THE CAPACITY ERA

Storage Days Tallinn

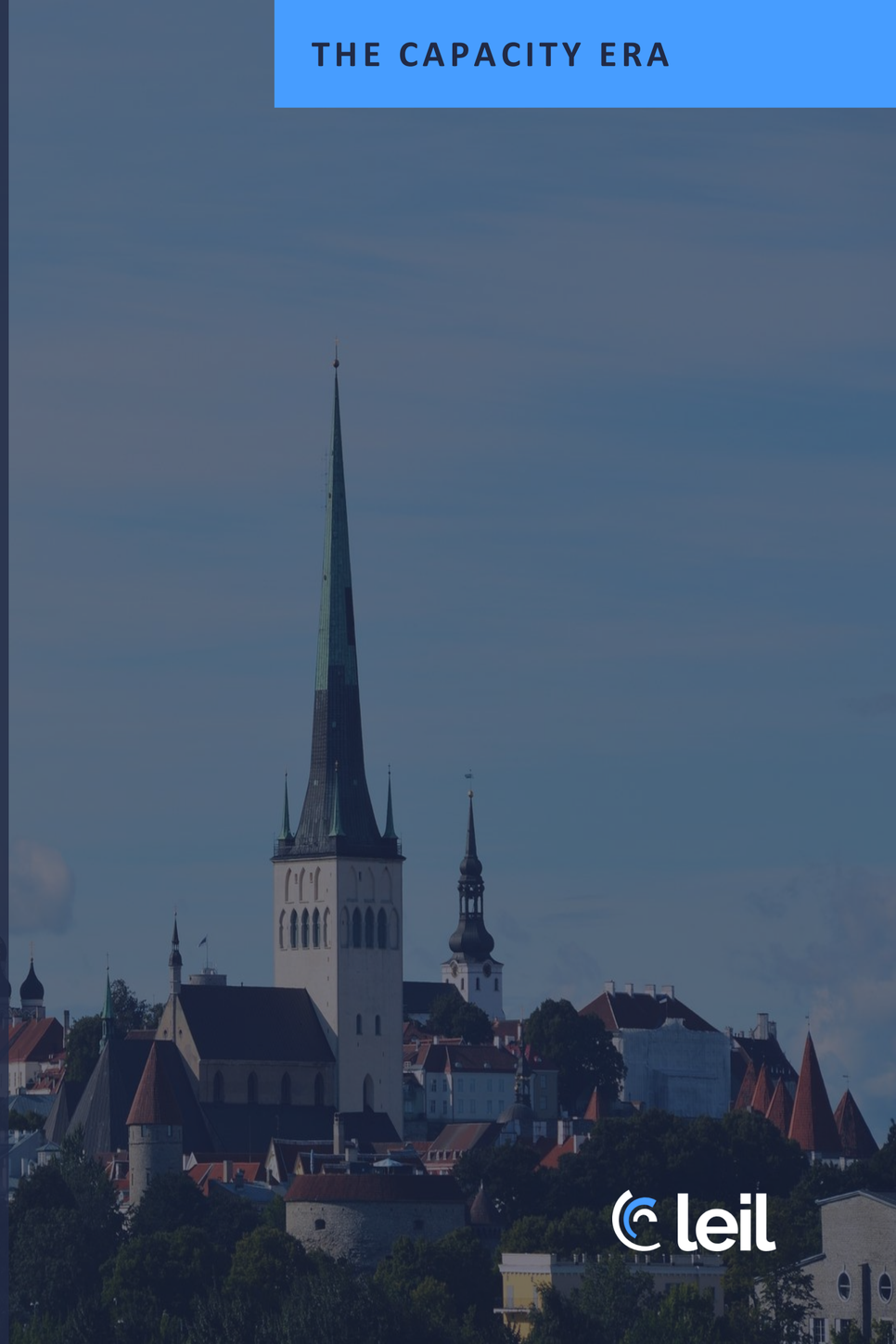
31 August – 1 September 2026

TALLINN, ESTONIA

Capacity efficiency at scale

How native SMR support and on-the-fly compression stack to cut the cost per TB

David Gerstein CTO · Leil



leil

One layer above the local filesystem

This talk sits directly on top of the kernel and filesystem foundation: the device, the zoned block layer and XFS from the session before it. My focus is the workload layer above — what it takes to keep a general-purpose POSIX workload sequential at the drive when the layers above don't naturally cooperate.

- The capacity end is (almost) solved — for append-only archive and backup, keeping it sequential is largely enough.
- The general-purpose end is not — scale-out chunk handling can turn even an append-only workload into random I/O at the drive.
- Two levers move the cost per TB — native SMR support, then compression stacked on top of it.

Agenda

01 Why generic SMR support isn't enough — the metadata problem

02 Native SMR support in LeilFS — split, fragment, reclaim

03 Boosting writes zone assignment — compression rides the write path

04 Compression on the fly — pluggable LZ4 or Zstd, per chunk

05 The economics — stacking density and ratio to cut the cost per TB

SECTION 01

Sequential isn't enough

Why a general-purpose workload needs native SMR support, not a filesystem hiding SMR underneath.

LeilFS in one slide

LeilFS is HDD-Native™ distributed filesystem — the concepts of the Google File System, hardened for on-premise capacity storage. Everything that follows happens inside one component: the chunkserver.

