

miao IO Profiling — Consolidated Report

GitHub Issue #1: AI-HHMI/miao | 21 configurations profiled | 100 samples each

TERM GLOSSARY

Disk Read & Decompress: Reading compressed data from shared network disk + decompressing
IO Bandwidth: How fast useful data flows from disk (data bytes / read time)
Read Amplification: Ratio of bytes read from disk vs bytes actually needed (>1 = waste)
TensorStore Read Lat.: Time per file/shard read operation inside TensorStore
Isotropic Resize: Resampling anisotropic voxels to equal spacing (F.interpolate)
Post-Process: Stack 3 resolution levels → reorder axes → normalize to [0,1]
TensorStore Cache: 1 GB in-memory cache; Hits=found, Misses=must read, Evictions=removed
Multi-thread Decomp.: How many CPU cores decompress data at once (CPU time / wall time)
Image/Label L0/L1/L2: Data at Full Resolution / 2x Downsampled / 4x Downsampled
Patch: A 3D sub-volume extracted for training (e.g. 128×128×32 voxels)

Config	Total (ms)	Read+Decomp (ms)	IO Bandwidth (MB/s)	Disk Read (MB)	Data Needed (MB)	Read Amplif.	TS Read Lat. (ms)
Jiefu (zarr2)	187	182	43	103.2	7.9	13.1x	16.6
H01 (zarr3 sharded)	51	31	131	8.5	4.1	2.1x	3.8
128 ³ patch (iso=true)	124	112	55	59.7	6.2	9.6x	6.9
128 ³ patch (iso=false)	111	103	109	52.9	11.3	4.7x	5.8
64 ³ patch	86	78	9	28.9	0.7	40.6x	4.3
256 ³ patch	200	109	424	102.2	46.5	2.2x	8.7
Worker sweep	109	93	64	50.2	5.9	8.5x	4.3
zarr2 (gzip)	182	176	45	102.8	7.9	13.1x	14.0
zarr3 no-shard	183	176	45	104.7	7.9	13.3x	17.5
zarr3 sharded	126	120	66	85.9	7.9	10.9x	8.2
zarr3 shard 128c	52	47	167	26.2	7.9	3.3x	5.9
z2 c64	16	13	125	4.6	1.6	2.9x	1.7
z2 c128	33	30	53	10.3	1.6	6.6x	4.7
z2 c256	97	94	17	42.4	1.6	26.9x	14.7
z3 ns c128	34	29	54	11.0	1.6	7.0x	5.7
z3 s512 c64	18	14	109	5.2	1.6	3.3x	2.4
z3 s512 c128	32	28	55	11.0	1.6	7.0x	6.7
z3 s1024 c128	36	29	54	11.2	1.6	7.1x	7.5
z3 s512 c256	102	100	16	40.1	1.6	25.5x	14.5
z3 s1024 c256	103	99	16	43.9	1.6	27.9x	18.6
Batch sweep	133	115	50	51.8	5.8	9.0x	6.1

Dataset comparison Patch/iso scaling Worker sweep Batch sweep Format comparison Chunk/shard sweep

Key Findings Overview

FINDING 1: Storage Format – zarr2 Is 6x Slower Than zarr3 Sharded (p.3-5)

COMPARED: Jiefu (zarr2, unsharded) vs H01 (zarr3 sharded), same patch 128³.

FOUND: Jiefu reads 103 MB from disk for 7.9 MB of data (13x amplification). H01 reads 8.5 MB for 4.1 MB (2.1x).

IO Bandwidth (higher = faster): H01 (131 MB/s) vs Jiefu (43 MB/s).

BOTTLENECK: CPU decompression. Both use 15-17 CPU cores in parallel.

Cache: Jiefu 1% hit, H01 10% – sharding causes accidental reuse (reading a shard for chunk A caches neighbor chunk B for free).

FIX: Convert Jiefu to zarr3 sharded. Use faster codec (zstd over gzip).

FINDING 2: Isotropic Mode Is Cheap – 3.4ms, 2.7% (p.6)

COMPARED: Same volumes (Jiefu + H01) with isotropic ON (124ms total) vs OFF (111ms).

FOUND: F.interpolate adds only 3.4ms – negligible vs disk read time. IO Bandwidth (higher = faster): noiso 109 vs iso 55 MB/s.

ROOT CAUSE: Isotropic resampling is pure CPU work, not an IO bottleneck.

MEANING: Continuous scale sampling (Issue #8) is computationally feasible.

FINDING 3: Larger Patches Are More IO-Efficient (p.7)

COMPARED: 64³, 128³, 256³ patches on same volumes (Jiefu + H01).

FOUND: Read amplification: 40.6x → 9.6x → 2.2x (64³ → 128³ → 256³). IO bandwidth (higher = faster): 9 → 55 → 424 MB/s.

ROOT CAUSE: Small patches cross more chunk boundaries. Larger patches spread the fixed cost of reading chunks over more useful data.

FIX: Use the largest patch size that fits in GPU memory.

FINDING 4: Workers – 4 Workers Optimal, 22.6 samp/s (p.8)

COMPARED: 0, 4, 8, 12, 16 parallel CPU worker processes.

FOUND: Throughput peaks at 4 workers (2.6x over single-process).

P95 (slowest 5% of batches): 750ms at 4w → 1184ms at 8w. Slow batches delay GPU training – the GPU has nothing to do until data arrives.

ROOT CAUSE: More workers = more simultaneous disk reads = filesystem contention.

FIX: Set num_workers=4 in training config.

FINDING 5: Batch Size Has Minimal Impact on Throughput (p.9-12)

COMPARED: batch sizes 1, 2, 4, 8, 16 at 4, 8, 12 workers.

FOUND: Throughput is nearly flat (~19 samp/s, ±16%). Best: batch_size=1 (20.2 samp/s). Worst: batch_size=16 (17.1 samp/s).

P95 latency: 256ms (bs=1) → 2386ms (bs=16). Larger batches wait for the slowest sample, stalling the GPU.

ROOT CAUSE: The bottleneck is per-patch disk IO (read + decompress), not batching overhead. Grouping more patches doesn't make each read faster.

FIX: Batch size can be whatever the training recipe calls for – it won't affect data loading speed.

FINDING 6: Cache – 1 GB TensorStore Cache Provides NO Benefit (p.9)

COMPARED: Cold cache (empty, first run) vs warm cache (holds data from prior run).

FOUND: Cold (106ms) vs warm (121ms) – nearly identical. If cache helped, warm should be noticeably faster. It is not.

ROOT CAUSE: Random sampling reads different chunks each time. Cached data is evicted (removed) before it can be reused.

FIX: Reduce cache to save memory, or use locality-aware sampling.

FINDING 7: Format Comparison – zarr3 sharded (128c) Is 3.7x Faster

COMPARED: 4 storage formats on same Jiefu 536 GB dataset, 128³ patch, 100 samples.

FOUND: Disk Read & Decompress: zarr2 (gzip): 176ms, zarr3 no-shard (zstd): 176ms, zarr3 sharded (256c): 120ms, zarr3 sharded (128c): 47ms.

Read amplification: zarr2 (gzip): 13.1x, zarr3 no-shard (zstd): 13.3x, zarr3 sharded (256c): 10.9x, zarr3 sharded (128c): 3.3x.

IO bandwidth: zarr2 (gzip): 45 MB/s, zarr3 no-shard (zstd): 45 MB/s, zarr3 sharded (256c): 66 MB/s, zarr3 sharded (128c): 167 MB/s.

ROOT CAUSE: Chunk-patch alignment (128³ chunks for 128×128×32 patches) eliminates most wasted reads.

Sharding alone (256c) gives 1.5x; alignment alone (128c) gives 3.7x.

Codec change (gzip→zstd) without sharding gives 0x benefit.

FIX: Convert training data to zarr3 sharded with chunk size matching patch size.

FINDING 8: Chunk Size Dominates – z2 c64 Is Fastest

COMPARED: 9 chunk/shard configs on 2048³ Jiefu crop (image only), 128³ patch, 100 samples.

FOUND: Fastest: z2 c64 (13ms, 2.9x amplif.).

Slowest: z3 s512 c256 (100ms, 25.5x amplif.).

ROOT CAUSE: Chunk size determines how much wasted data is read per patch.

64³ chunks: 2.9-3.3x amplif. | 128³: 6.6-7.1x | 256³: 25-28x.

Sharding/shard-size has minimal effect on this crop size.

FIX: Use smallest chunk size ≥ patch size for best alignment.

FINDING 9: Read Amplification Varies by Format AND Patch Size (p.10)

COMPARED: All configs – patch sizes (64³, 128³, 256³ on Jiefu+H01) and single-dataset runs (Jiefu zarr2, H01 zarr3 sharded) with 128³ patches.

FOUND: Amplification ranges from 2.1x (h01) to 40.6x (patch64).

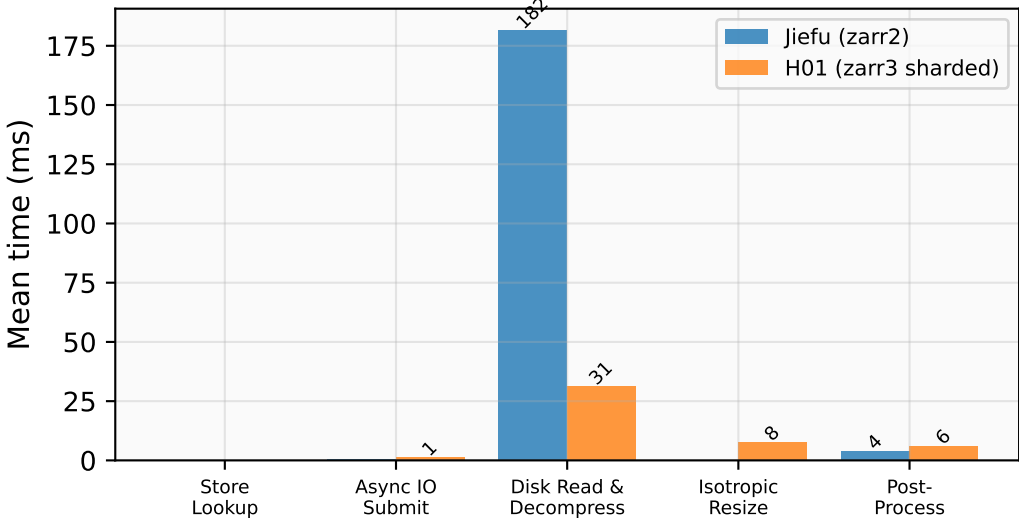
ROOT CAUSE: (1) zarr2 stores each chunk as a separate file → more waste than zarr3 sharded; (2) small patches cross more chunk boundaries per byte.

FIX: Convert to zarr3 sharded. Use larger patches when possible.

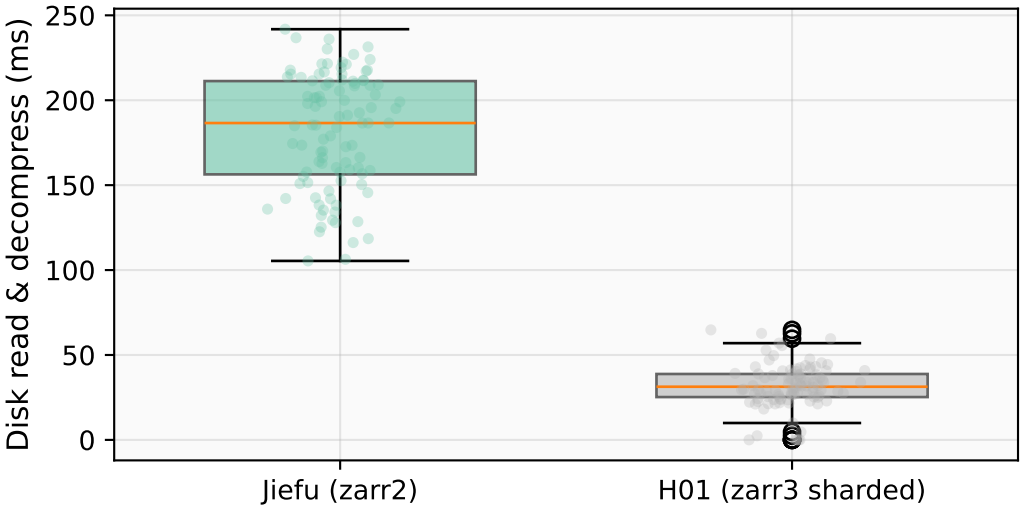
Comparison: Jiefu (zarr2) vs H01 (zarr3 sharded)

WHAT: Compare two storage formats reading the same patch size. Jiefu = zarr2 (unsharded, separate chunk files), H01 = zarr3 (sharded, chunks colocated). Same patch [128,128,32], 3 scales, isotropic=true, 100 samples.
WHY: Isolate how much storage format affects read speed.

Per-Stage Mean Latency



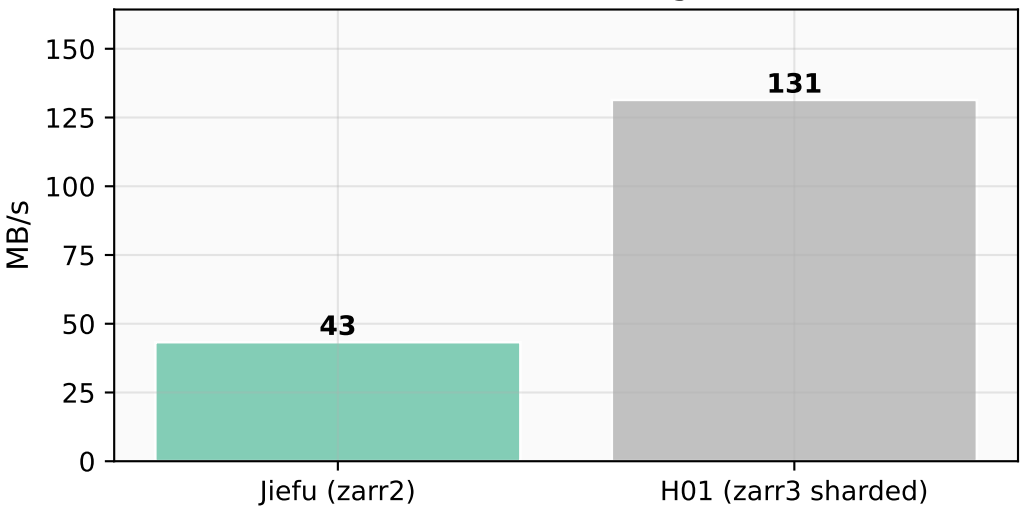
Disk Read & Decompress Distribution



Summary Statistics

Metric	Jiefu (zarr2)	H01 (zarr3 sharded)
Store Lookup	0.2 ms	0.2 ms
Async IO Submit	0.4 ms	1.3 ms
Disk Read & Decompress	181.6 ms	31.4 ms
Isotropic Resize	0.0 ms	7.8 ms
Post- Process	4.1 ms	6.2 ms
TOTAL	187.2 ms	51.3 ms
Bytes/sample	7.9 MB	4.1 MB
Isotropic	True	True
Patch size	[128, 128, 32]	[128, 128, 32]

Effective IO Bandwidth (higher = faster)



FINDING: Jiefu (zarr2) is 6x slower than H01 (zarr3).

