv on top -- your packages, inheriting the module's torch
uv venv --system-site-packages --python python3 .venv
source .venv/bin/activate
uv pip install "git+https://github.com/saforem2/ezpz"   # resolves in seconds
```

> **The rule:**
> `venv` first, clone last
> - `--system-site-packages` is what lets the venv **see the module's torch** (skip it and you pull a fresh CPU-only wheel)
> - never `pip install` into base; clone conda `base` **only** if a package hard-requires conda (slow, huge)

Why `import` melts Lustre at scale

One `import torch` = thousands of `stat()` / `open()`. Now × 50,000 ranks.

- **Metadata server chokes**, not bandwidth
- Laptop-seconds → **cluster-minutes** before step 1
- Per-file `rsync` past 256 nodes: **1-2 hours**

> **Python is pathologically small-file** x
> The fix is not a faster FS. It is not hitting the FS from every rank.

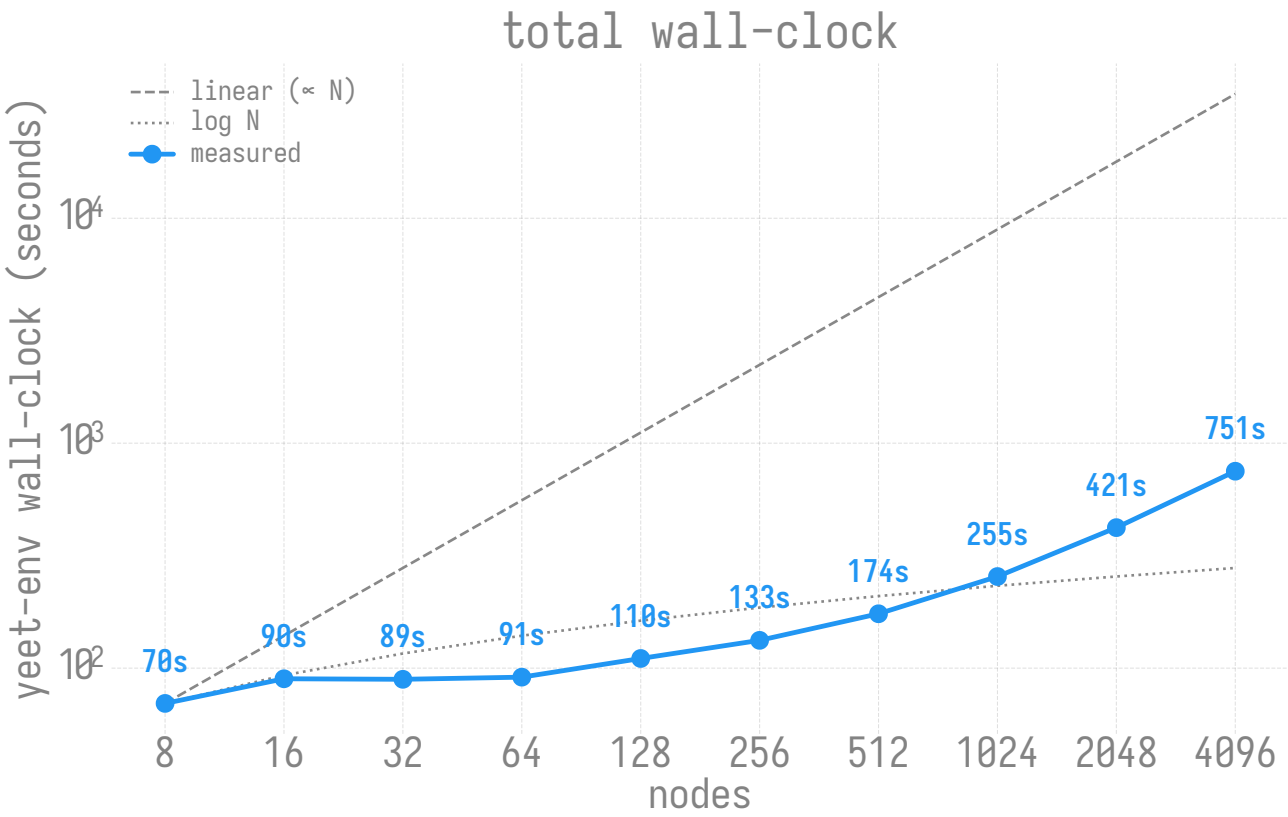
Deep dive: Running 50k Python processes on Aurora with `ezpz yeet`

`ezpz yeet`: broadcast the environment

Copy the env **once** to node-local `/tmp`, fan out node-to-node. Imports hit local SSD.

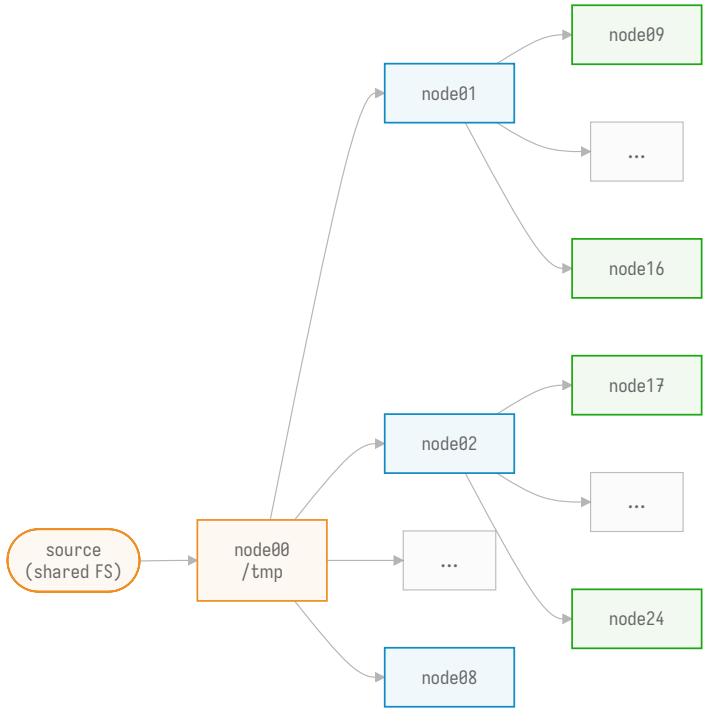
```
ezpz yeet --compress          # 1 tarball off Lustre
source /tmp/.venv/bin/activate
ezpz launch python3 -m your_app.train
```

- Greedy `O(log N)` fan-out tree, not a star
- Per-node cost **8.7s → 0.18s** (48x)
- Full Aurora: pre-launch **under 13 min**



Why it stays `O(log N)`: greedy fan-out

A single source saturates its NIC at ~8 outbound copies. So **each node that finishes becomes a source** for the next batch: the tree grows recursively.



- Cap **8 outbound / source**; a thread pool load-balances to the least-busy one
- Faster nodes don't wait: finish early → serve early
- Result: **~O(log N)** wall-clock, not the O(N) of a single-source star

The parallelism menu

Bethany covered these in depth. Fast recap as a **decision menu**: five axes, mix and match ("4D parallelism" = several stacked). The only question: *what each splits*.

Strategy	Abbr.	What it splits	You reach for it when...
Data Parallel	`DP`	the *batch*: every GPU holds a full model copy, sees different data	the model fits on one GPU
Tensor Parallel	`TP`	individual *weight matrices* across GPUs (within a layer)	a single layer is too big; needs fast interconnect
Pipeline Parallel	`PP`	the model *by layer* (stages), streaming micro-batches through	the model is deep and will not fit vertically
Sequence / Context	`SP`	the *sequence dimension* (long context) across GPUs	context length blows up activation memory
Expert Parallel	`EP`	*experts* of an MoE layer across GPUs	you are training a Mixture-of-Experts

> **Read it as a hierarchy of cost** x
> `DP` is nearly free (no model changes). `TP` is chatty (all-reduce inside every layer, keep it on-node). `PP` needs careful micro-batch scheduling. Add them in that order of pain.

In pictures (1/2): replicate, shard, split a layer

Fig: *Data Parallel* — *model replicated, batch sliced, grads averaged*

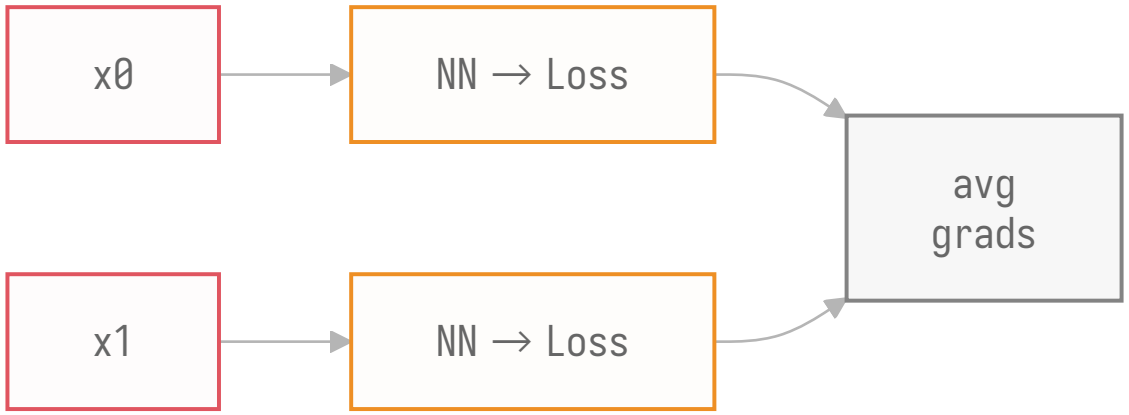
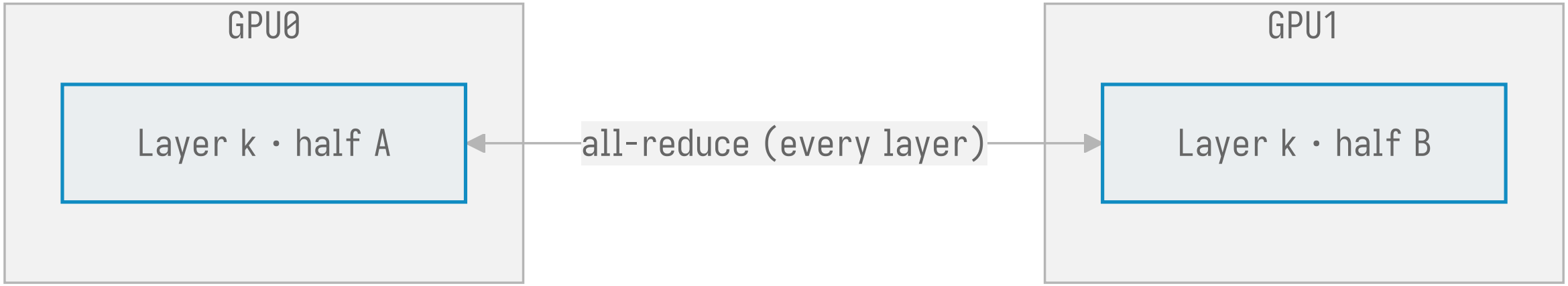


Fig: *FSDP / ZeRO* — *params sharded, all-gathered just-in-time per layer*

Fig: *Tensor Parallel* — *each layer's matmul sliced; all-reduce every layer*

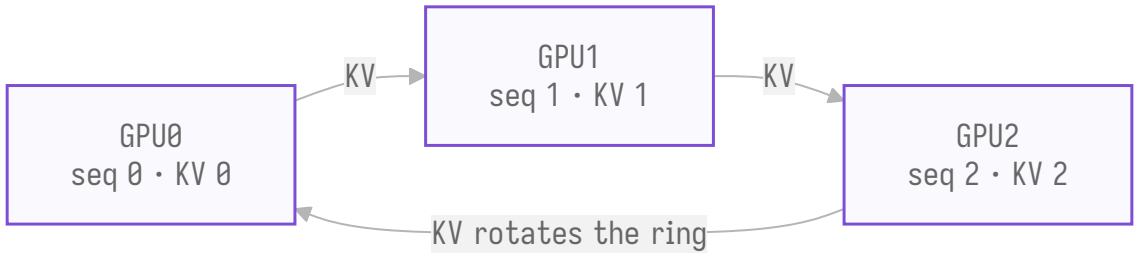


- **DP** is the default (nothing changes). **FSDP** trades comms for memory. **TP** is chatty (all-reduce every layer): keep it **on-node**.