Per-Stage Mean Latency: Jiefu's Disk Read & Decompress (182ms) dominates because zarr2 has 13x read amplification — it reads 103 MB from disk for 7.9 MB of useful data. Isotropic Resize is 0.0ms for Jiefu (FIB-SEM data has equal-sized voxels, so no resampling needed) vs 7.8ms for H01 (serial-section TEM data has unequal voxels and needs resampling).

Distribution: Jiefu has wide spread because each patch lands differently on chunk boundaries — some patches fit neatly inside chunks (fast), others span many chunks and force reading extra data that gets thrown away (slow). H01's sharding packs chunks together, giving consistent latency.

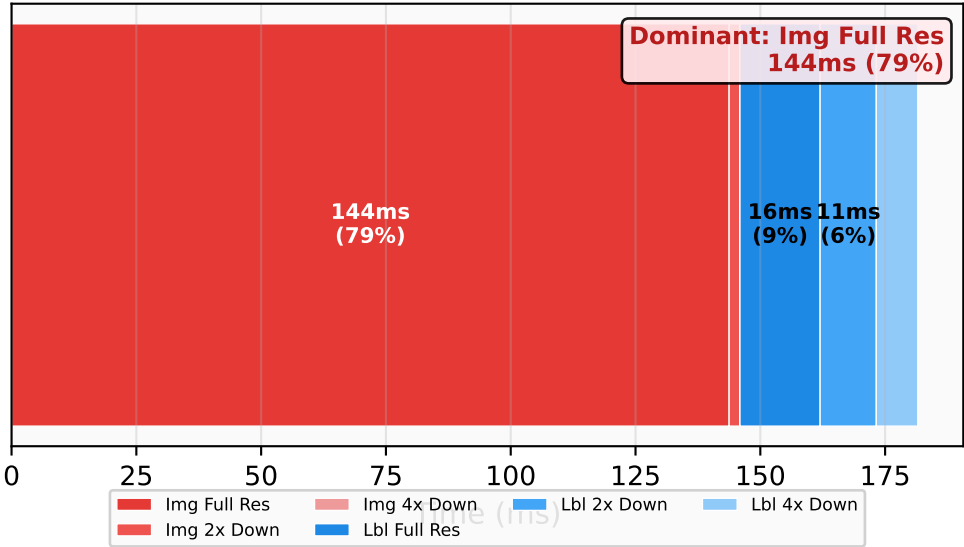
IO Bandwidth: H01 (131 MB/s) vs Jiefu (43 MB/s) — H01 delivers data 3x faster because sharding means less wasted reading per useful byte.

FIX: Convert Jiefu to zarr3 with sharding, or re-chunk to align with patch size.

Disk Read & Decompress — Jiefu Cerebellum (zarr2)

WHAT: Break down the Disk Read & Decompress stage into sub-parts for Jiefu (zarr2).
WHY: We know disk reading is the bottleneck — this shows exactly where the time goes inside it.

Where Does the 182ms Go?



CPU vs IO Analysis

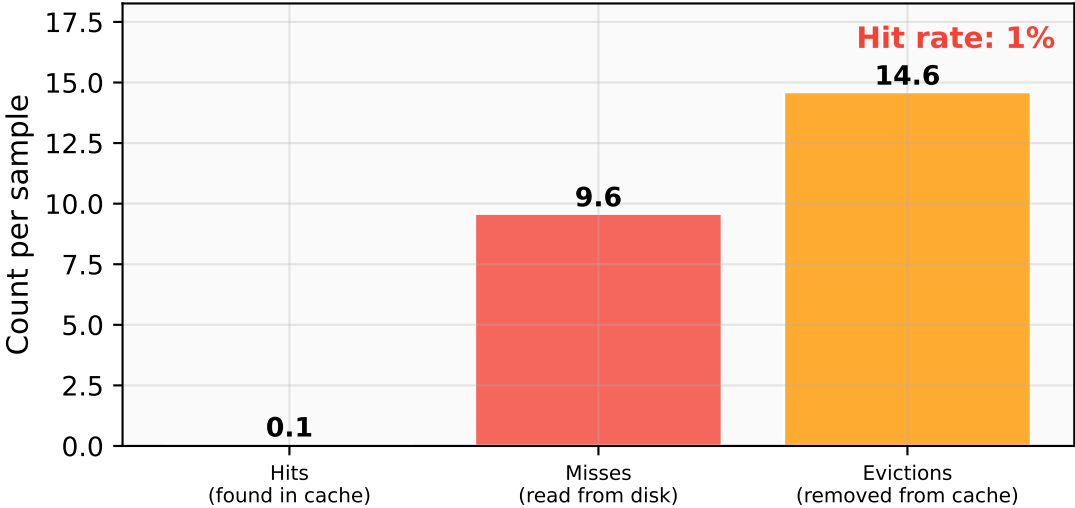
Wall time (disk read+decomp): 181.6 ms
CPU time (all threads): 2726.0 ms
Multi-thread Decompression: 15.0x
(TensorStore uses 15 CPU cores at once to decompress chunks in parallel)

User CPU: 2431.1 ms (decompression)
System CPU: 294.9 ms (kernel/IO)

Major page faults: 0.0/sample
(data fetched from NFS — 0 means cached)
Minor page faults: 5308/sample
(data served from OS page cache — fast)

INTERPRETATION:
CPU time exceeds wall time because 15 cores decompress simultaneously.
Bottleneck: CPU decompression, not NFS network transfer.

TensorStore In-Memory Cache (1 GB)



Key Findings

STACKED BAR (top-left):
Image: 146ms (80%), Label: 36ms (20%)
L0 (full res) dominates — it has the most voxels to read and decompress.

CPU vs IO (top-right):
15x multi-thread decompression = 15 CPU cores decompress at once.
Bottleneck = CPU decompression.

CACHE (bottom-left):
1% hit rate (hits = found in cache, misses = had to read from disk, evictions = old data removed for new).
15 evictions > 10 misses = cache is full, constantly replacing data that won't be reused.

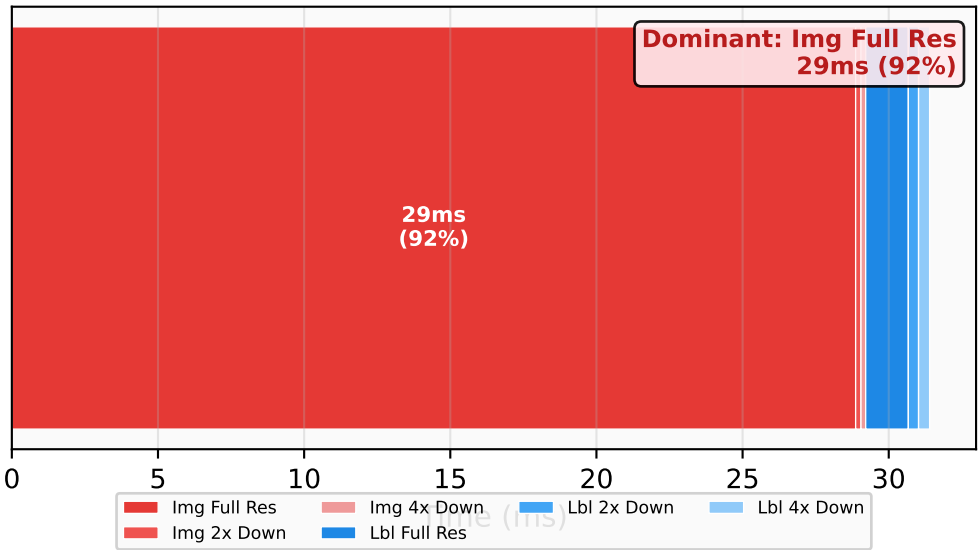
DISK IO STATS:
Disk: 103.2 MB, Need: 7.9 MB
Amplification: 13.1x (massive waste)

FINDING: Jiefu reads 103 MB from disk for 7.9 MB of data (13x read amplification). 15 CPU cores decompress in parallel (15.0x).
CACHE: TensorStore keeps recently read data in a 1 GB memory buffer so repeated reads skip disk. Jiefu's 1% hit rate: zarr2 reads one small chunk per file, so each random patch reads different files with no overlap. H01's 10%: zarr3 packs many chunks into one shard file — sharding causes accidental reuse: you read a shard for chunk A, and chunk B happens to be in the same shard, so it's already cached when needed later. Both rates are too low to help.
COMMON WITH H01: Both bottlenecked by CPU decompression, not NFS network speed.
DIFFERENT: Jiefu has 13x amplification (vs H01 ~1x) because zarr2's separate chunk files force reading entire chunks when patches cross boundaries.
FIX: Use a faster decompression codec (zstd instead of gzip), reduce read amplification by converting to zarr3 sharded, or increase CPU cores available per worker.

Disk Read & Decompress — H01 Human Cortex (zarr3 sharded)

WHAT: Break down the Disk Read & Decompress stage into sub-parts for H01 (zarr3 sharded).
WHY: Compare with Jiefu (previous page) to show how zarr3 sharding reduces read amplification.

Where Does the 31ms Go?



CPU vs IO Analysis

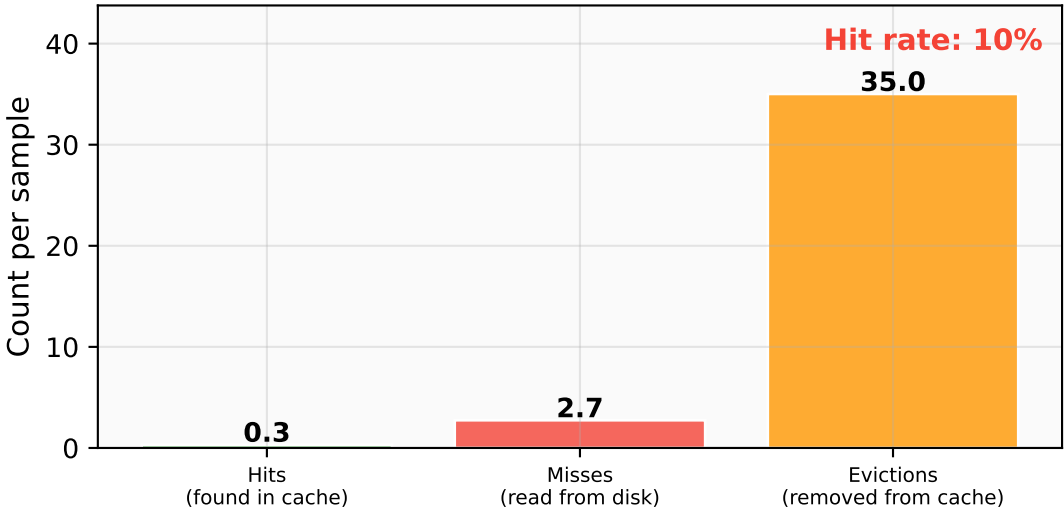
Wall time (disk read+decomp): 31.4 ms
CPU time (all threads): 549.8 ms
Multi-thread Decompression: 17.5x
(TensorStore uses 17 CPU cores at once to decompress chunks in parallel)

User CPU: 501.2 ms (decompression)
System CPU: 48.6 ms (kernel/IO)

Major page faults: 0.0/sample
(data fetched from NFS — 0 means cached)
Minor page faults: 5153/sample
(data served from OS page cache — fast)

INTERPRETATION:
CPU time exceeds wall time because 17 cores decompress simultaneously.
Bottleneck: CPU decomposition, not NFS network transfer.

TensorStore In-Memory Cache (1 GB)



Key Findings

STACKED BAR (top-left):
Image: 29ms (93%), Label: 2ms (7%)
L0 (full res) dominates — it has the most voxels to read and decompress.

CPU vs IO (top-right):
17x multi-thread decompression = 17 CPU cores decompress at once.
Bottleneck = CPU decomposition.

CACHE (bottom-left):
10% hit rate (hits = found in cache, misses = had to read from disk, evictions = old data removed for new).
35 evictions > 3 misses
= cache is full, constantly replacing data that won't be reused.

DISK IO STATS:
Disk: 8.5 MB, Need: 4.1 MB
Amplification: 2.1x (moderate waste)

FINDING: H01 reads 8.5 MB from disk for 4.1 MB of actual data needed (2.1x amplification). 'MB' = megabytes read by TensorStore's file operations; 'data needed' = the raw patch voxels requested. All times are in ms (milliseconds).

CACHE: H01's 10% hit rate vs Jiefu's 1%: zarr3 packs many chunks into one shard file, so reading one chunk caches its neighbors for free. The next random patch may land on a neighbor already in cache — sharding causes accidental reuse: you read a shard for chunk A, and chunk B in the same shard is already cached when needed later. Zarr2 reads one chunk per file — no free neighbors. Both rates still too low to matter.

COMMON WITH JIEFU: Both bottlenecked by CPU decompression (17 cores active).

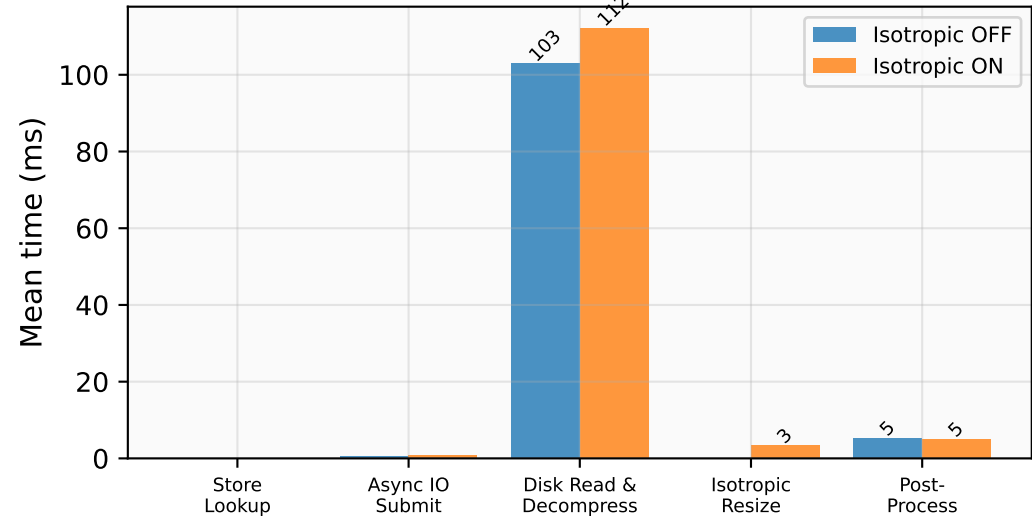
DIFFERENT: H01's sharding reduces amplification — reads 8.5 MB vs Jiefu's 103 MB.