In pictures (2/2): split the depth, split the sequence

Fig: *Pipeline Parallel* — *layer stack split into stages; micro-batches pipelined*

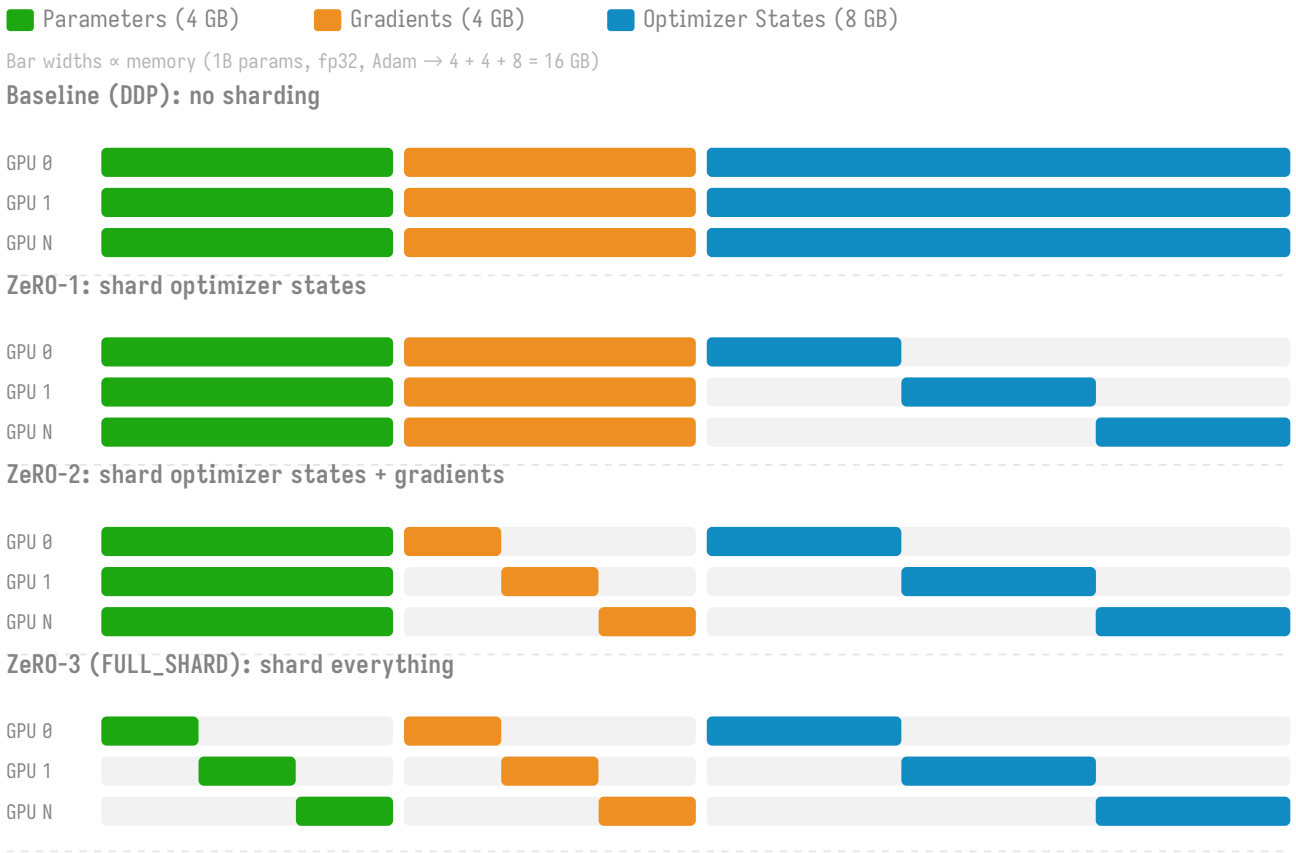
Fig: *Sequence / Context* — *sequence sliced per rank; ring-attention rotates KV*



- **PP** splits the layer stack into stages across nodes; **SP/CP** splits the **sequence** for long context (ring-attention over KV). **EP** (not shown) splits MoE experts.

ZeRO / FSDP: the memory vs comms trade

Still data parallel: shard the replicated state, gather on demand. Each stage shards one more thing, so per-GPU memory drops.[1]



Per-GPU memory (1B params, fp32, N GPUs):

Baseline = Params + Grads + Optim = **16 GB**

ZeRO-1 = Params + Grads + Optim/N = **8 + 8/N GB**

ZeRO-2 = Params + Grads/N + Optim/N = **4 + 12/N GB**

ZeRO-3 = Params/N + Grads/N + Optim/N = **16/N GB**

ezpz: reshard_after_forward=False → ZeRO-2 · =True → ZeRO-3

The trade

Buy memory with **comms**: an all-gather per step. More stages = less memory, more comms.

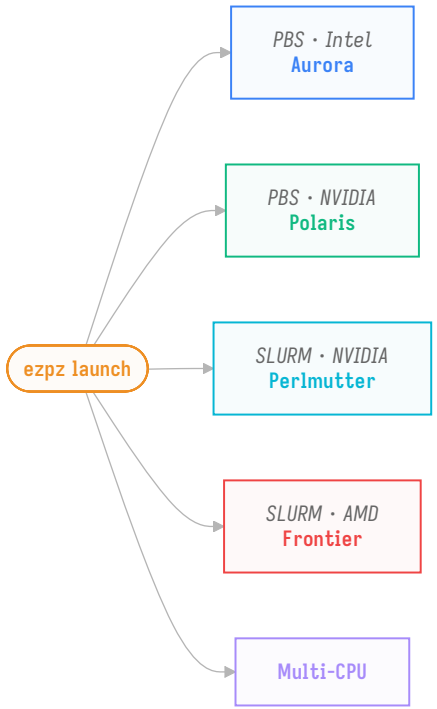
- ✅ model almost fits, or you want a bigger batch
- ✕ a single *layer* won't fit → that's `TP` / `PP`

What you'll actually pick

Climb only when memory forces you: `DP → FSDP/ZeRO → TP → PP`.

- **Fits on one GPU** → `DP`, scale batch
- **Almost fits** → `FSDP`/`ZeRO` (stage 1, then 3)
- **A layer won't fit** → `TP` on-node, `PP` across
- **Long context** → `SP` · **MoE** → `EP`

> **Tip** x
> `ezpz launch` Same script, every machine. No `mpiexec` / `srun` / bind wrappers.

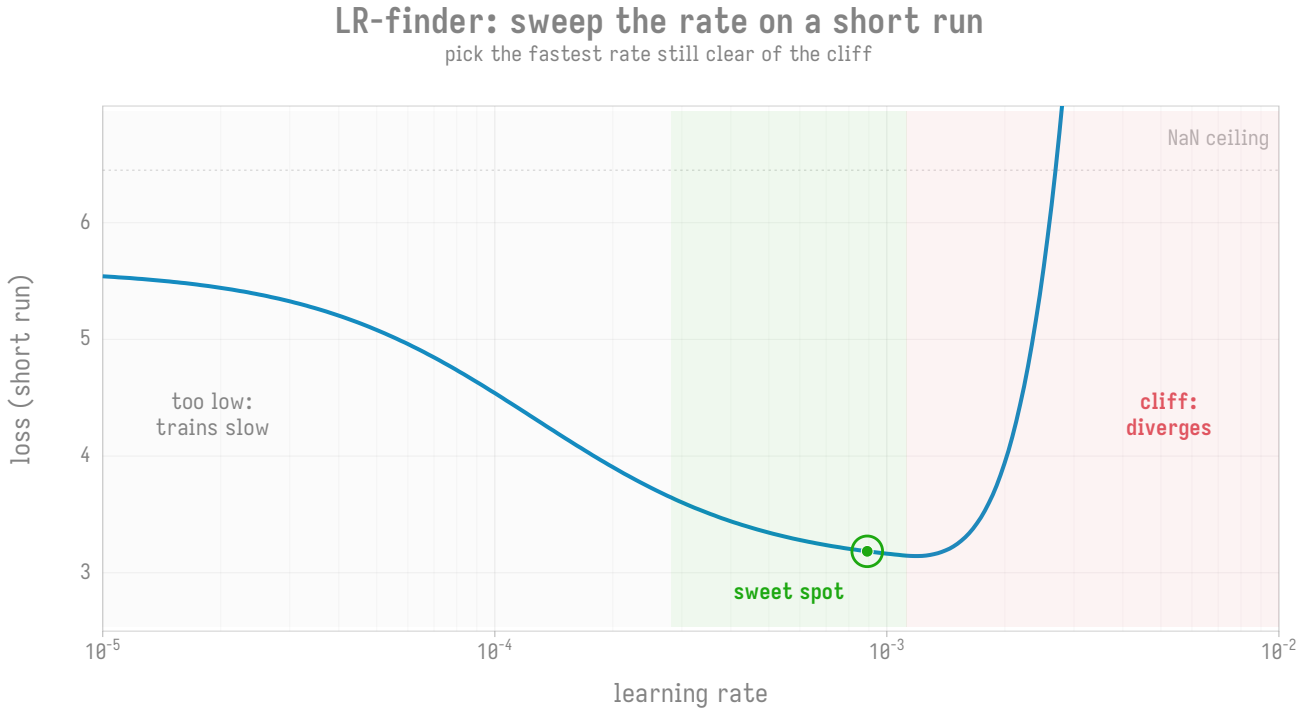


Before you commit: find the learning rate

The last knob before a full-machine run. Sweep the LR on a **short** run and read the curve, don't guess a number and burn 2k nodes finding out it was wrong.