FIX: Use faster codec (zstd), reduce CPU load by lowering compression level, or increase CPU cores per worker.

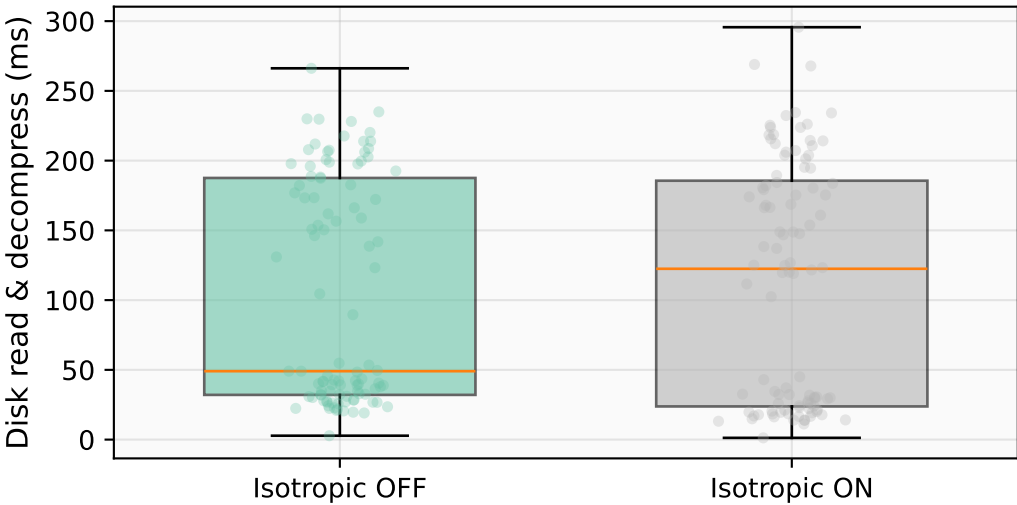
Comparison: Isotropic OFF vs Isotropic ON

WHAT: Compare isotropic ON vs OFF on the same two volumes (Jiefu + H01). Both use patch [128,128,32], 3 scales, 100 samples.
WHY: The real training pipeline uses isotropic mode — this measures the extra cost of resampling anisotropic data to equal voxel spacing via F.interpolate.

Per-Stage Mean Latency



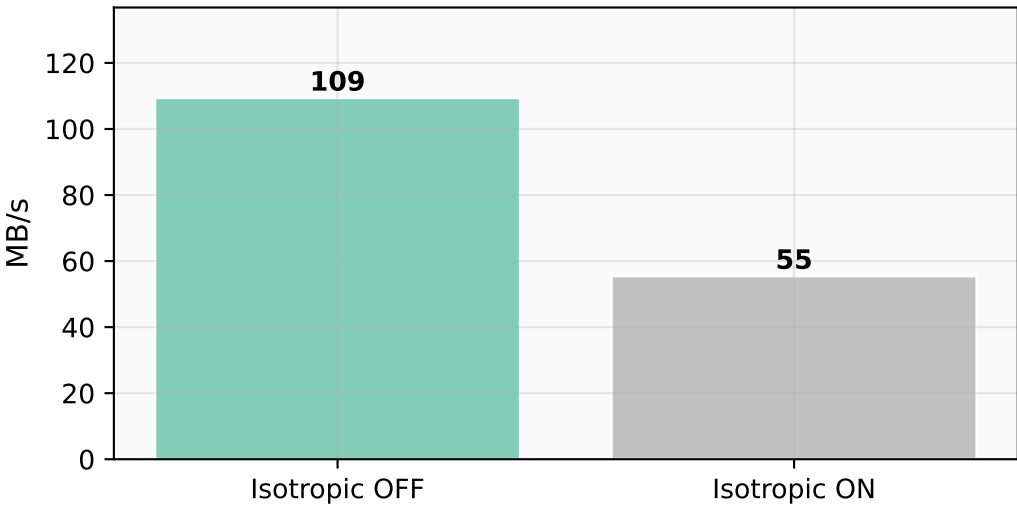
Disk Read & Decompress Distribution



Summary Statistics

Metric	Isotropic OFF	Isotropic ON
Store Lookup	0.2 ms	0.2 ms
Async IO Submit	0.5 ms	0.8 ms
Disk Read & Decompress	102.9 ms	112.1 ms
Isotropic Resize	0.0 ms	3.4 ms
Post- Process	5.4 ms	5.1 ms
TOTAL	110.8 ms	124.2 ms
Bytes/sample	11.3 MB	6.2 MB
Isotropic	False	True
Patch size	[128, 128, 32]	[128, 128, 32]

Effective IO Bandwidth (higher = faster)

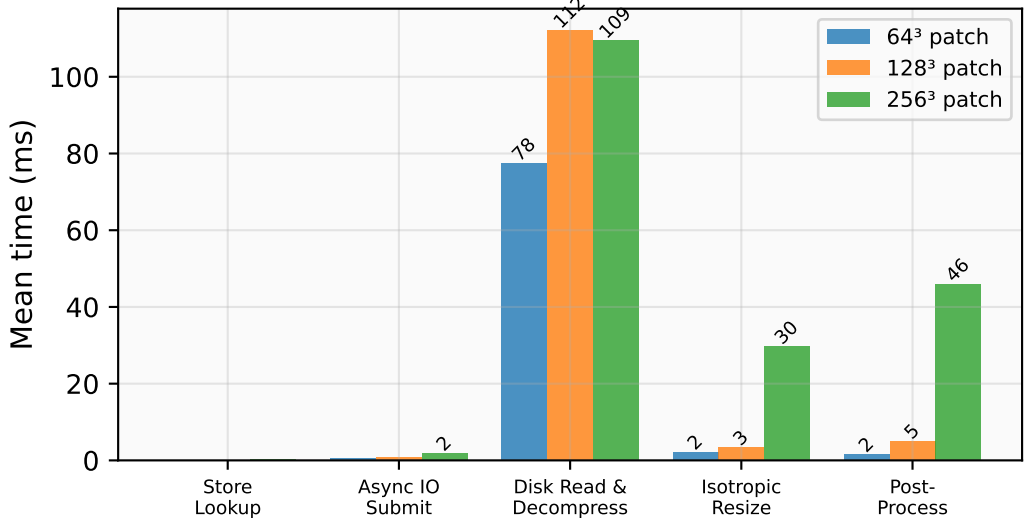


FINDING: Isotropic mode adds 3.4ms (2.7%) from F.interpolate. Data from both Jiefu (zarr2) and H01 (zarr3) combined.
Per-Stage: iso=true (124ms) vs iso=false (111ms) — the extra time is entirely in the Isotropic Resize stage (3.4ms vs 0ms).
Distribution: Disk Read & Decompress looks almost the same — isotropic mode does not affect how data is read from disk, only what happens after (CPU resampling).
IO Bandwidth (higher = faster): noiso (109 MB/s) vs iso (55 MB/s) — a 2.0x gap. Both read the same bytes from disk, but iso spends extra CPU time resampling after reading. This lowers the overall 'data delivered per second' number, even though disk speed is the same.
CONCLUSION: Isotropic resampling cost is negligible. Issue #8 (continuous scale sampling) is computationally feasible.

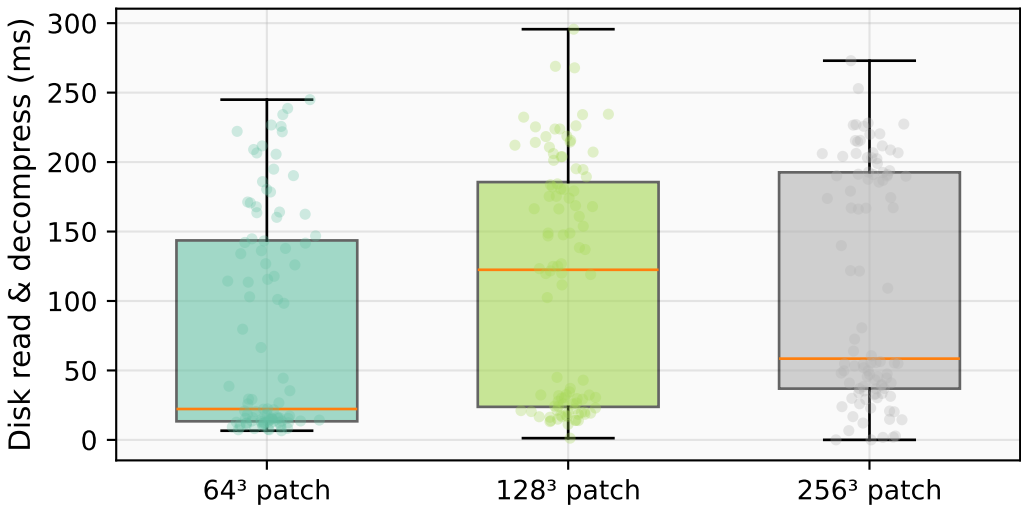
Comparison: 64³ patch vs 128³ patch vs 256³ patch

WHAT: Compare three patch sizes (64³, 128³, 256³) on the same volumes (Jiefu + H01). All use 3 scales, isotropic=true, 100 samples.
WHY: Larger patches read more data but may be more efficient per byte. This shows the optimal patch size for IO efficiency.

Per-Stage Mean Latency



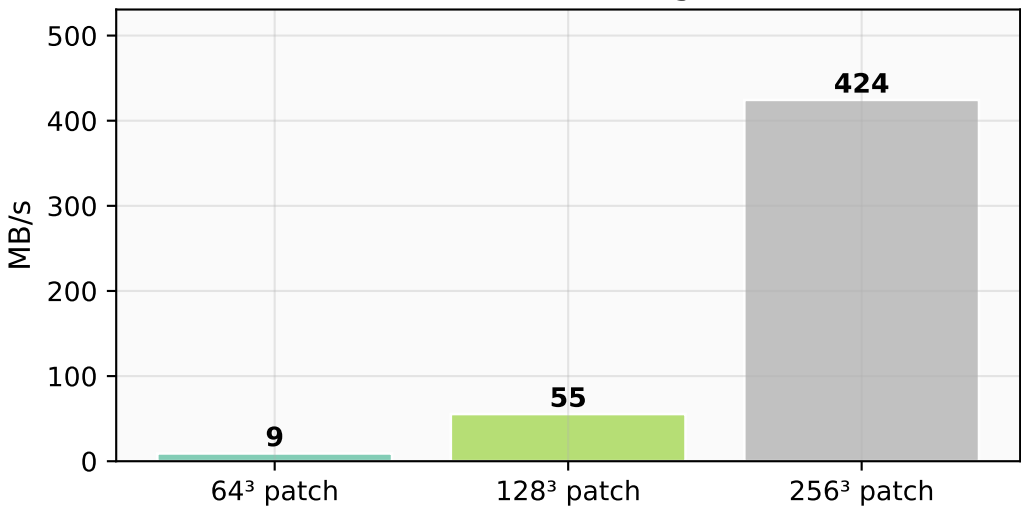
Disk Read & Decompress Distribution



Summary Statistics

Metric	64 ³ patch	128 ³ patch	256 ³ patch
Store Lookup	0.2 ms	0.2 ms	0.2 ms
Async IO Submit	0.5 ms	0.8 ms	2.0 ms
Disk Read & Decompress	77.5 ms	112.1 ms	109.5 ms
Isotropic Resize	2.3 ms	3.4 ms	29.8 ms
Post- Process	1.6 ms	5.1 ms	46.1 ms
TOTAL	85.9 ms	124.2 ms	200.4 ms
Bytes/sample	0.7 MB	6.2 MB	46.5 MB
Isotropic	True	True	True
Patch size	[64, 64, 16]	[128, 128, 32]	[256, 256, 64]

Effective IO Bandwidth (higher = faster)



FINDING: Larger patches are dramatically more IO-efficient.

Per-Stage: Disk Read & Decompress: 78ms → 112ms → 109ms (64³→128³→256³). Larger patches take longer in absolute time but read proportionally less wasted data.

Distribution: Latency spread varies with patch size — smaller patches are more sensitive to how they land on chunk boundaries, causing more variation between samples.

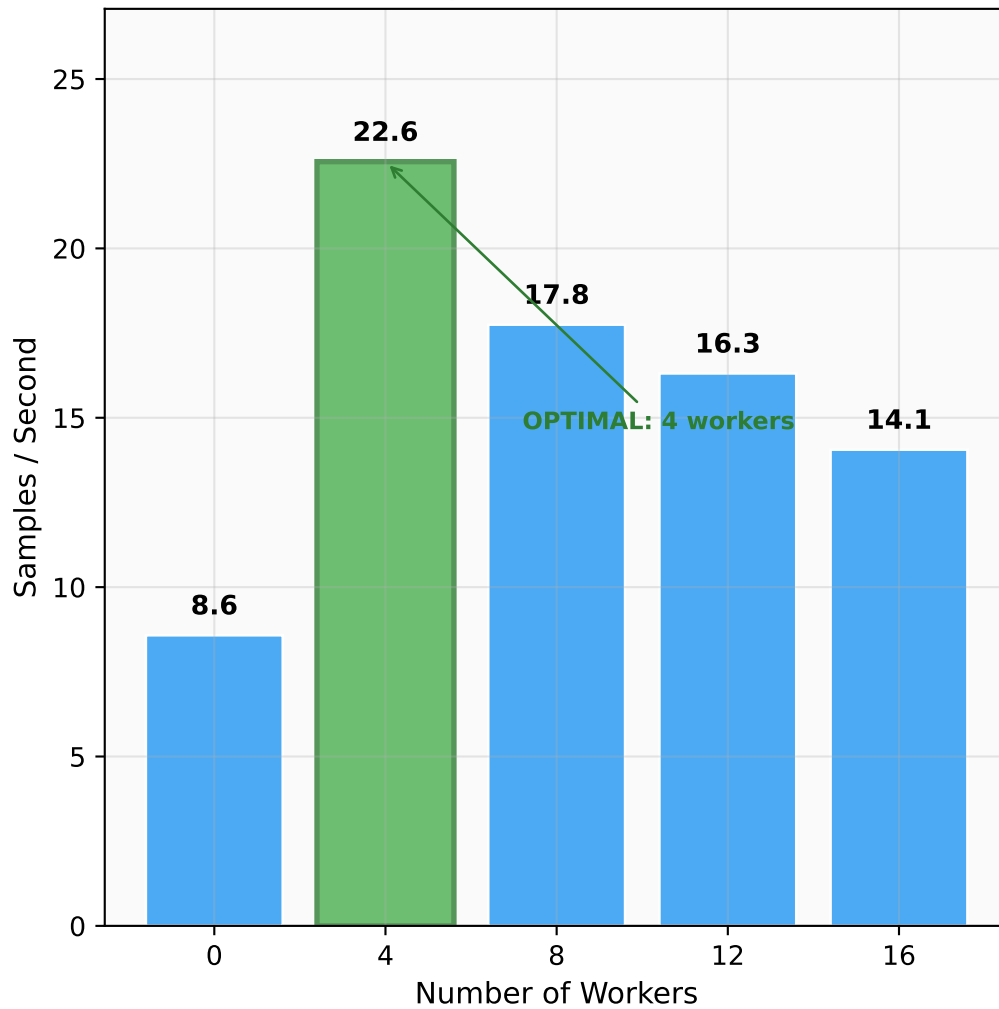
IO Bandwidth: 9 MB/s → 55 MB/s → 424 MB/s — a 47x improvement. Larger patches spread the fixed cost of opening and reading chunk files over more useful data, so each byte of actual data costs less IO time.

FIX: Use the largest patch size that fits in GPU memory.

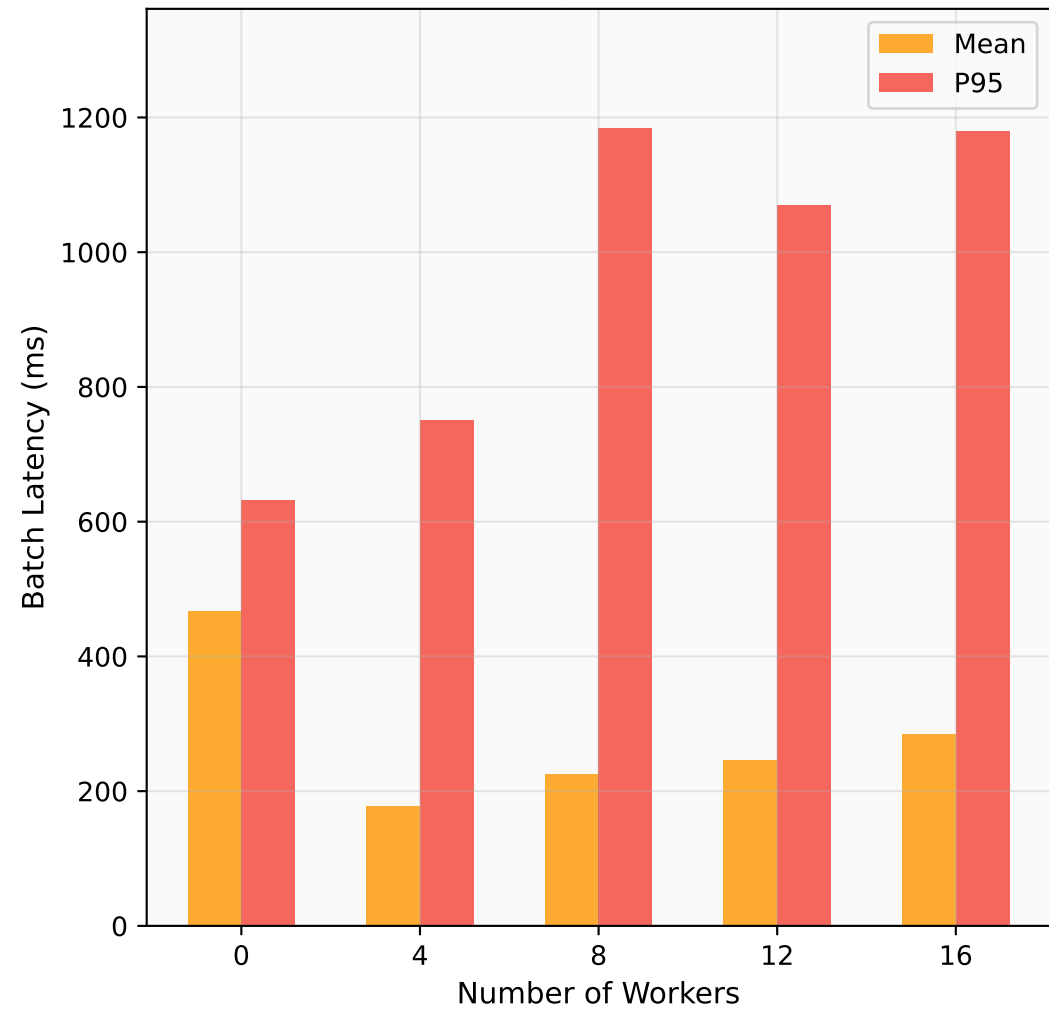
DataLoader Throughput vs. Worker Count

WHAT: Run the data loader with 0, 4, 8, 12, 16 parallel worker processes (batch_size=4) and measure throughput. 100 samples each.
WHY: Find the optimal number of workers before NFS contention degrades performance.

Throughput (higher is better)



Batch Latency (lower is better)



FINDING: 4 workers is optimal (22.6 samp/s, 2.6x over single-process). Note: these workers run on CPU — they read and prepare data, then hand it to the GPU for training. miao's data loading pipeline uses CPU workers; the GPU only does model training.

Throughput (left plot): Samples delivered per second. Peaks at 4 workers then drops — more workers means more simultaneous disk reads, which overwhelms NFS.

Batch Latency (right plot): Time to fetch one batch (group of samples). 'Mean' = average time. 'P95' = the slowest 5% of batches. P95 matters because during training, the GPU has nothing to do until the slowest batch arrives — one slow batch delays the entire training step. P95 at 4w: 750ms. P95 latency explodes: 750ms at 4w → 1184ms at 8w.

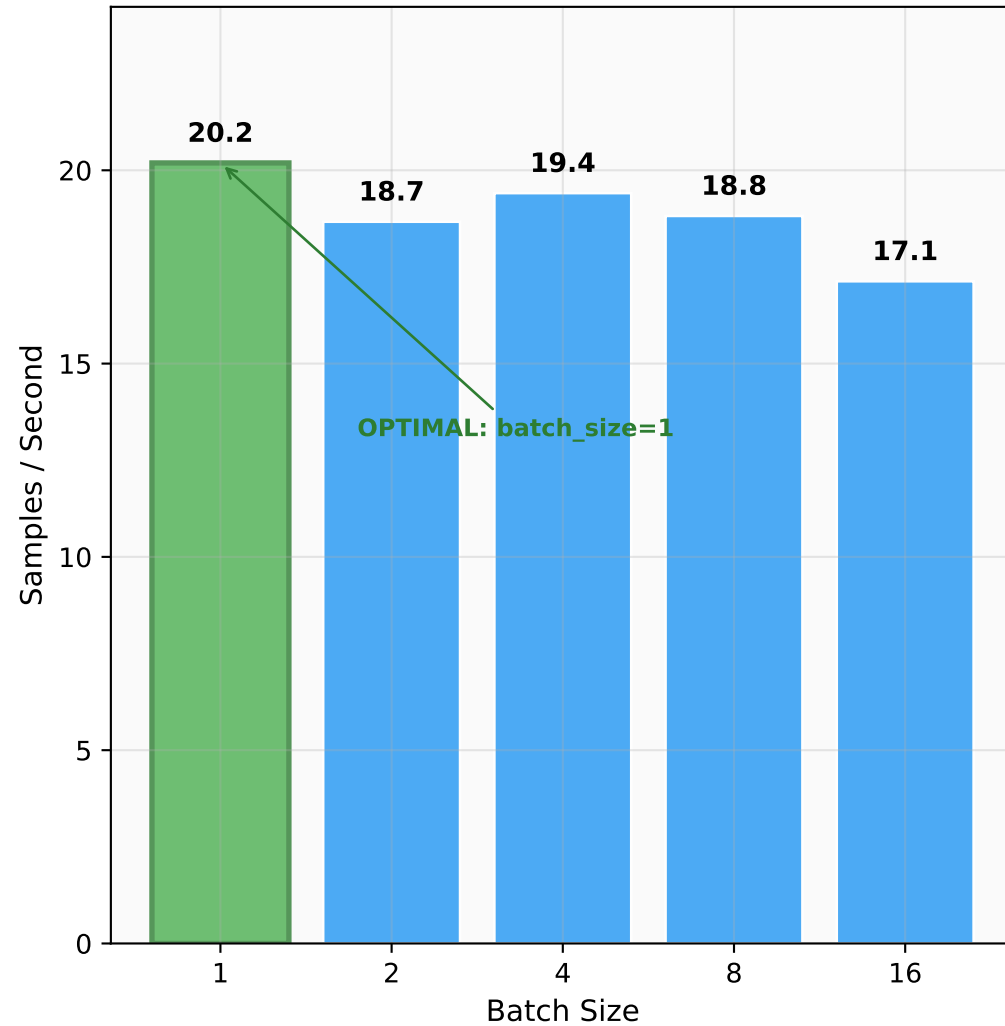
Results may vary between runs due to NFS load fluctuations, but 4 workers is consistently optimal.

FIX: Set num_workers=4 in training config.

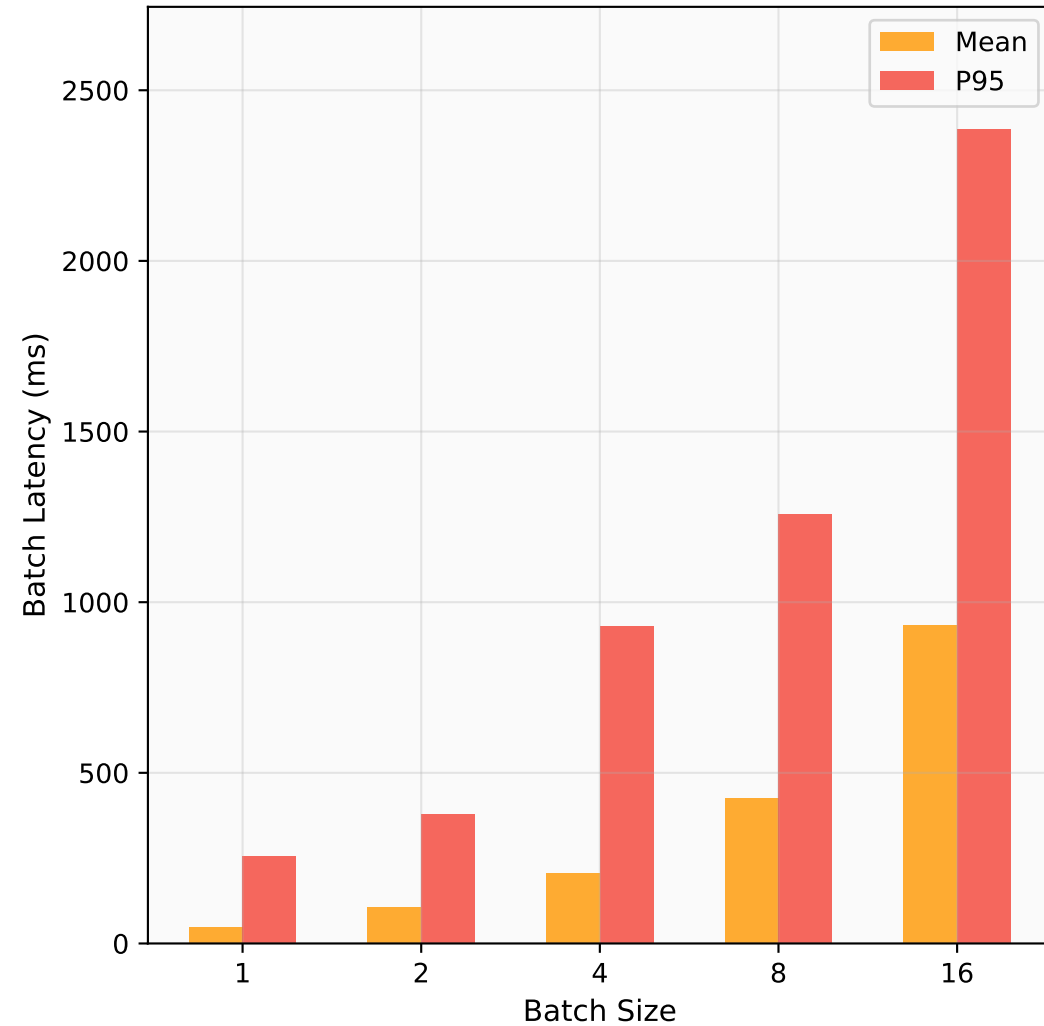
Batch Size Sweep (num_workers=4)

WHAT: Run the data loader with batch sizes 1, 2, 4, 8, 16 at 4 workers. 5 batch sizes tested.
WHY: Find the batch size that maximizes GPU utilization. Larger batches amortize per-batch overhead but use more GPU memory.

Throughput (higher is better)



Batch Latency (lower is better)

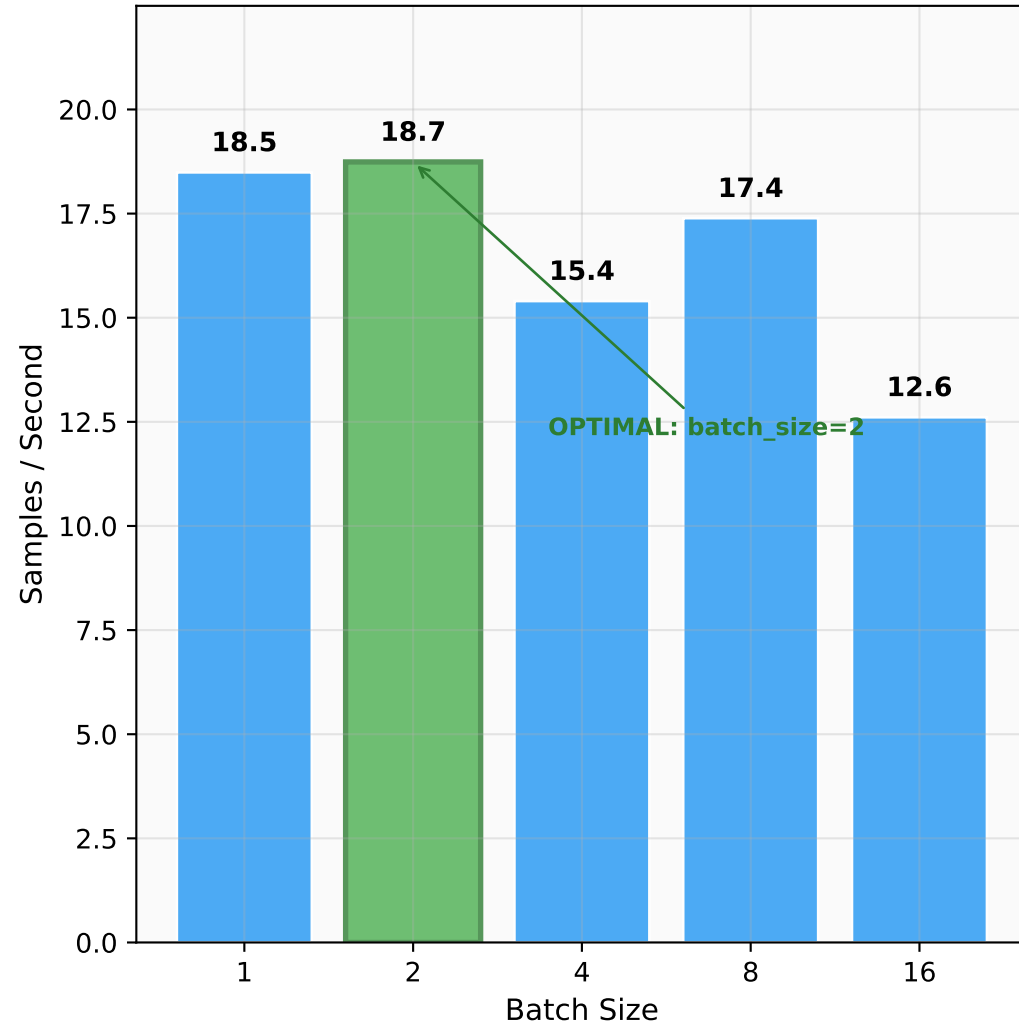


FINDING: Optimal batch_size=1 at 4 workers (20.2 samp/s). Worst: batch_size=16 (17.1 samp/s).
Throughput (left): Samples delivered per second. Batch size has minimal impact on throughput because IO is the primary bottleneck.
Latency (right): Per-batch latency scales with batch size — larger batches take longer but deliver more samples. P95 latency: 256ms (bs=1) → 2386ms (bs=16). High P95 stalls GPU training.
FIX: batch_size=1 balances throughput and latency.