- **flat/high** at small LR: trains, just slowly
- **basin**: loss drops into the usable band
- **cliff**: one step too big → grads to `inf` → `NaN`
- **pick just below the cliff**: the fastest rate that still has safety margin as the run heats up

> **Cheap insurance** x
> A few hundred steps to sweep saves a full-scale run that NaNs at step 7.



You’ve launched. Now everything tries to kill it.

Part 1 spent the compute well. Part 2 is about not **wasting** what you’ve spent.
See you after the break.

Part 2: now keep it alive

Part 1 got a run **started**. That was the easy part.

- Not a program you start: one you **babysit for weeks**
- Every crash you don't recover is **compute you already paid for, thrown away**
- So the question becomes: **how cheaply do you recover when (not if) it breaks?**

> **The rest of this talk** x
> Good news: the loss curve. Bad news: everything that can interrupt it.

Where we're headed: AuroraGPT-2B to 7.77T tokens

One continuous run, **154K steps**, three data-mix stages. Everything in Part 2 is what it takes to keep *this curve* going.

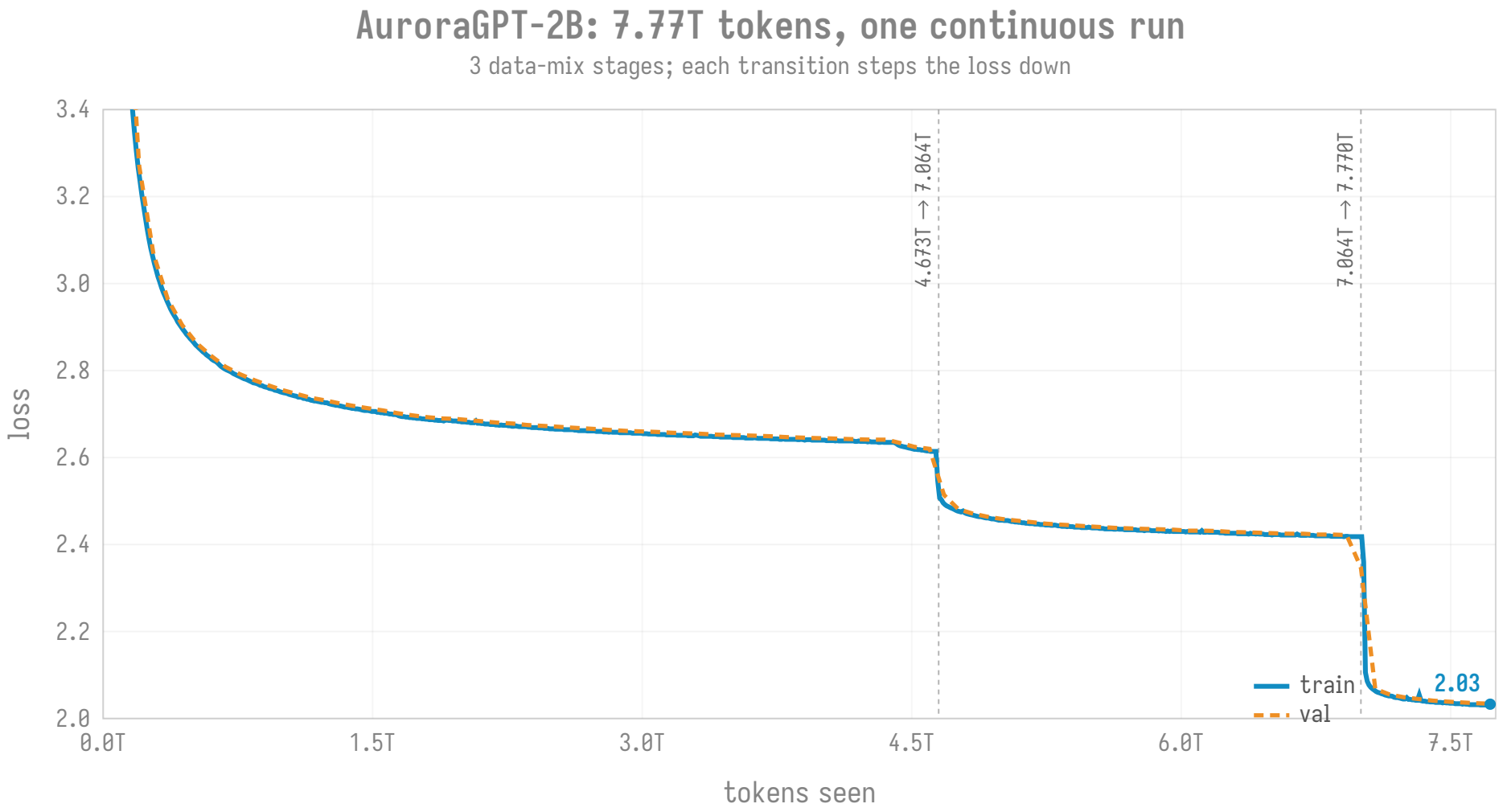


Fig: *Each data-mix transition (dashed) steps the loss down; train and val stay locked together to a final 2.03.*

And the full picture: every run, one axis

Every canonical AuroraGPT pretraining run, loss vs tokens. Bigger models fall **faster per token**; the 2B reference rides three data-mix stages the furthest.

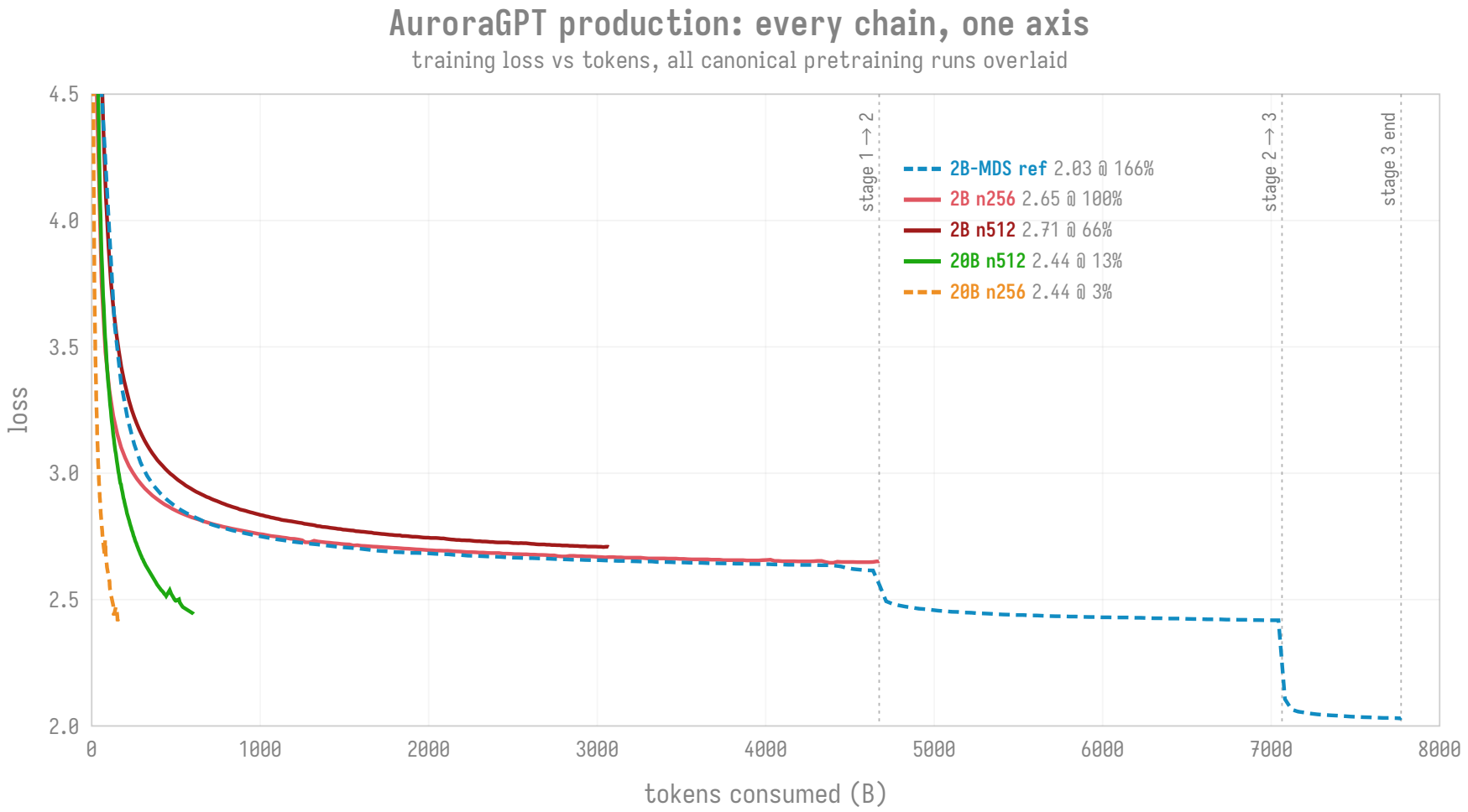
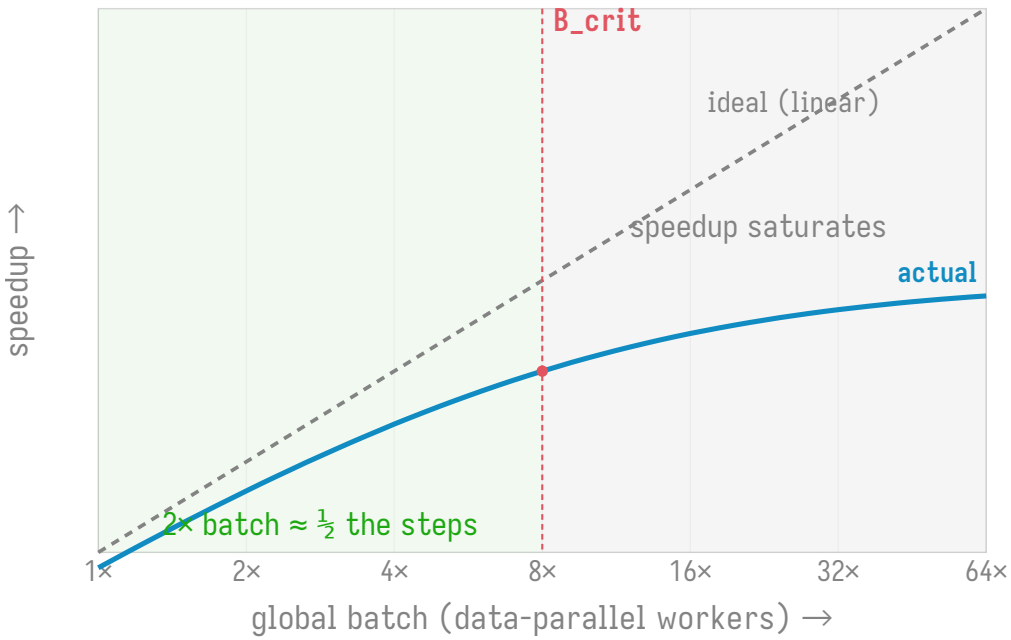


Fig: *The 20B chains (green, orange) drop fastest per token; the 2B-MDS reference (blue) goes furthest, stepping down at each data-mix transition to 2.03.*

Batch size: the ceiling, and the knobs

B_{crit}: the largest batch where more data-parallel workers still buy near-linear speedup.[2]



The ceiling

- below: 2x batch ≈ **half the steps**, more nodes nearly free
- above: speedup **saturates**, training gets **unstable**[3][4]
- HPC hits it first: INCITE jobs want **~2k nodes**, past B_{crit}
- fix is **TP/PP/EP + a batch you *chose***, not "more DP"

The knobs (and where they break)

- **LR scaling** (linear / B)[2] + **longer warmup** + grad clip
- **linear LR backfires:** 80B NaN'd *earlier* (step 7 vs 29)
- **LRs don't transfer:** `mano` @48 loses to AdamW@384 (unless you use **µP**[5])

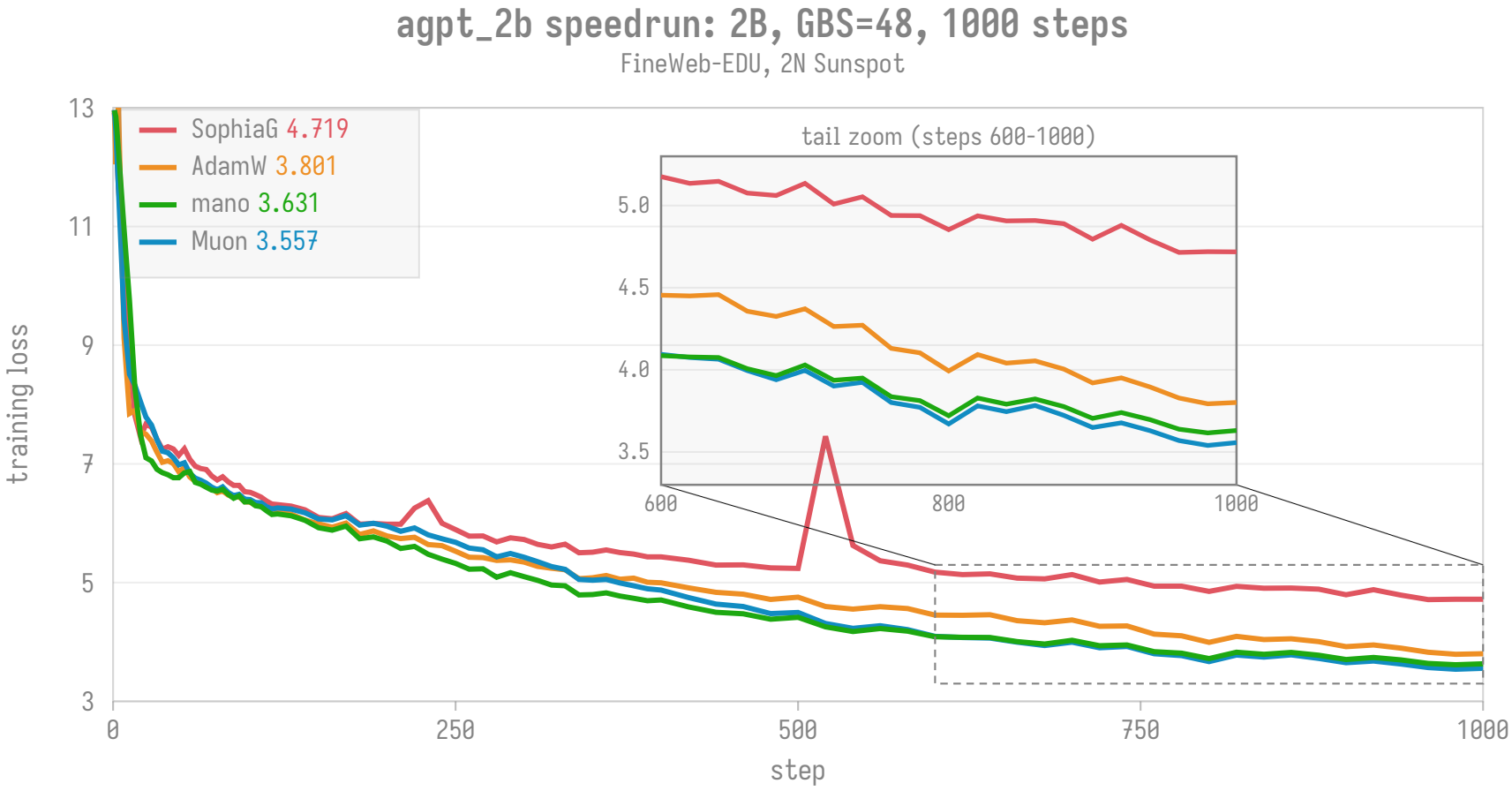
> **Tip** x
>
> Scheduling (grab the machine) and the math (stay under B_{crit}) pull opposite ways. **Re-tune at the target batch and node count**, don't extrapolate a small-batch LR up a 100x jump and hope.

mano : Muon quality at AdamW speed

mano [6] normalizes updates on a rotating **Oblique manifold** with $O(\dim)$ vector-norm ops (no Newton-Schulz iterations). So it matches Muon's loss without Muon's throughput tax.

Optimizer	Loss	TPS
Muon	3.557	4,556
mano	**3.631**	**7,048**
AdamW	3.801	7,245
SophiaG	4.719	7,208

- **Muon and mano tie on loss (~3.6);** mano **runs at AdamW speed** (~7,000 vs ~4,600 TPS), so it **wins on wall-clock.**
- Caveat: at **large batch** (GBS=384) AdamW still wins (2.71 vs 2.88); mano's LR was tuned at GBS=48 and needs re-tuning.



Second-order optimizers: the landscape

AdamW is the baseline. Everything else buys **curvature** at some compute price.

Optimizer	Core idea
AdamW [7]	diagonal 2nd moment + decoupled weight decay
SophiaG [8]	clipped Hessian-diagonal: light curvature, cheap
Shampoo [9]	Kronecker-factored preconditioner per layer
SOAP [10]	Adam in Shampoo's eigenbasis: stabler, fewer knobs
Muon [11]	Newton-Schulz orthogonalize: a cheap Shampoo-like step

Shampoo → SOAP → Muon is one family: same second-moment structure, cheaper each step. **SophiaG** takes the other cheap route: a clipped diagonal Hessian. [12]

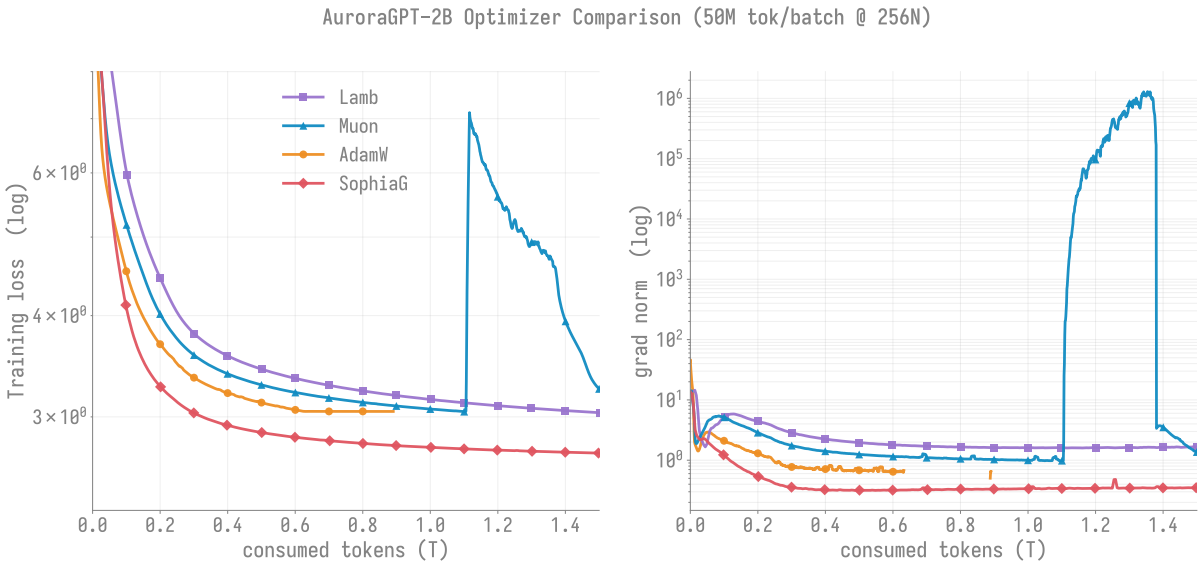


Fig: *Real 2B runs at GBS 6,144:* ***SophiaG*** *reaches the lowest loss with bounded grad norm;* ***Muon*** *diverges at ~1.1T tokens.*

80B: one bf16 cliff, three dead optimizers

At 80B (`dim=9216`), **every** constant-finder LR NaN-ed in the first ~dozen steps: `grad_norm` went to `inf` one step before the loss, while loss was still flat (~12.9).