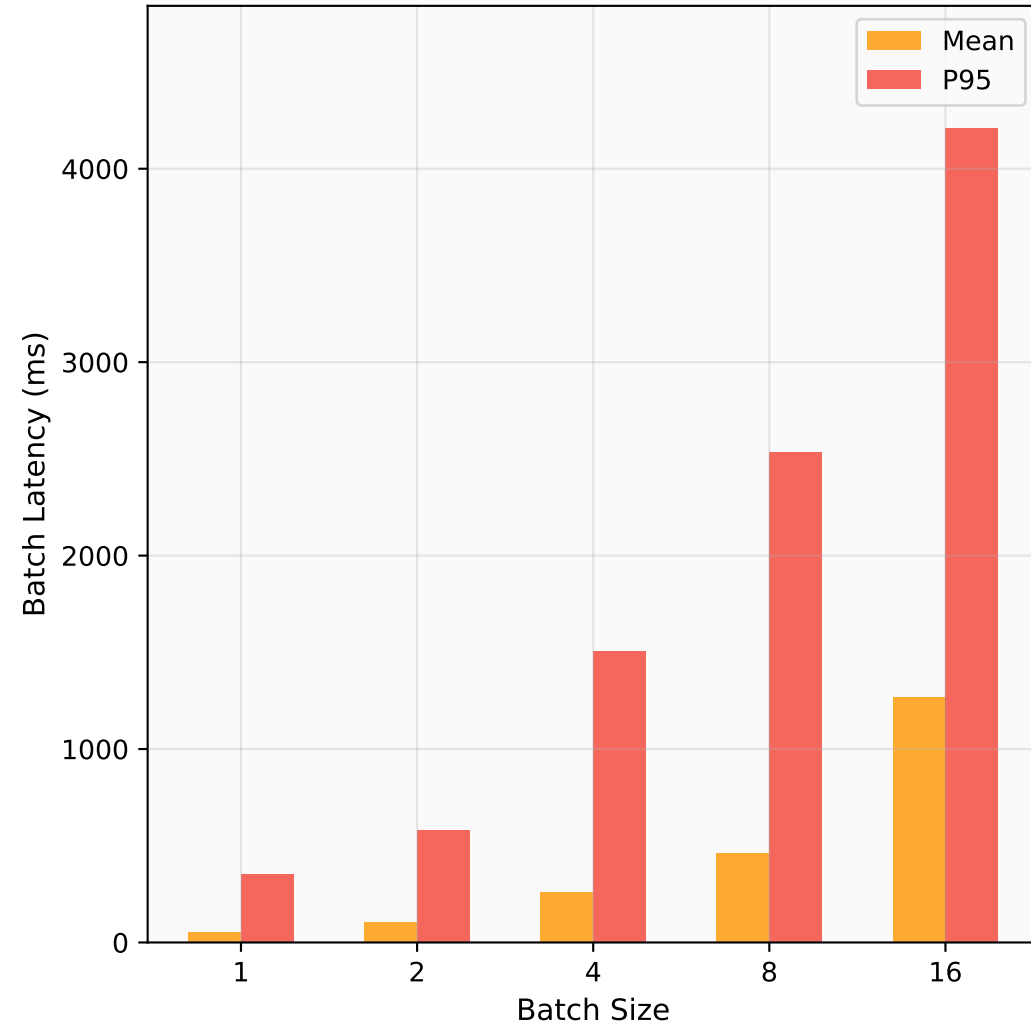
Batch Size Sweep (num_workers=8)

WHAT: Run the data loader with batch sizes 1, 2, 4, 8, 16 at 8 workers. 5 batch sizes tested.
WHY: Find the batch size that maximizes GPU utilization. Larger batches amortize per-batch overhead but use more GPU memory.

Throughput (higher is better)



Batch Latency (lower is better)

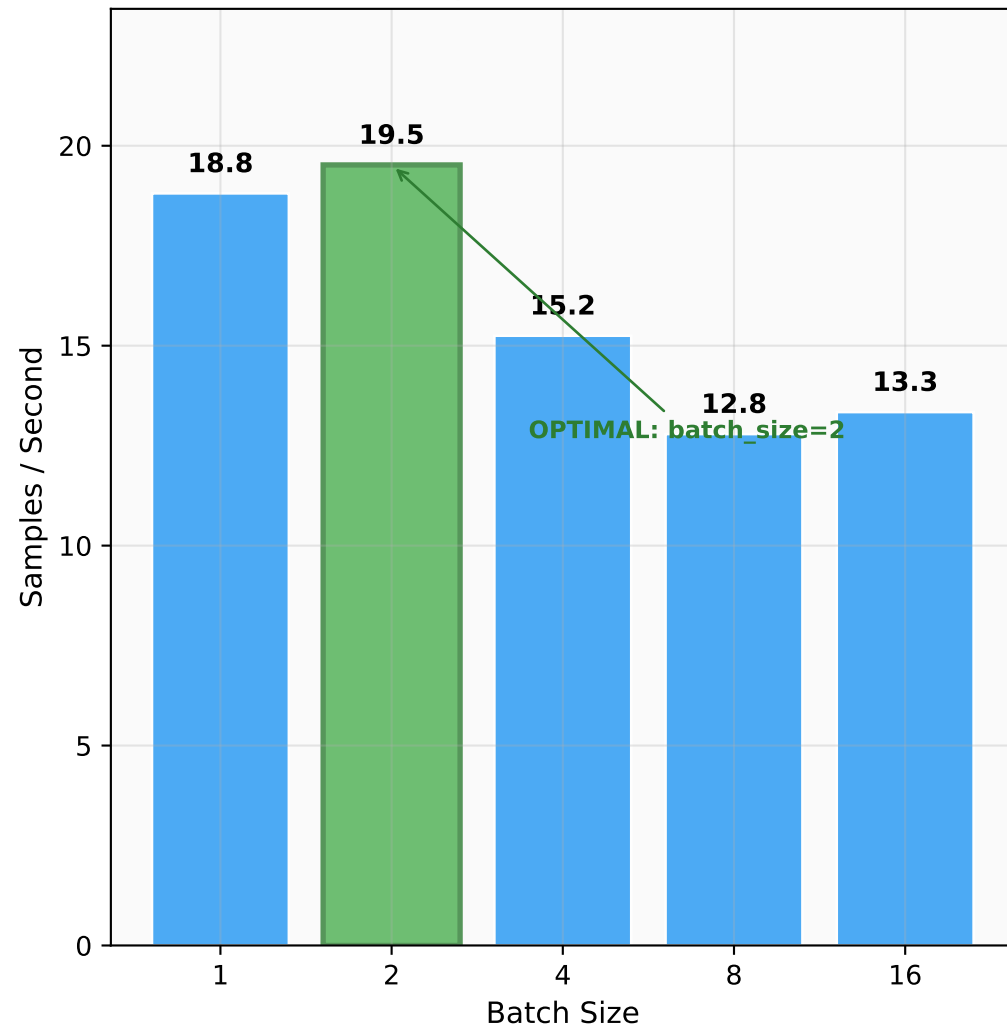


FINDING: Optimal batch_size=2 at 8 workers (18.7 samp/s). Worst: batch_size=16 (12.6 samp/s).
Throughput (left): Samples delivered per second. Batch size has minimal impact on throughput because IO is the primary bottleneck.
Latency (right): Per-batch latency scales with batch size — larger batches take longer but deliver more samples. P95 latency: 356ms (bs=1) → 4211ms (bs=16). High P95 stalls GPU training.
FIX: batch_size=2 balances throughput and latency.

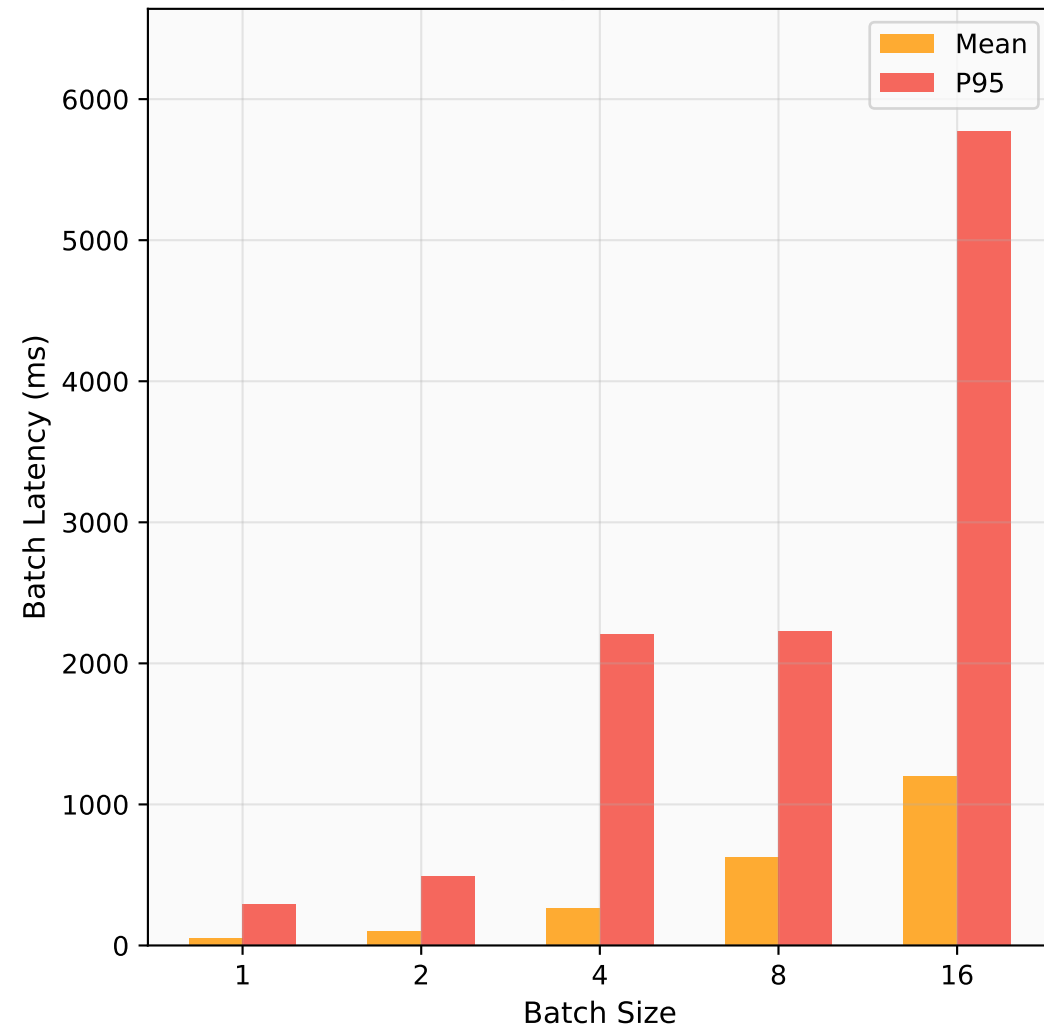
Batch Size Sweep (num_workers=12)

WHAT: Run the data loader with batch sizes 1, 2, 4, 8, 16 at 12 workers. 5 batch sizes tested.
WHY: Find the batch size that maximizes GPU utilization. Larger batches amortize per-batch overhead but use more GPU memory.

Throughput (higher is better)



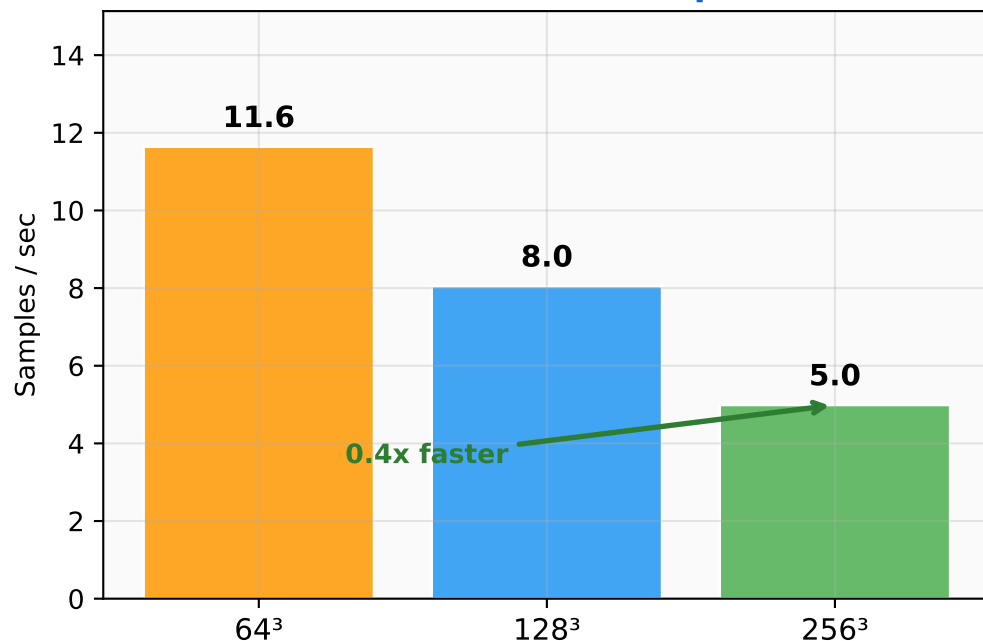
Batch Latency (lower is better)



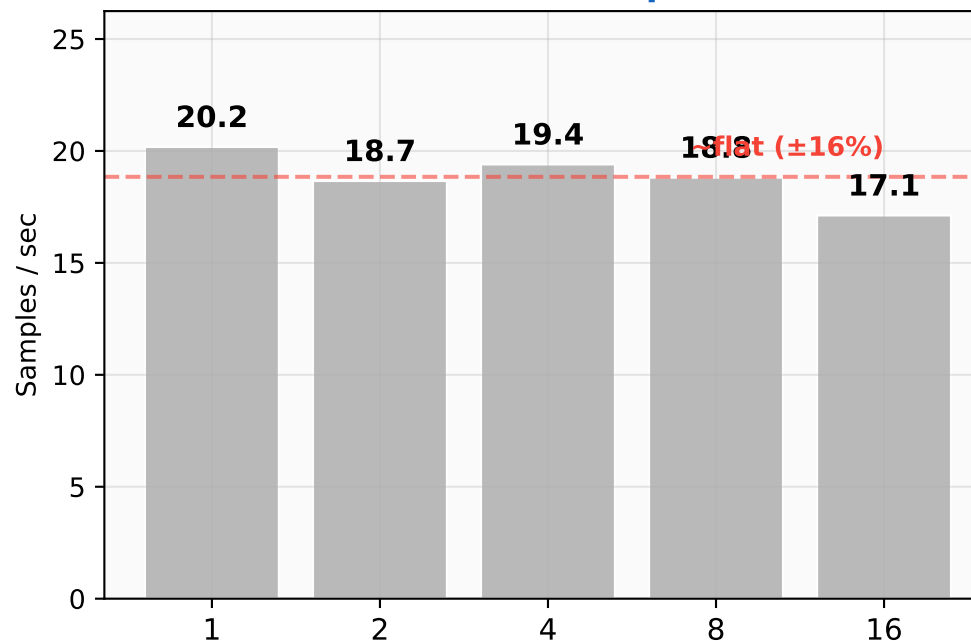
FINDING: Optimal batch_size=2 at 12 workers (19.5 samp/s). Worst: batch_size=8 (12.8 samp/s).
Throughput (left): Samples delivered per second. Batch size has significant impact on throughput because IO is the primary bottleneck.
Latency (right): Per-batch latency scales with batch size — larger batches take longer but deliver more samples. P95 latency: 294ms (bs=1) → 5774ms (bs=16). High P95 stalls GPU training.
FIX: batch_size=2 balances throughput and latency.