Optimizer	Died at	Signature
`mano`	step 5	grad NaN (diverged **first**)
AdamW	step 9	grad NaN
SophiaG	step 14	Hessian-term overflow, ~6,100 node-h

- `mano` **died first** (step 5), despite the *safest*-looking LR band
- shared failure across 3 → a **bf16-at-dim=9216 corner**, not tuning
- **fix:** long warmup + grad clip; stable at **TP=4 · LBS=1**

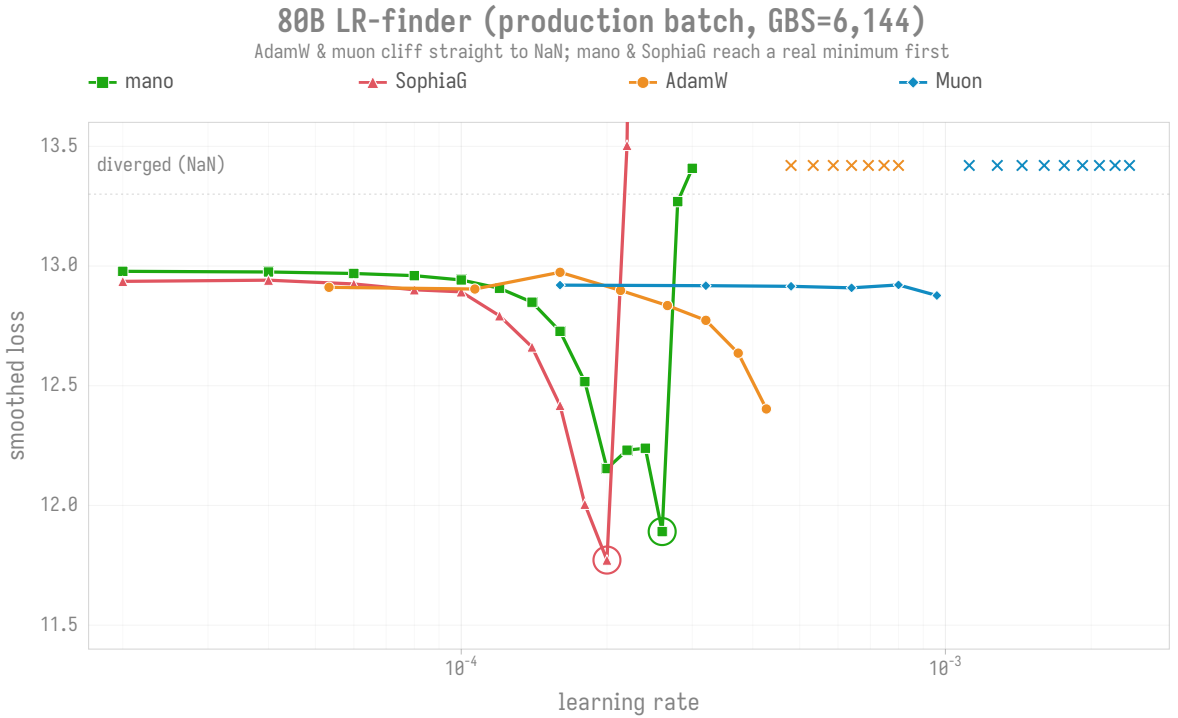


Fig: *80B LR-finder (GBS=6,144):* ***AdamW and muon cliff straight to NaN;*** `mano` *and SophiaG reach a minimum first, then blow up past it.*

At scale, failure is the default

You are running one job across thousands of nodes in lockstep. The job dies when **any** node dies, so cluster MTBF is roughly single-node MTBF divided by N.

- one node fails **~once a year**; put **16,000** in one job → **~44 failures/day**

Run	Scale	Failures
Llama 3 405B[13]	16K H100s · 54 d	**419** (1 every 3h); 99% auto-recovered
OPT-175B[14]	~1K A100s · 2 mo	35 manual restarts + 100+ hosts cycled
BLOOM-176B[15]	384 A100s · 3.5 mo	frequent loss spikes
GLM-130B[16]	768 A100s · 2 mo	spikes worsen; some NaN

Plan for the failure, not for the happy path.

The Lustre tax comes back, now on the write path

In Part 1 the shared-FS tax hit **reads**: staging the env off Lustre with `ezpz yeet`. Checkpointing hits the **same wall from the other side**.

- **reads** (Part 1): every rank `stat()`s the same env → stage node-local
- **writes** (now): every rank flushes a **232 GB** checkpoint to the **same MDS**
- same fix, same fan-out: **coordinate the writes**, don't stampede

> **Why it's worse for checkpoints** x
> A stale env costs you a slow startup. A checkpoint write that contends with training collectives can take the **whole job** down (next slide).

Checkpointing: what, when, how big

Your only insurance policy. Resume from the last one, everything since is wasted.

What goes in
- model weights (fp32 master)
- optimizer state (~2x the model)
- LR sched, step, RNG, dataloader pos

How big (measured)
- **20B → 232 GB.** Sync save stalls training **~23.6 s**; reload is **~55-63 s** (dominated by `dcp.load`).

The **~24 s** stall is why you push it off the critical path → **async**, next.

When: the cadence tradeoff
- **too frequent:** write overhead eats throughput
- **too rare:** a crash throws away hours

EV rule: interval where **wasted work on a crash ≈ write overhead**. At a failure every few hours, and a **~24 s** blocking save, that lands in the tens-of-minutes range.

Asynchronous checkpointing, and how it melted the cluster

Done right, async hides the 232 GB write behind the next few hundred steps: snapshot to host memory, flush on a background thread, bound the fan-out.

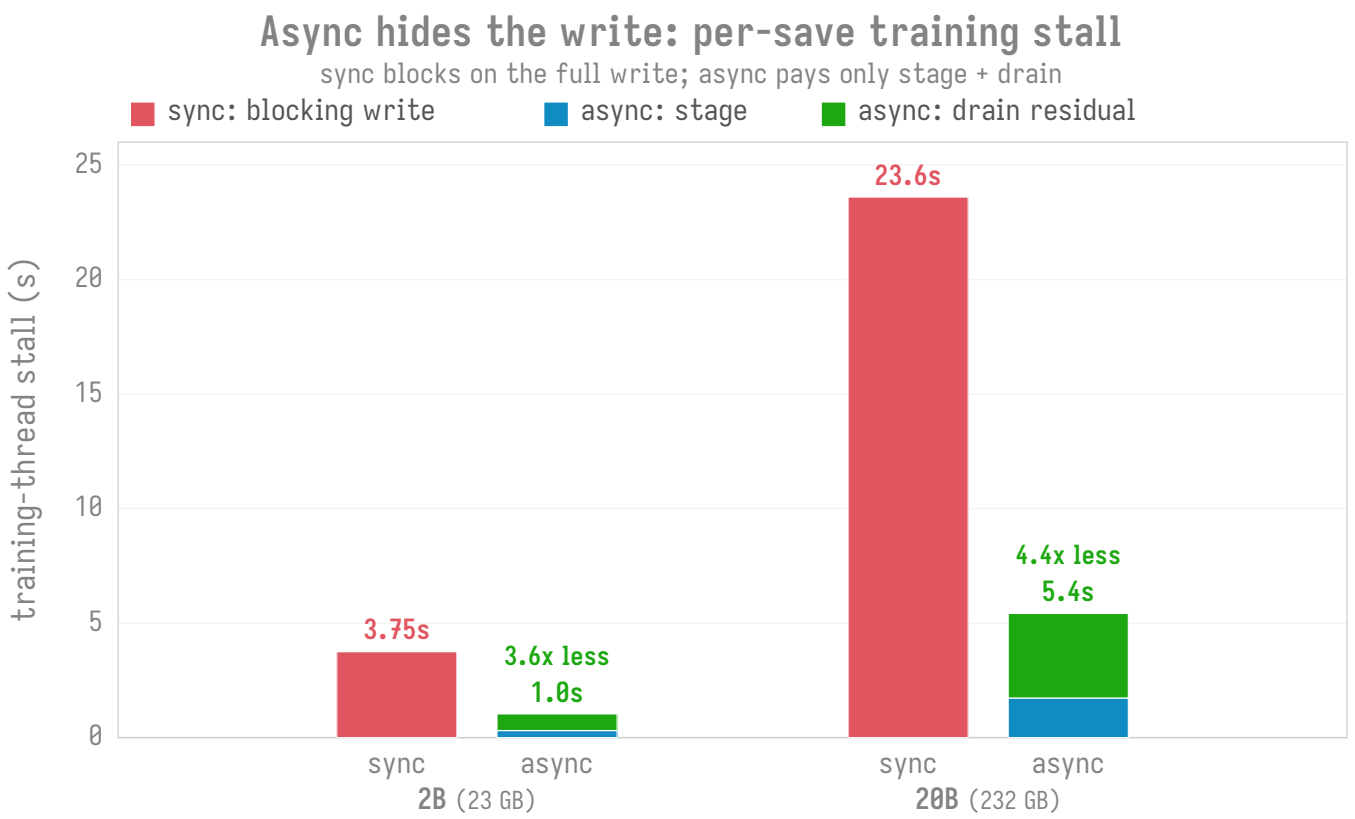


Fig: *Sync blocks the training thread on the whole write; async pays only stage + drain while the GPUs keep going. Measured on Aurora (ezpz guide).*

- **snapshot to host memory**, then flush on a background thread
- **bound concurrency:** cap ranks flushing at once
- reload is **~55-63 s** at 20B (`dcp.load`-bound)

The naive version took the job down x

Unbounded async at **20B / 512N+** contends with training collectives on the same fabric. Bound the fan-out; write a `.complete` marker last.

Restarting quickly

The recovery clock runs crash → first-step-back-training, not crash → "file exists."
Reload fast (reload is **~55-63 s** at 20B, `dcp.load`-bound)

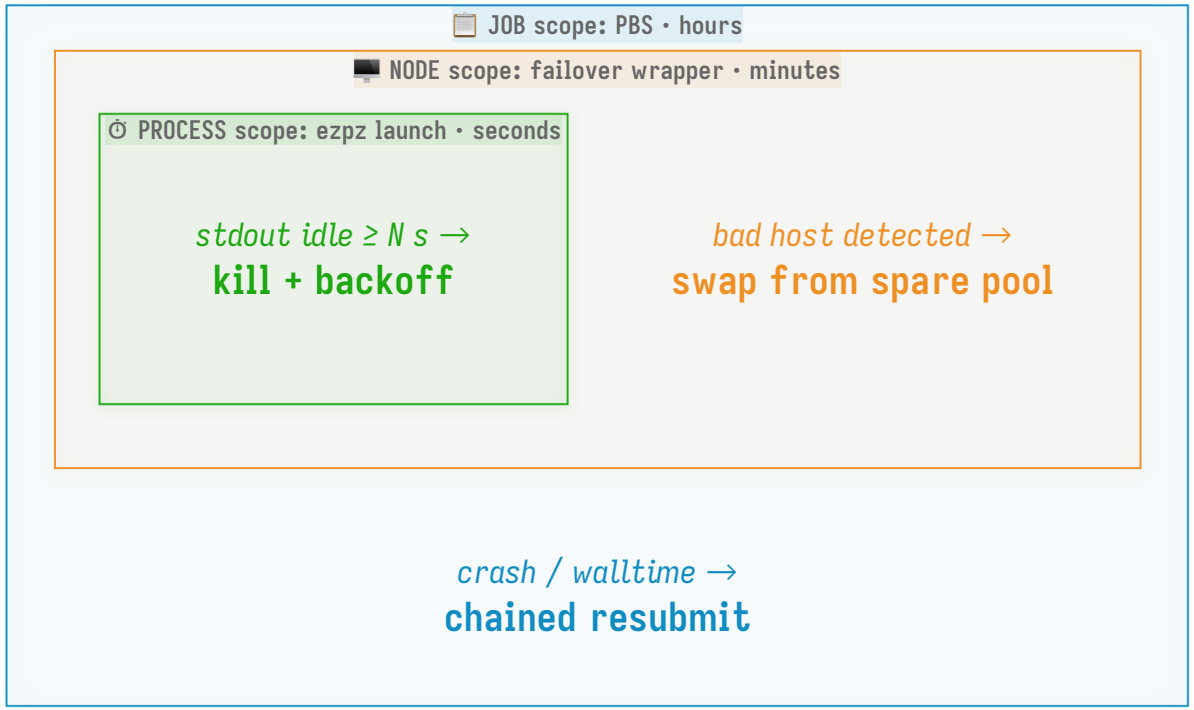
- **stage node-local:** `ezpz yeet checkpoint-729` → `/tmp`, not 512 ranks pulling 232 GB off Lustre at once
- same fan-out as the venv, any dir or tarball

Reload anywhere: converters

- **Megatron ⇄ 🤗 HF ⇄ ZeRO ⇄ Universal**
- **Universal** decouples from the parallelism layout → resume at a *different* TP/PP/DP than you saved at
- the converted HF checkpoint is what you **evaluate, serve, or fine-tune from**

”The restarts”: three layers of recovery

Bad-node failover, hang-watchdog, and PBS resubmit each operate at a different scope.



Inner loops catch most failures; outer loops catch the rest.

`--auto-retry`: bad-node failover, on tap

The NODE layer: allocate spares up front, swap them in on failure. Out of bash, into `ezpz`.

```
# 522 nodes allocated, train on 512, keep 10 as spares.
# Loop until success, walltime, or spare exhaustion.
ezpz launch --auto-retry --np 512 -- python -m torchtitan.train …
```

- classifies exit: `success` / `walltime` / `bad-node` / `stuck-pre-training`
- **bad-node** → scrape host from log, swap spare, re-exec
- **config-bug guard:** 2 attempts with zero `step=` → stop

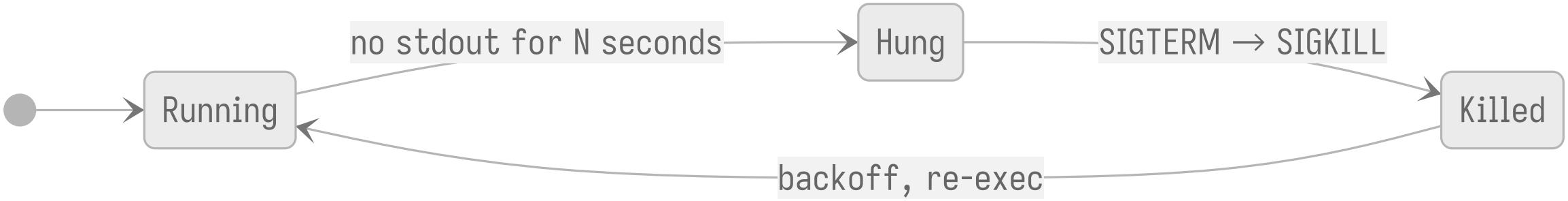
Broke the 20B/512N stall: that stale placeholder → walltime-blocked **weeks**; relaunch drove it **4,400 → 5,400**. Ships in `ezpz#144`.

The detector under it all: the idle-timeout watchdog

`--auto-retry` (previous slide) needs to *know* a job hung. A hang **quiets stdout**, so idle stdout is the signal, and this watchdog is what fires on it.

```
# --auto-retry turns the watchdog on for you (idle default = FAILOVER_IDLE_TIMEOUT).
# Standalone, without node failover: --timeout is the detector, --retries re-execs.
ezpz launch --timeout 600 --retries 3 \
    python -m torchtitan.train --config-file ./config.toml
```

- `--timeout SECONDS`: kill if stdout goes **idle** (not walltime); exit 124
- `--retries N`: bounded re-exec, backoff 5/10/20/40/60s. **Mutually exclusive** with `--auto-retry` (bounded per-process vs unbounded node-level)



Fires on **absence of progress**, not a heartbeat ping (a hung job is "alive" by `kill -0`).

It caught a real silent hang in production

Job `8505298`, 2026-05-23. Trains steps 1-37, then the log goes **silent**. No traceback, no rank dying. Just dead.

Time (CT)	Event
21:06:41	step 37 logged · `loss 11.80 · tps 3,919`
21:36:41	**30 min dead air** · `ezpz launch --timeout=1800` SIGTERMs
21:36:43	wrapper classifies exit 124 to silent-hang (not walltime)
21:36:43	no traceback to scrape, so **blind swap** of rank-0 host
21:36:45	attempt 2 launches on swapped node set
21:57:49	walltime hit · step 296 · loss 5.68 · ckpts persisted

Three new pieces fired in sequence on a real hang: the `--timeout=1800` watchdog, the `exit 124` classification, and `failover_swap_one_blind()`. Unattended, 9:36 PM on a Friday. Full writeup.

Where production stands

What the machinery above actually produced. The next few slides zoom in on these.

Run	Nodes	Steps	Tokens	Loss	Status
2B base	256	92,859	4.674T (100%)	2.65	✅ **complete**
2B-MDS ref	256	154,391	7.770T (166%)	2.03	✅ 3-stage reference
20B/512N	512	6,010	605B (13%)	2.44	advancing (auto-retry)
20B/256N	256	6,301	159B (3%)	2.44	advancing
80B	512	14	(NaN'd)	NaN	✕ optimizer cliff

- Two 2B runs **complete**; both 20B chains **advancing**; the **80B** hit the bf16 optimizer cliff (the wall from Part 2).

The flagship run: AuroraGPT-2B to 7.77T tokens

One continuous run, **154K steps**, three data-mix stages. It rode the whole Part 2 machinery: async checkpoints, layered auto-restart, silent-hang recovery.

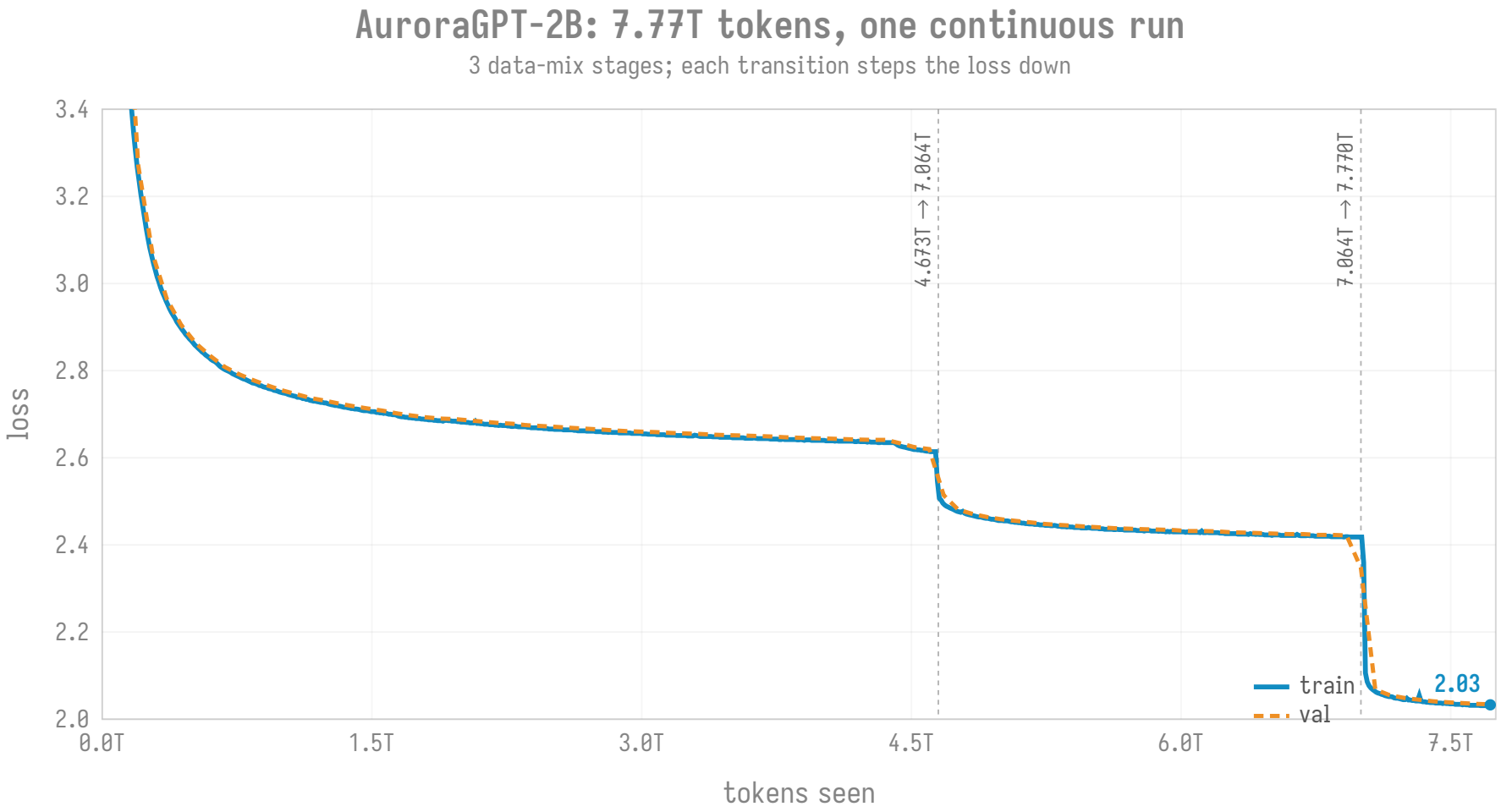


Fig: *Each data-mix transition (dashed) steps the loss down; train and val stay locked together to a final 2.03.*

The whole program, one axis

Every canonical AuroraGPT pretraining run, loss vs tokens. Bigger models fall **faster per token**; the 2B reference rides three data-mix stages the furthest.