Why Patch Size Matters but Batch Size Doesn't

Patch Size Sweep



Batch Size Sweep (4w)



PATCH SIZE changes how much disk IO each sample needs:

64^3 patch: 11.6 samp/s → wasteful IO (reads whole chunks, uses ~5%)

128^3 patch: 8.0 samp/s → better (uses ~30% of each chunk)

256^3 patch: 5.0 samp/s → efficient (uses ~80% of each chunk)

Each sample reads a 3D crop from the zarr volume. A small crop touches many chunks but uses only a sliver of each = wasted reads.

A large crop fills whole chunks = efficient reads.

► Patch size controls IO COST PER SAMPLE. Bigger = less waste.

BATCH SIZE just changes how many samples are grouped for the GPU:

batch=1: 20.2 samp/s | batch=2: 18.7 samp/s | batch=4: 19.4 samp/s | batch=8: 18.8 samp/s | batch=16: 17.1 samp/s

The DataLoader reads N patches independently, stacks them into one tensor $[N, C, X, Y, Z]$, sends to GPU. Each patch still does the same disk IO regardless of how many are in the batch.

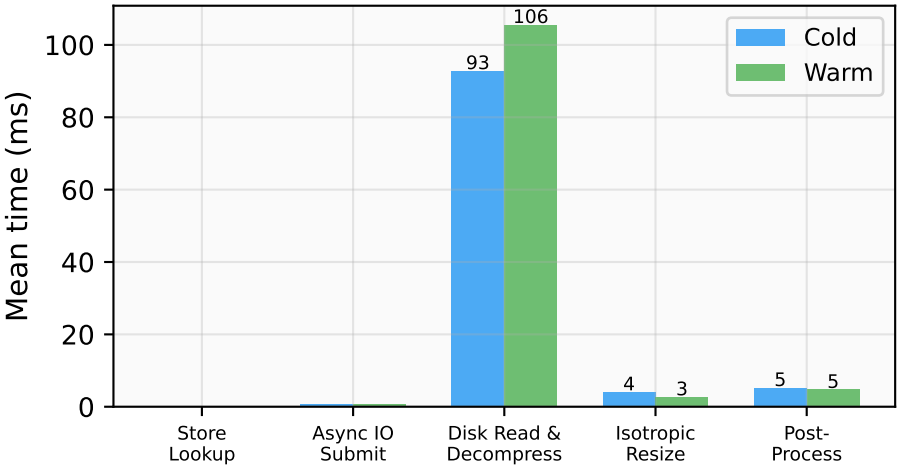
► Batch size controls GROUPING, not IO cost. The bottleneck (disk read + decompress) is per-patch, so grouping doesn't help.

CONCLUSION: To speed up data loading, make each read more efficient (larger patches, better storage format) – not bundle more reads together.

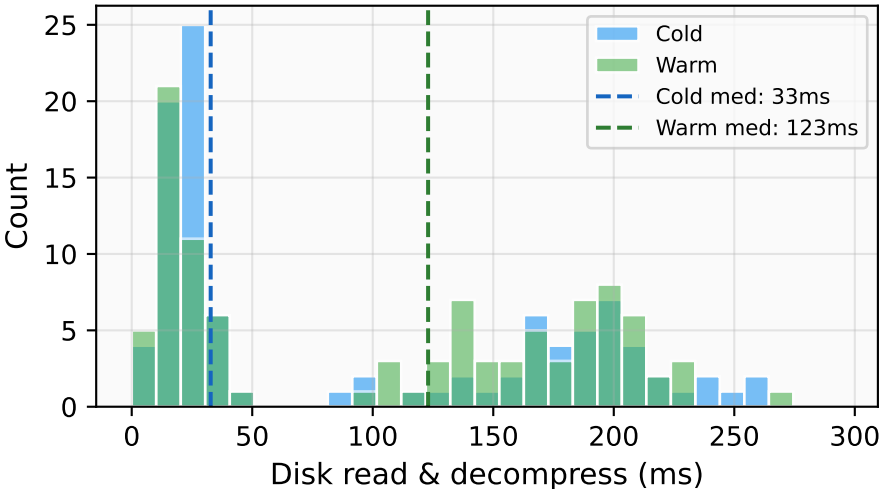
Cache Comparison: Cold vs Warm

WHAT: Run profiling twice — first with empty cache (cold), then with populated cache (warm). Same volumes, patch size, 100 samples each.
WHY: Test whether TensorStore's 1 GB in-memory cache helps when randomly sampling from large volumes.

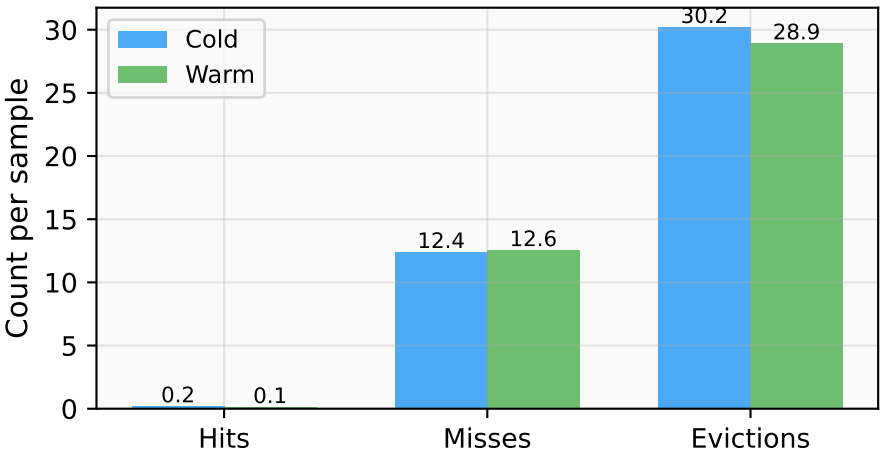
Per-Stage: Cold vs Warm



Disk Read & Decompress Distribution



TensorStore Cache Metrics



Cache Effect Summary

Metric	Cold	Warm	Change
Total (ms)	106.3	120.5	+13.4%
Disk read & decompress (ms)	92.8	105.6	+13.7%
TS file bytes (MB/sample)	46.8	61.3	+31.1%
TS read latency (ms)	3.81	4.11	+7.6%
Warm cache effect:	1.13x slower (cache adds overhead)		

FINDING: TensorStore's 1 GB in-memory cache provides NO benefit for random sampling.

WHAT IS THE CACHE? TensorStore keeps recently read data in a 1 GB memory buffer. If the same data is needed again, it can be served from memory (fast) instead of disk (slow). 'Cold' = empty cache (first run), 'Warm' = cache still holds data from the previous run.

Per-Stage (top-left): Cold (106ms) vs warm (121ms) — if the cache helped, warm should be noticeably faster. It is not — both are nearly identical.

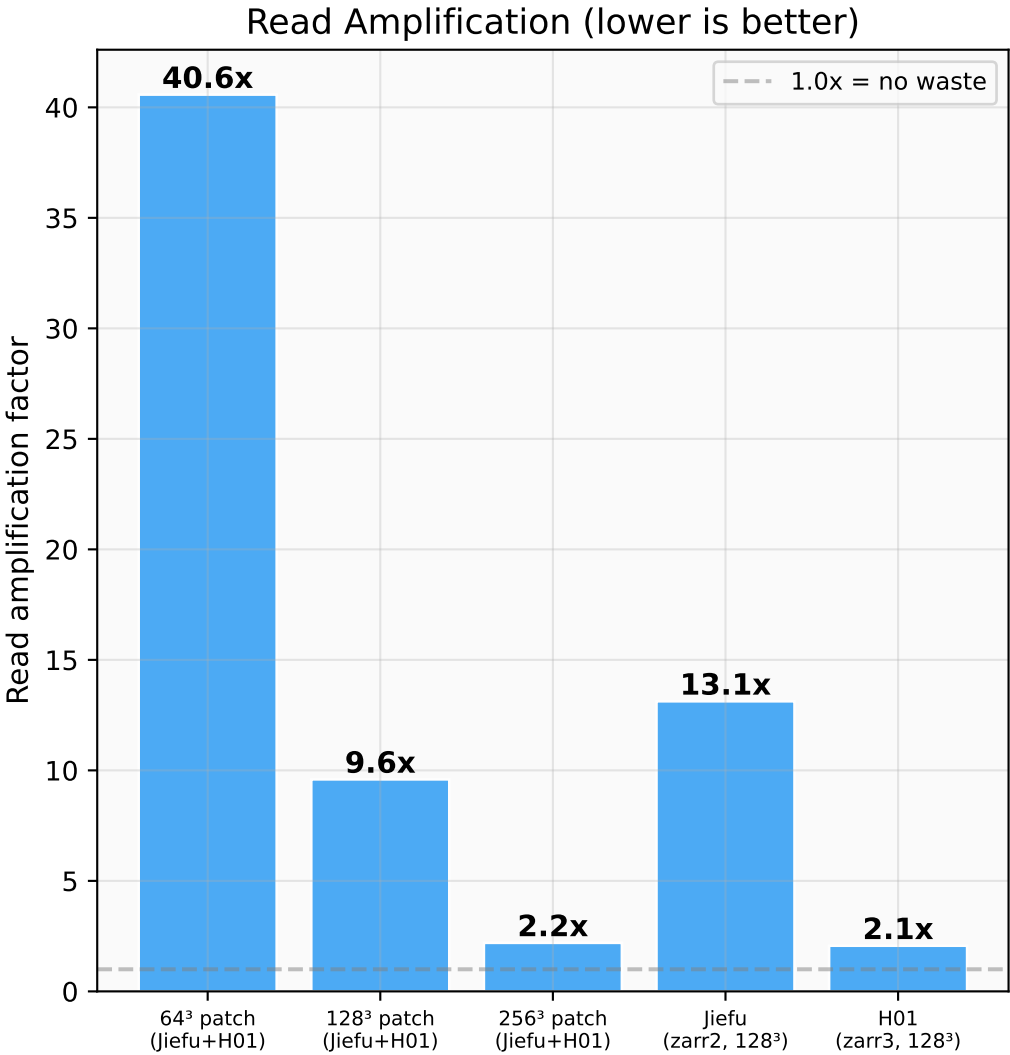
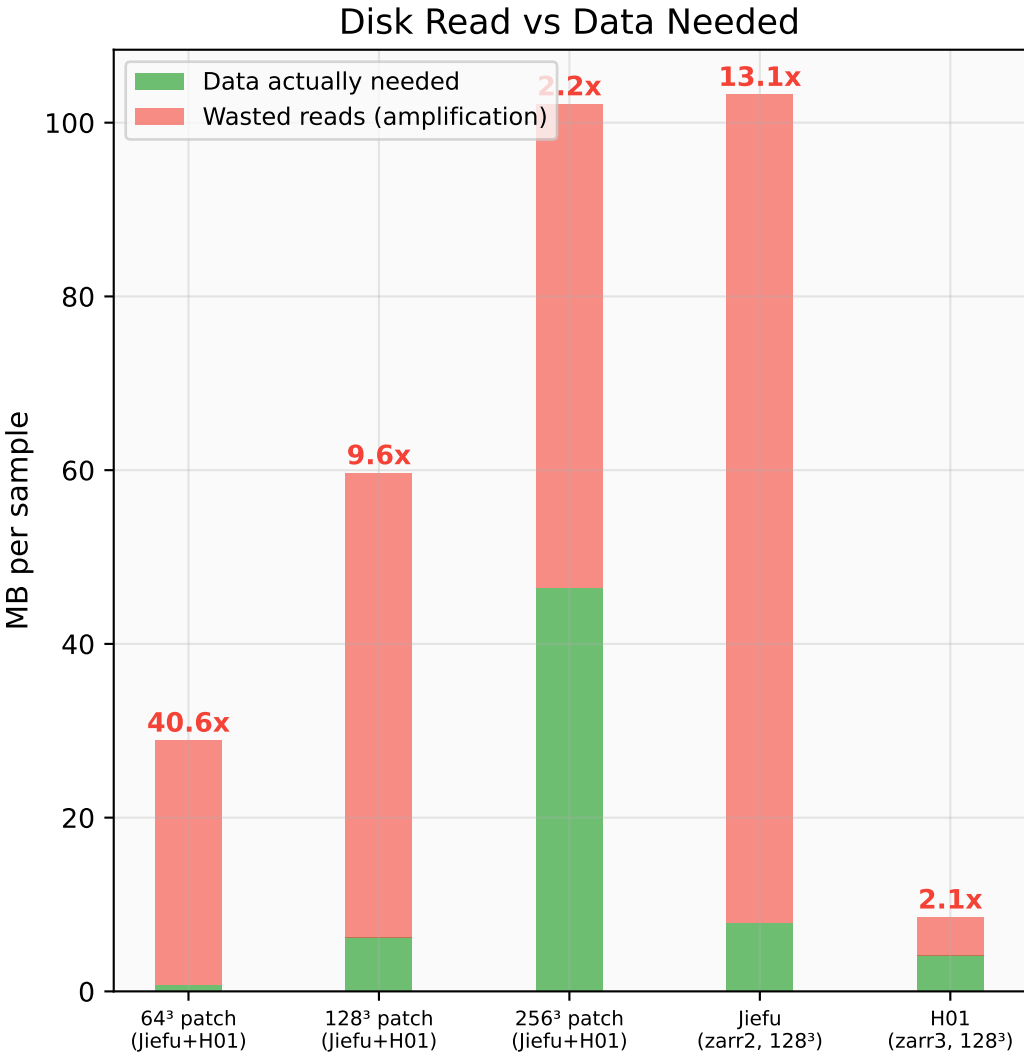
Distribution (top-right): Disk Read & Decompress histograms overlap — warm was slower (106ms vs cold 93ms), well within normal run-to-run variation. No real improvement from having a warm cache.