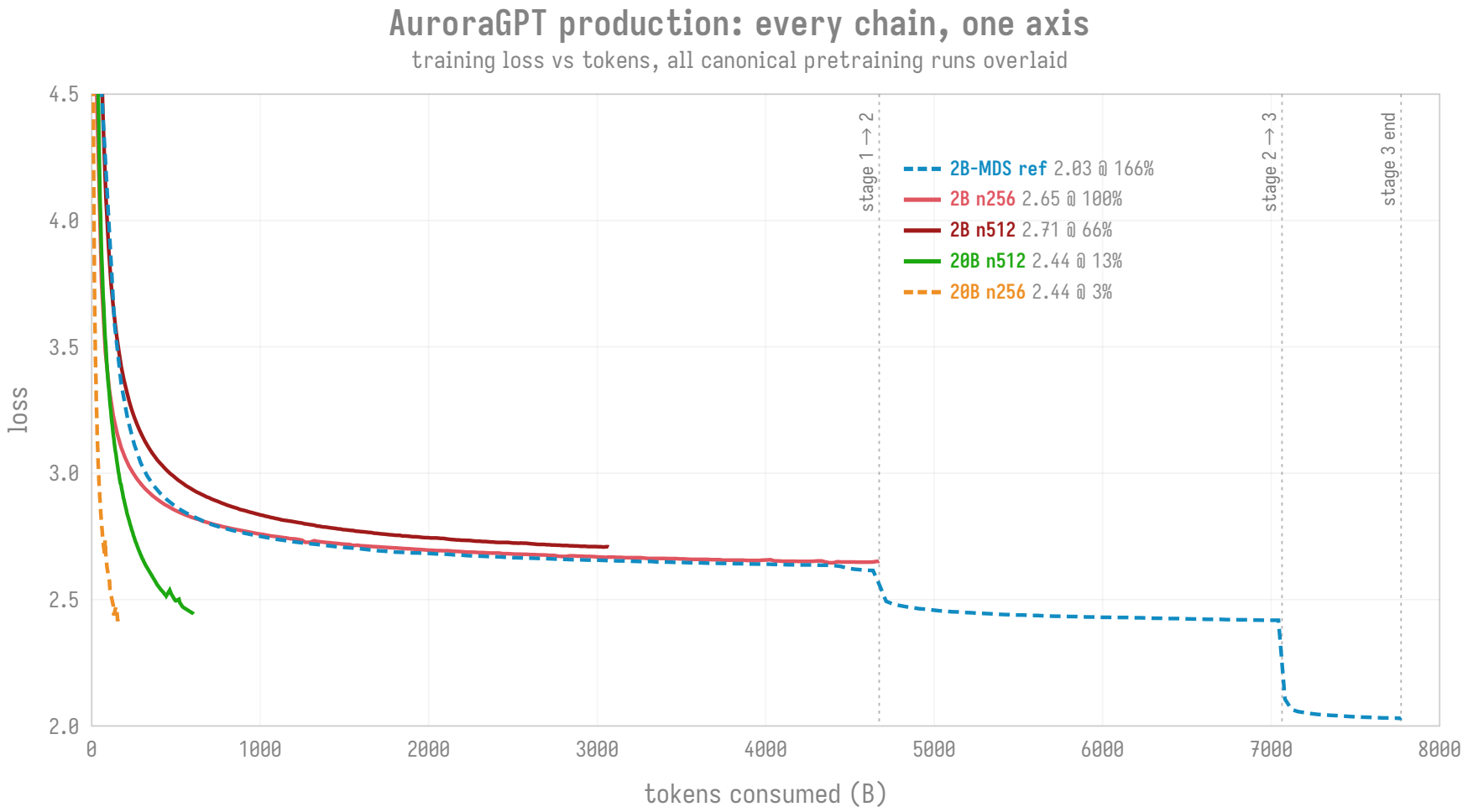


Fig: *The 20B chains (green, orange) drop fastest per token; the 2B-MDS reference (blue) goes furthest, stepping down at each data-mix transition to 2.03.*

What it costs: throughput and MFU

Same chains, the compute side: sustained per-GPU throughput and how much of the hardware's FLOPs we actually use.

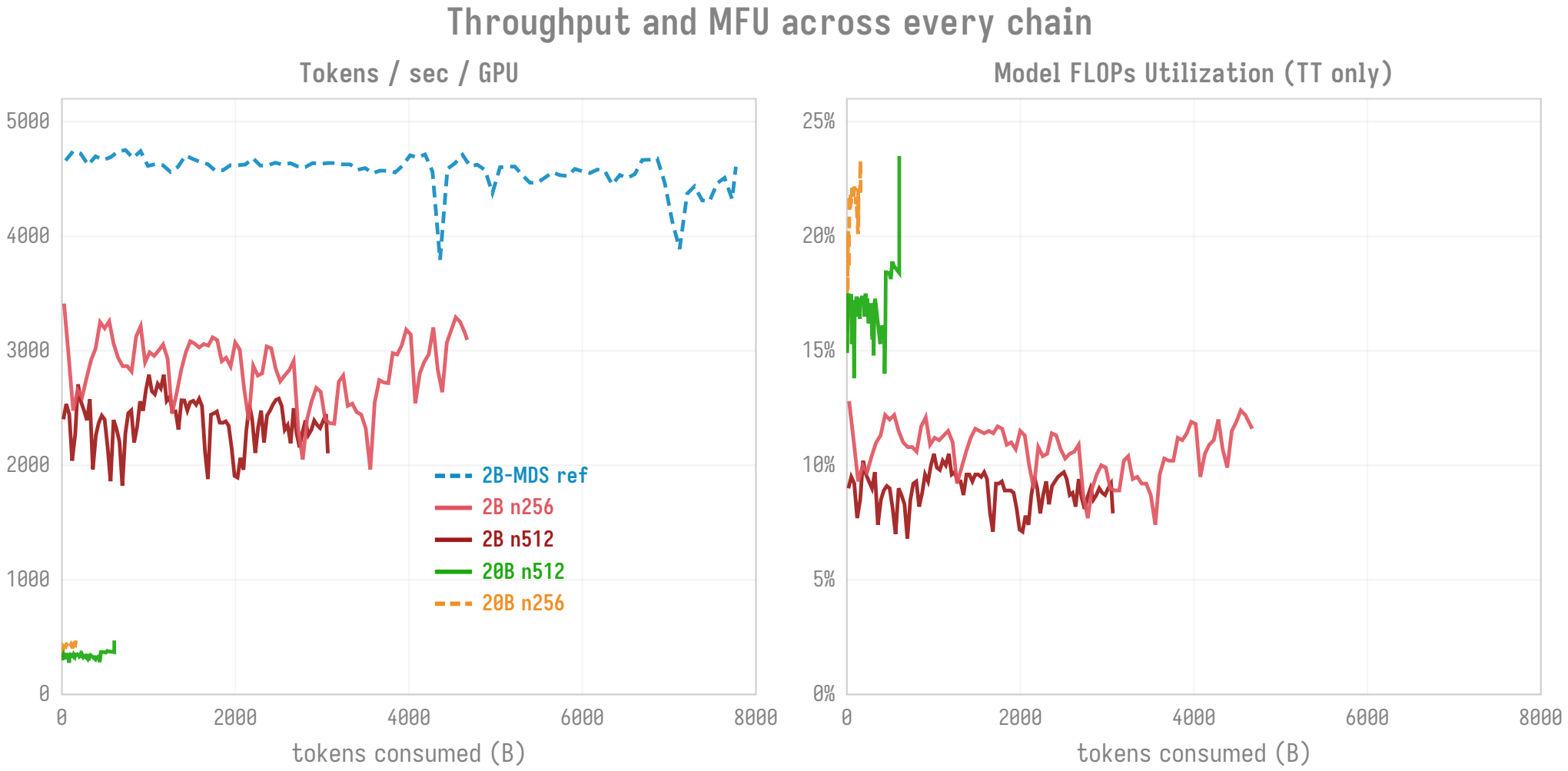


Fig: *Bigger model = fewer tokens/sec/GPU but **higher MFU** (20B ~17-22% vs 2B ~9-12%): the 20B keeps the XPUs busier per step.*

It learns: eval vs tokens, and the 20B is most efficient

Loss falling is necessary, not sufficient. The eval gate is what says the model learned, and **per token the 20B pulls ahead of both 2B chains.**

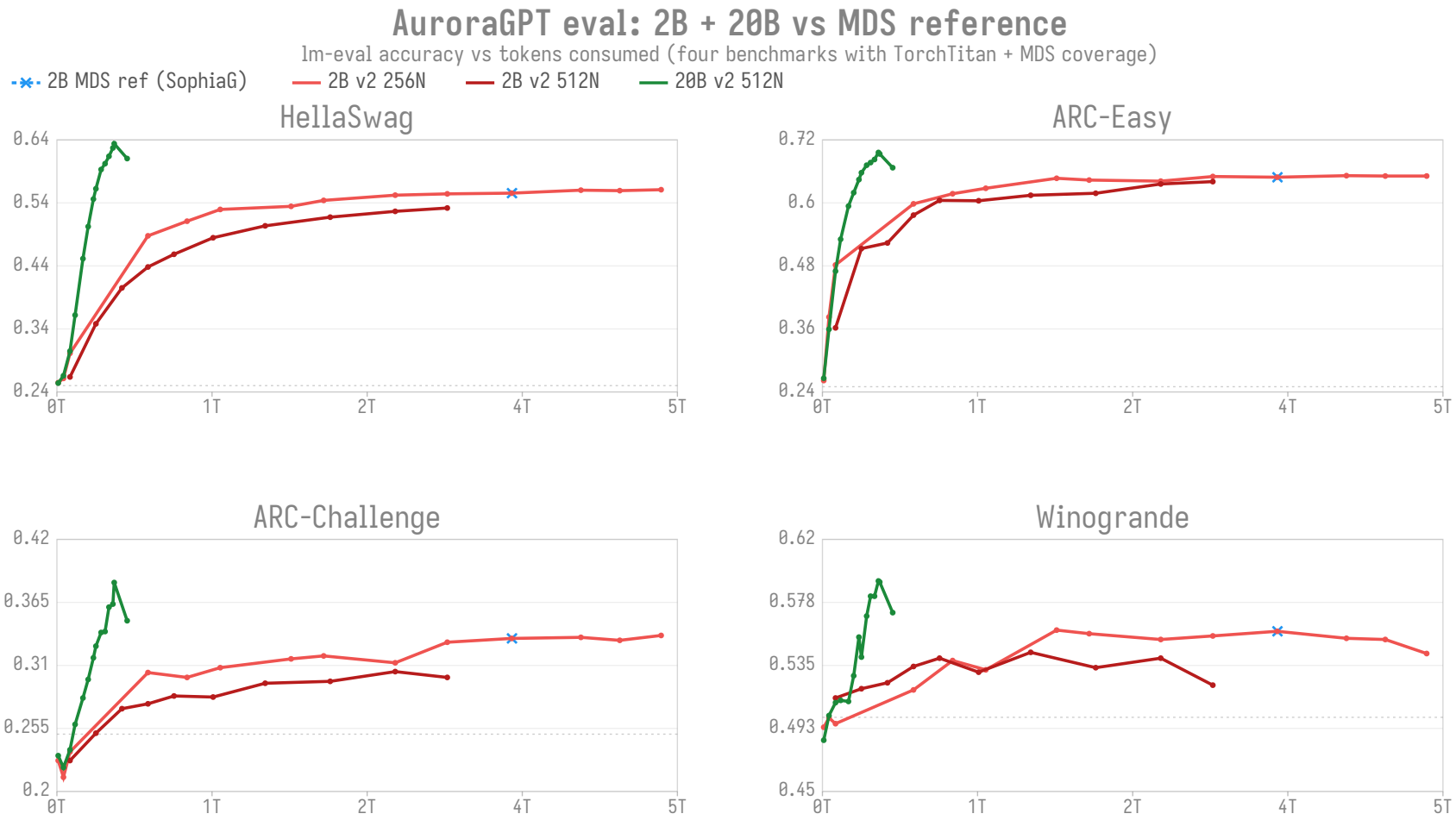


Fig: *Four benchmarks vs tokens. The **20B (green) climbs steepest per token**: ARC-Easy ~0.69 by ~440B tokens, already past both 2B chains. More model per GPU-hour, made literal.*

Putting it together: one resilient launch

Everything in Part 2 collapses into a startup ritual and a single launch line. Nothing here is aspirational: each flag is one we saw earn its keep.

```
# 1. stage the env node-local (off Lustre) · Part 1
ezpz yeet --compress
source /tmp/.venv/bin/activate
# 2. launch with all three recovery layers armed · Part 2
ezpz launch \
    --auto-retry \                # NODE:    swap bad hosts from the spare pool
    --np 512 \                    #          (allocate 522, keep 10 spare)
    --timeout 1800 \              # PROCESS: kill on 30 min of stdout silence
    --retries 3 \                 #          re-exec with exponential backoff
    -- python3 -m torchtitan.train --config-file ./config.toml
# JOB layer: a chained PBS resubmit wraps the whole thing (hours scope)
```

- **Data** blended + pinned, **env** broadcast off Lustre, **checkpoints** async and node-local-stageable, **failures** caught at process / node / job scope.
- The person who launches this can **close the laptop**. The 9:36-PM-Friday silent hang recovered itself.

> **The whole talk in one line** x
>
> It all serves one number: **model quality per GPU-hour.** Spend the compute well (Part 1), and make recovery so cheap that a failure wastes almost none of it (Part 2).

What generalizes, what doesn't

Generalizes *(vendor / site / model)*

- Bit-exact deterministic **smoke gate** after every upstream sync
- **lm-eval** as ground truth for "is it actually learning?"
- **Spare-node failover** wrapper: same idea on Slurm
- **Launcher / env autodetect**: push vendor-shaped assumptions *out* of training code

Doesn't *(re-tune per config / hardware / version)*

- `torch.compile` decisions (helps dense, can *hurt*)
- Collective tuning (CCL vs gloo fallbacks, NCCL/CCL env)
- **Optimizer stability at the bf16 corner**: LR-finder rank ≠ sustained stability

What to take home

Five ways to get more model per GPU-hour:

The five lessons

1. **Data prep is a distributed-systems problem.** Tokenize once, blend by weight, pin everything for reproducibility.
2. **Do not hit Lustre from every rank.** Stage node-local; broadcast the env.
3. **Add parallelism in order of pain:** `DP → FSDP/ZeRO → TP → PP`, and stay under the critical batch size.
4. **Loss is not ground truth.** Gate on evals; a bit-exact smoke test catches silent corruption.
5. **Do not prevent failures, recover cheaply.** Async checkpoints + layered auto-restart.

Start here

- 🍋 **ezpz** (launch anywhere, `--auto-retry`, `yeet`): github.com/saforem2/ezpz · ezpz.cool
- 🧠 **torchtitan fork** (XPU + our experiments): github.com/saforem2/torchtitan
- 📚 **blendcorpus** (weighted data blending): github.com/zhenghh04/blendcorpus

Deeper write-ups

- Running 50k Python processes on Aurora with `ezpz yeet`
- BlendCorpus + TorchTitan @ ALCF

Try it yourself: FSDP + TP in four lines

Everything in this talk, runnable now. **Copy-paste onto any of these machines.**

```
# 1. bootstrap the ezpz environment (auto-detects the machine)
source <(curl -fsSL https://bit.ly/ezpz-utils) && ezpz_setup_env
uv pip install --no-cache --link-mode=copy "git+https://github.com/saforem2/ezpz"
# 2. smoke test first (always)
ezpz launch python3 -m ezpz.examples.test
# 3. full FSDP + TP pre-training
ezpz launch python3 -m ezpz.examples.fsdp_tp
```

> **Same script, every machine** x
> No `mpiexec` / `srun` / bind wrappers: `ezpz launch` auto-detects the scheduler and device.
> Smoke-test before the real run, every time.

samf.sh/talks/2026/08/03

Thanks

AuroraGPT team: Venkat Vishwanath, the AI/ML Group at ALCF, collaborators across ANL.
Argonne Leadership Computing Facility: Aurora time, Sunspot staging.
Intel: Intel Max 1550 XPU + oneAPI / XCCL / IPEX support throughout.

Code & docs

- `ezpz`: github.com/saforem2/ezpz
- `torchtitan` fork (experiments/ezpz): github.com/saforem2/torchtitan
- These slides: samf.sh/talks/2026/08/03

> This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357.

Questions?

Footnotes

1. ZeRO: Memory Optimizations Toward Training Trillion Parameter Models (Rajbhandari et al. 2020), Fig. 7. Per-param Adam mixed-precision budget: 2 (fp16 params) + 2 (fp16 grads) + 12 (fp32 master + momentum + variance) = 16 bytes; sharding across `N` ranks drives the sharded piece toward `16/N`. ↩
2. An Empirical Model of Large-Batch Training (McCandlish et al. 2018): LR and warmup both need to grow with batch, and the payoff flattens at the critical batch size. ↩ ↩2
3. How Does Critical Batch Size Scale in Pre-training? (Zhang et al. 2025). ↩
4. How to Set the Batch Size for Large-Scale Pre-training? (Zhou et al. 2025). ↩
5. **µP** (maximal-update parametrization): Tensor Programs V (Yang et al. 2022). Reparametrize so the optimal LR is invariant to width, tune on a small proxy, then transfer to the full model ("µTransfer"). ↩
6. `mano`: manifold-normalized optimization (2026). Implemented in our fork alongside SPAM (2501.06842). ↩
7. Decoupled Weight Decay Regularization (Loshchilov & Hutter 2019). ↩
8. Sophia: A Scalable Stochastic Second-order Optimizer (Liu et al. 2023). A clipped diagonal-Hessian preconditioner; SophiaG estimates the diagonal with the Gauss-Newton-Bartlett trick. ↩
9. Shampoo: Preconditioned Stochastic Tensor Optimization (Gupta et al. 2018). Kronecker factors of $GG^\top$. ↩
10. SOAP: Improving and Stabilizing Shampoo using Adam (Vyas et al. 2024). Runs Adam in Shampoo's eigenbasis, refreshed every k steps. ↩
11. Muon: MomentUm Orthogonalized by Newton-Schulz (Jordan et al. 2024/2025). ↩
12. Also common: **LAMB** (You et al. 2020), layer-wise trust ratio for huge batches; **Adopt** (Taniguchi et al. 2024), converges for any β_2; **Sophia** (Liu et al. 2023), clipped Hessian-diagonal. ↩
13. Llama 3 herd of models (Meta AI, 2024), §3.3.2 (Training reliability). ↩
14. OPT-175B chronicles + dev log (Zhang et al., 2022). ↩
15. BLOOM: A 176B-Parameter Open-Access Multilingual Language Model (BigScience, 2022). ↩
16. GLM-130B: An Open Bilingual Pre-Trained Model (Zeng et al., ICLR 2023). ↩

Appendix: backup slides

Material that didn't make the main path but is here for Q&A. Pull the full slides from the AuroraGPT-at-scale deck as needed:

- **Post-training:** CPT · SFT · GRPO curves (covered in Jane's talk)
- **MoE on Intel XPU:** throughput and where it stands
- **AERIS:** a production case study on the same stack
- **Failover engineering:** the full silent-hang recovery write-up
- **Open questions** we'd like this community's help on

A field guide to what breaks

The cost is telling transient (retry, fine) from systemic (retry reproduces it).

Two axes: where it broke x how you treat it

	TRANSIENT retry, fine	SYSTEMIC retry reproduces it
Hardware	ECC blip thermal throttle one bad node	dead XPU (persistent)
Software	OOM (transient spike)	config bug ● **corrupt shard** ● **bad upstream commit**
Network	fabric flap gloo drop	● **collective hang**
System	PBS walltime scheduler hiccup	Lustre stall un-drained bad node

● ● = silent: no traceback, loss still looks fine (the dangerous ones)

War story: the fabric that wouldn't sit still

At full-machine scale the interconnect is never fully healthy. It shows up two ways:

- **Loud:** `gloo` `Connection closed by peer` (job `8470102`). You get an exit code.
- **Quiet:** a collective just **stops**. Every rank blocks in the same `all_reduce`.

The quiet one is dangerous:

- **alive** by `kill -0`, nothing crashed
- `xccl` ignores `train_timeout_seconds` → eats **full PBS walltime**
- job `8479579` hung at **step 803**, heartbeat still ticking

The one signal you can trust: a hang **quiets stdout**. Absence of progress is the detector (→ watchdog).