WHY: Random sampling picks patches from random locations in large volumes. Each new patch reads different chunks. By the time a chunk might be needed again, it has already been evicted (removed) from the cache to make room for newer data.

FIX: Reduce cache size to save memory, or use locality-aware/sequential sampling so nearby patches reuse the same chunks.

Read Amplification: Disk Reads vs Actual Data Needed

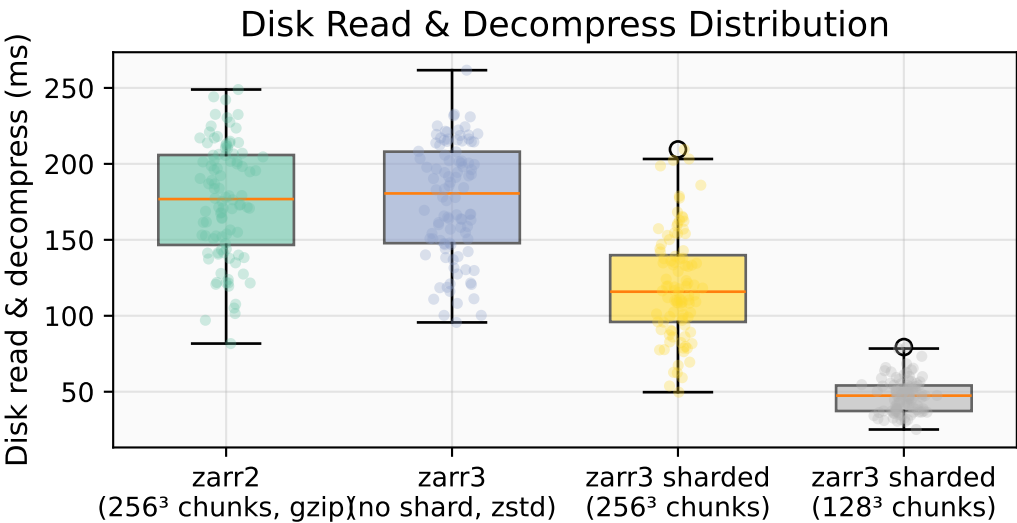
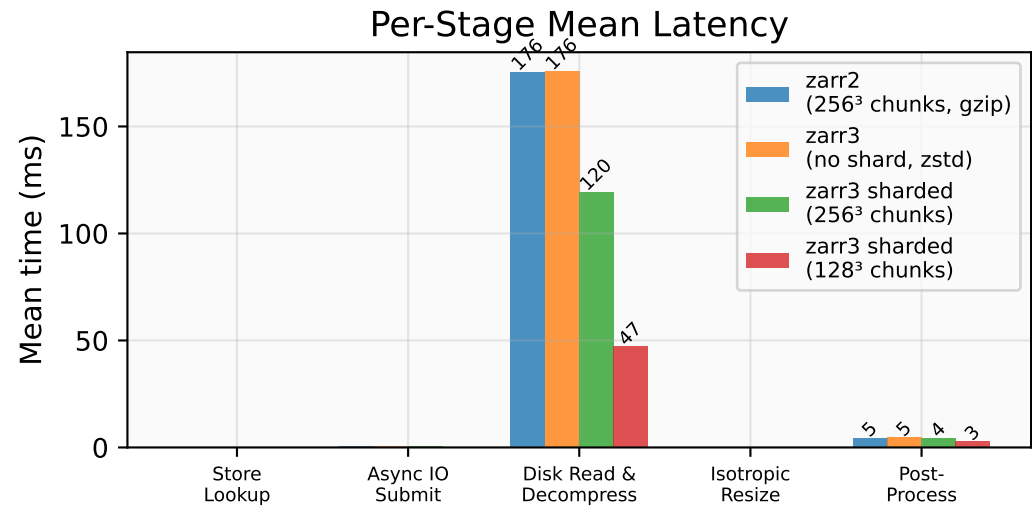
WHAT: Compare bytes read from disk vs bytes actually needed across all configs. Configs: 64³ patch: Jiefu + H01 combined; 128³ patch: Jiefu + H01 combined; 256³ patch: Jiefu + H01 combined; jiefu: Jiefu only (zarr2), patch 128³; h01: H01 only (zarr3 sharded), patch 128³.
WHY: Read amplification quantifies wasted IO — the root cause of slow data loading.



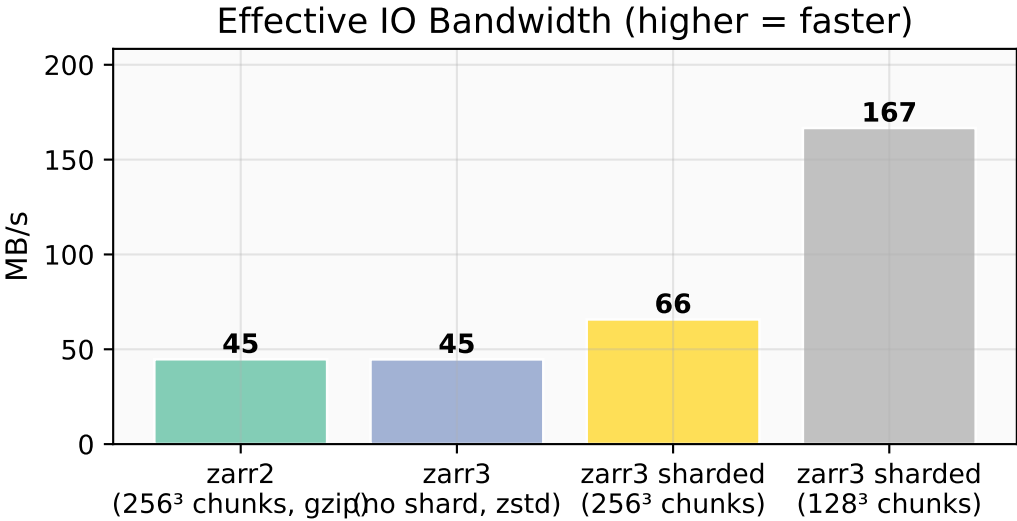
FINDING: Read amplification varies by BOTH storage format AND patch size.
Patch sizes (64³, 128³, 256³) use both Jiefu + H01 combined. 'Jiefu' and 'h01' are single-dataset runs with 128³ patches.
Disk Read vs Data Needed (left): Green = useful data, red = wasted reads. Jiefu (zarr2) has the tallest red bar — most waste. H01 (zarr3) has almost no red.
Read Amplification (right): Values >1x mean wasted reads. Two factors: (1) format — zarr2 stores each chunk as a separate file, causing more waste than zarr3 sharded; (2) patch size — small patches cross more chunk boundaries.
FIX: Convert to zarr3 sharded. Use larger patches when possible.

Comparison: zarr2 (256³ chunks, gzip) vs zarr3 (no shard, zstd) vs zarr3 sharded (256³ chunks) vs zarr3 sharded (128³ chunks)

WHAT: Compare 4 storage formats on the SAME Jiefu Cerebellum dataset (full 536 GB). Same patch [128,128,32], 3 scales, isotropic=true, 100 samples each. Only variable: on-disk format (zarr2 vs zarr3, sharded vs unsharded, chunk size).
WHY: Quantify exactly how much faster zarr3 sharded is vs zarr2 on a real production dataset.



Summary Statistics				
Metric	zarr2 (256 ³ chunks, gzip)	zarr3 (no shard, zstd)	zarr3 sharded (256 ³ chunks)	zarr3 sharded (128 ³ chunks)
Store Lookup	0.2 ms	0.2 ms	0.2 ms	0.2 ms
Async IO Submit	0.5 ms	0.4 ms	0.5 ms	0.4 ms
Disk Read & Decompress	175.7 ms	175.9 ms	119.6 ms	47.2 ms
Isotropic Resize	0.0 ms	0.0 ms	0.0 ms	0.0 ms
Post- Process	4.5 ms	4.7 ms	4.3 ms	2.8 ms
TOTAL	182.4 ms	183.3 ms	125.9 ms	51.8 ms
Bytes/sample	7.9 MB	7.9 MB	7.9 MB	7.9 MB
Isotropic	True	True	True	True
Patch size	[128, 128, 32]	[128, 128, 32]	[128, 128, 32]	[128, 128, 32]

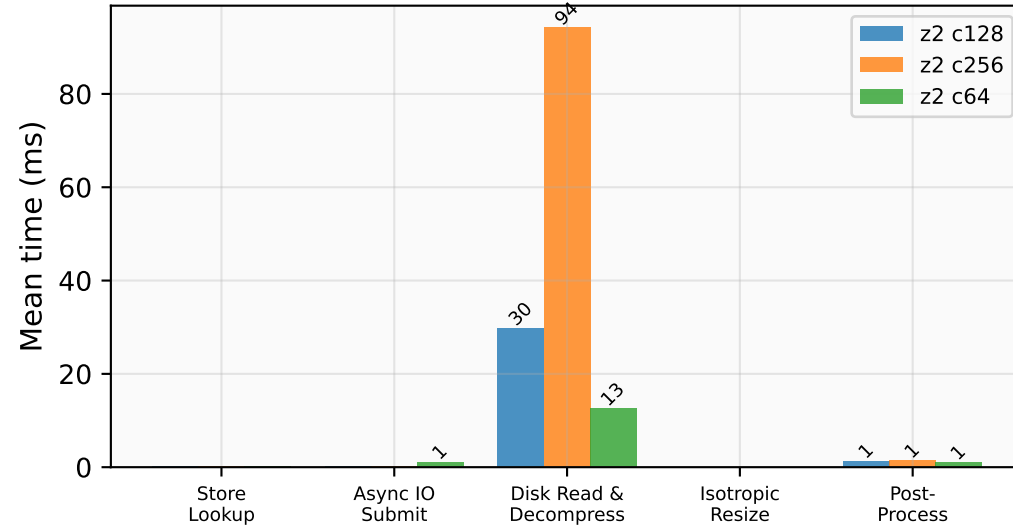


FINDING: zarr3 sharded is 3.7x faster than zarr3.
Disk Read & Decompress: zarr2: 176ms, zarr3: 176ms, zarr3 sharded: 120ms, zarr3 sharded: 47ms.
Read Amplification: zarr2: 13.1x, zarr3: 13.3x, zarr3 sharded: 10.9x, zarr3 sharded: 3.3x.
IO Bandwidth: zarr2: 45 MB/s, zarr3: 45 MB/s, zarr3 sharded: 66 MB/s, zarr3 sharded: 167 MB/s.
FIX: Convert training data to the fastest format.

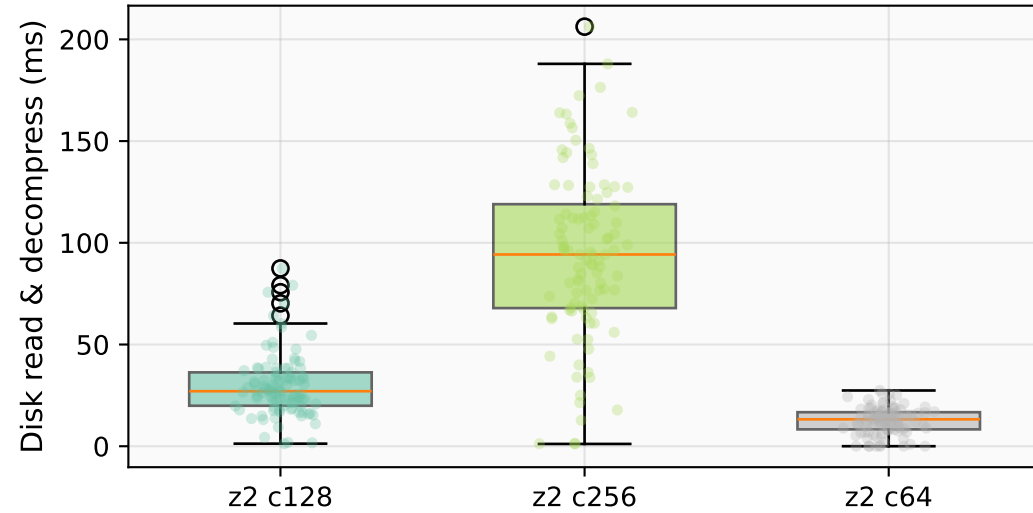
Comparison: z2 c128 vs z2 c256 vs z2 c64

WHAT: Compare zarr2 chunk sizes (64^3 , 128^3 , 256^3) on a 2048^3 Jiefu bbox crop. Same patch $[128, 128, 32]$, 100 samples. Only variable: chunk size.
WHY: Does chunk-patch alignment matter? 128^3 chunks = 1 chunk per 128^3 patch read.

Per-Stage Mean Latency



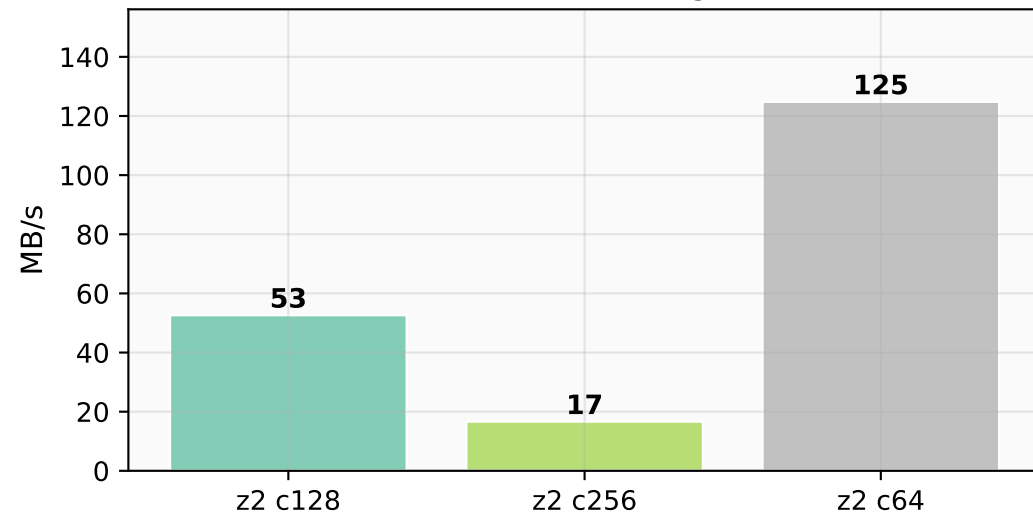
Disk Read & Decompress Distribution



Summary Statistics

Metric	z2 c128	z2 c256	z2 c64
Store Lookup	0.2 ms	0.2 ms	0.2 ms
Async IO Submit	0.3 ms	0.3 ms	1.1 ms
Disk Read & Decompre	29.8 ms	94.2 ms	12.6 ms
Isotropic Resize	0.0 ms	0.0 ms	0.0 ms
Post- Process	1.4 ms	1.4 ms	1.0 ms
TOTAL	32.6 ms	97.0 ms	16.4 ms
Bytes/sample	1.6 MB	1.6 MB	1.6 MB
Isotropic	True	True	True
Patch size	[128, 128, 32]	[128, 128, 32]	[128, 128, 32]

Effective IO Bandwidth (higher = faster)

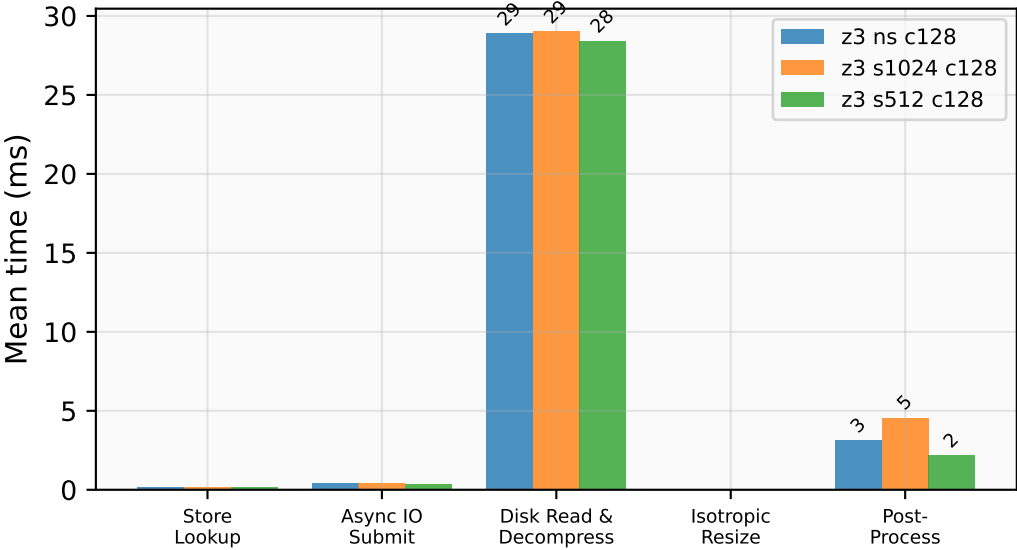


FINDING (zarr2 chunk sweep): z2 c128: 30ms, z2 c256: 94ms, z2 c64: 13ms.
Read amplification: z2 c128: 6.6x, z2 c256: 26.9x, z2 c64: 2.9x.
 128^3 chunks should align with $128 \times 128 \times 32$ patches on XY, reducing wasted reads.
 256^3 chunks force reading 8x more data per patch.

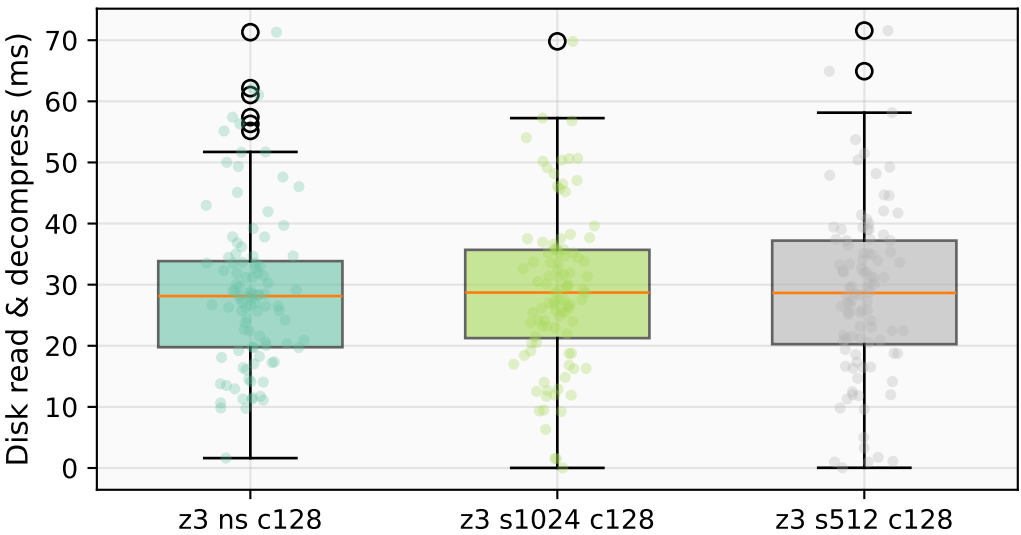
Comparison: z3 ns c128 vs z3 s1024 c128 vs z3 s512 c128

WHAT: Compare zarr3 sharding options at fixed 128³ chunks: no-shard, shard 512³, shard 1024³. 2048³ bbox crop, same patch, 100 samples.
WHY: Does sharding help or hurt for random-access reads? Does shard size matter?

Per-Stage Mean Latency



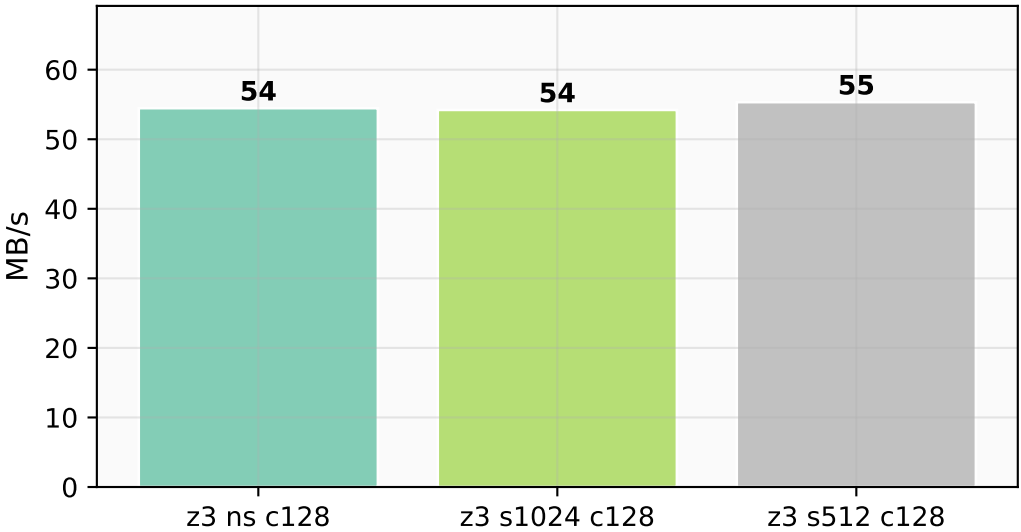
Disk Read & Decompress Distribution



Summary Statistics

Metric	z3 ns c128	z3 s1024 c128	z3 s512 c128
Store Lookup	0.2 ms	0.2 ms	0.2 ms
Async IO Submit	0.4 ms	0.4 ms	0.4 ms
Disk Read & Decompre	28.9 ms	29.0 ms	28.4 ms
Isotropic Resize	0.0 ms	0.0 ms	0.0 ms
Post- Process	3.1 ms	4.6 ms	2.2 ms
TOTAL	33.5 ms	35.6 ms	32.5 ms
Bytes/sample	1.6 MB	1.6 MB	1.6 MB
Isotropic	True	True	True
Patch size	[128, 128, 32]	[128, 128, 32]	[128, 128, 32]

Effective IO Bandwidth (higher = faster)

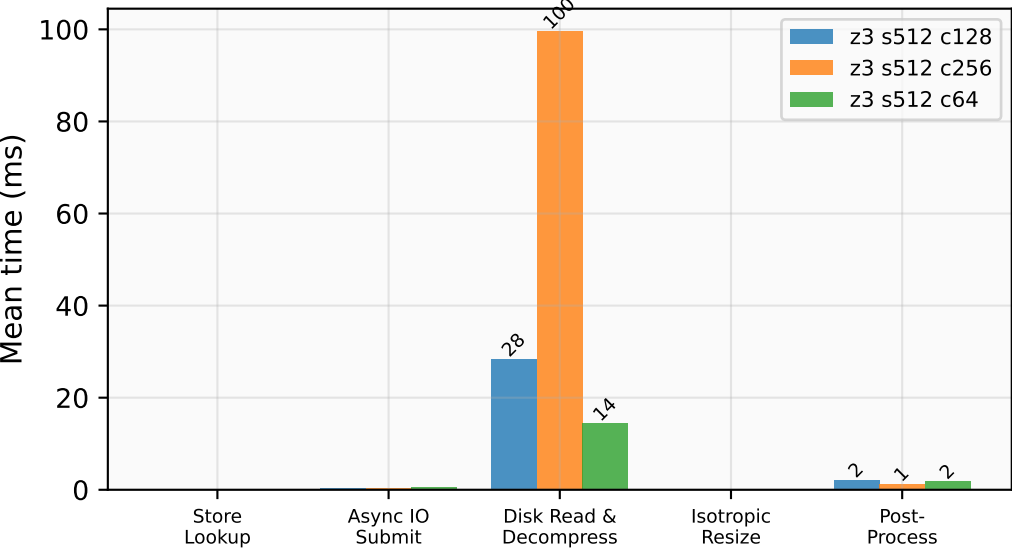


FINDING (zarr3 chunk128, shard sweep): z3 ns c128: 29ms, z3 s1024 c128: 29ms, z3 s512 c128: 28ms.
Sharding packs multiple chunks into one file — fewer file opens, but partial-shard reads may waste bandwidth. Larger shards amplify this tradeoff.

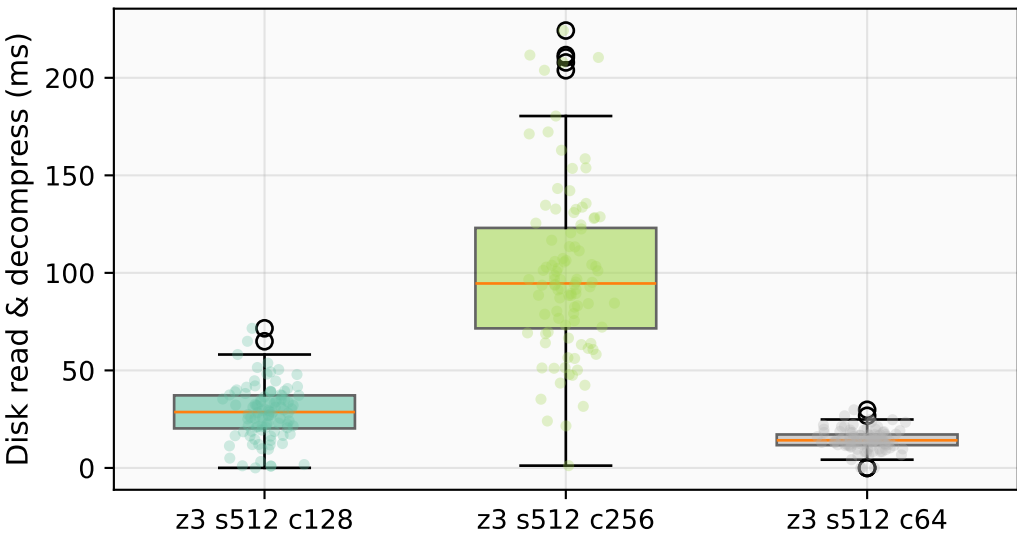
Comparison: z3 s512 c128 vs z3 s512 c256 vs z3 s512 c64

WHAT: Compare chunk sizes (64³, 128³, 256³) within zarr3 shard512. 2048³ bbox crop, same patch, 100 samples.
WHY: Isolate chunk size effect within a fixed shard size.

Per-Stage Mean Latency



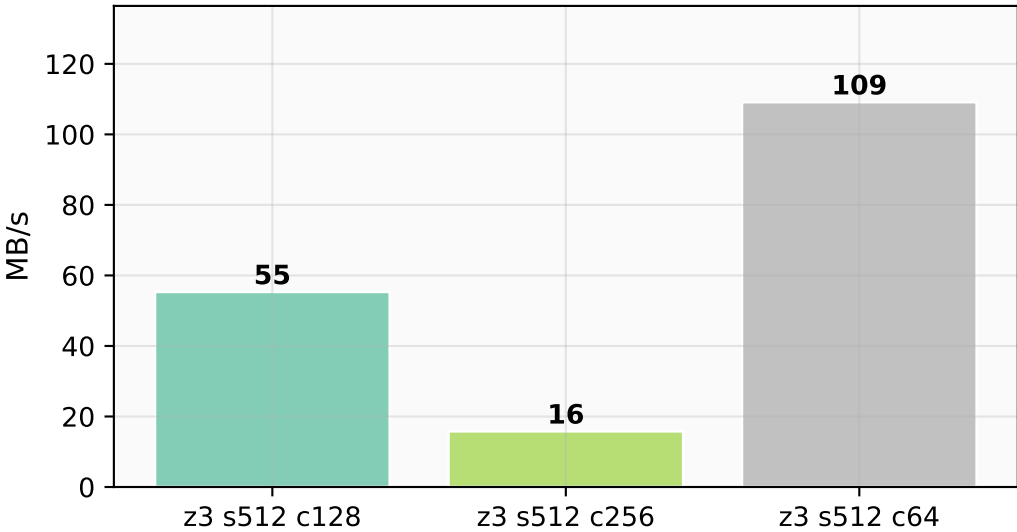
Disk Read & Decompress Distribution



Summary Statistics

Metric	z3 s512 c128	z3 s512 c256	z3 s512 c64
Store Lookup	0.2 ms	0.2 ms	0.2 ms
Async IO Submit	0.4 ms	0.3 ms	0.6 ms
Disk Read & Decompre	28.4 ms	99.5 ms	14.4 ms
Isotropic Resize	0.0 ms	0.0 ms	0.0 ms
Post- Process	2.2 ms	1.3 ms	1.8 ms
TOTAL	32.5 ms	102.2 ms	18.5 ms
Bytes/sample	1.6 MB	1.6 MB	1.6 MB
Isotropic	True	True	True
Patch size	[128, 128, 32]	[128, 128, 32]	[128, 128, 32]

Effective IO Bandwidth (higher = faster)



FINDING (zarr3 shard512, chunk sweep): z3 s512 c128: 28ms, z3 s512 c256: 100ms, z3 s512 c64: 14ms.
Within the same shard size, smaller chunks mean more chunks fit in a shard, potentially improving cache reuse but adding per-chunk overhead.