- Metadata servers + data servers (chunkservers) + POSIX clients
- Files are split into chunks of up to 64 MiB
- Each chunk is a sequence of 64 KiB blocks, each with a 4-byte CRC
- Erasure coding across chunkservers — e.g. EC 6+2 for capacity workloads

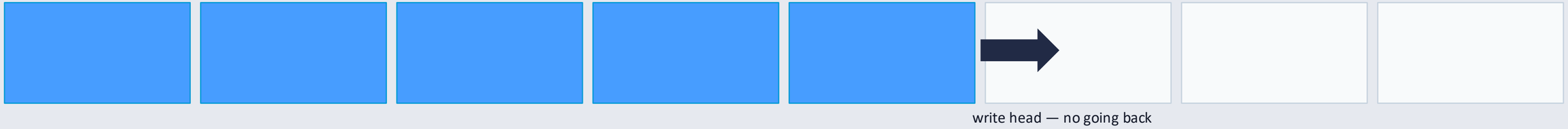
What an SMR drive demands

How a host-managed SMR drive is written

CONVENTIONAL ZONES · random writes allowed

Small pool — good for metadata, not for capacity

SEQUENTIAL ZONES · append-only at the write head, ~256 MiB each, 4 KiB-aligned I/O

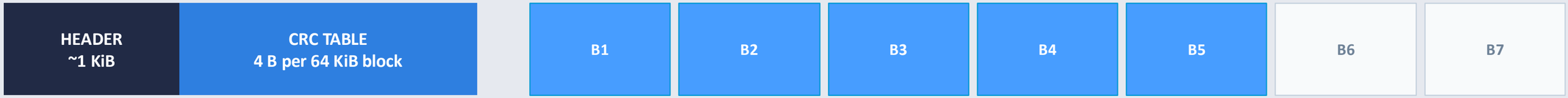


Write out of order and the drive rejects it. Every layer above must guarantee sequential arrival.

Append-only data, random-access metadata

A sequential zone can only be appended at its write head. But a conventional chunk stores its metadata interleaved with its data:

A conventional chunk keeps metadata next to data



Header + CRC = e.g. ~5 KiB — NOT aligned to the drive's 4 KiB I/O block

DATA — 64 KiB blocks, appended sequentially

The catch: every 64 KiB data append forces a 4-byte CRC write back into the metadata band — and the header changes too. Append-only data, but the metadata is rewritten in place, over and over.

Why hiding SMR under the local FS doesn't fix it

Point a stock filesystem at an SMR drive and it sees the chunkserver's in-place metadata churn as exactly what it is: random writes.

XFS SUPPORTING SMR UNDERNEATH

The drive is hidden, the pattern isn't

- The filesystem can't see chunk, block or CRC semantics — only opaque in-place writes
- Constant metadata updates + overwrites become non-sequential writes to the zone
- At the zoned layer this surfaces as write amplification and reclaim, not clean appends

NATIVE SMR SUPPORT IN LeilOS

The layer that knows the workload wins

- Separate the random-access metadata from the append-only data
- Fragment the chunk so out-of-order writes still land sequentially
- Garbage-collect and reclaim zones with full knowledge of what's live

Who decides when to reclaim?

Reclaim isn't a local choice. In a distributed system, when and how hard to garbage-collect depends on cluster state no single filesystem instance can see.

XFS RECLAIMS PER NODE

Local trigger, cluster-blind

- Fires on one node's free-zone threshold — blind to cluster free space, a degraded stripe or an ingest burst
- Every node garbage-collects on its own trigger → correlated GC storms and unpredictable tail latency
- GC whose timing you don't control — the DM-SMR failure mode, one layer up

LeilFS RECLAIMS GLOBALLY

The layer that owns the cluster wins

- Times and shapes reclaim with cluster free space, rebuild state and workload phase in view
- Backs off under load and staggers reclaim across nodes — no correlated stalls
- Host-managed SMR exists so the host controls GC; in a cluster, that host is the distributed FS

SECTION 02

Native SMR support

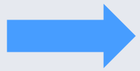
Three mechanisms turn a messy write pattern back into clean, sequential drive I/O.

Mechanism 1 — split metadata from data

The header and CRC move off the shingled tracks entirely, so nothing that changes in place ever touches a sequential zone.

One chunk, split by access pattern

LeilFS chunk



METADATA header + CRC

→ conventional zones / NVMe · random-friendly, updated in place

DATA 64 KiB blocks

→ SMR sequential zones · pure append, nothing rewritten in place

Result: the two problems SMR creates for a conventional chunk — in-place CRC writes and 5 KiB misalignment — simply disappear.

Mechanism 2 — fragment the chunk

When writes arrive out of order, LeilFS appends each one at the zone's write head and records where it landed — so the zone only ever sees sequential I/O.

ONE SEQUENTIAL ZONE — physical order = arrival order

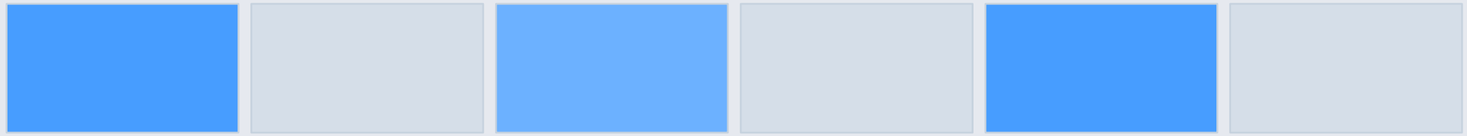
The client wrote blocks 1, then 3, then 2. On the platter they are strictly in order; a small fragment table in the metadata records the logical mapping. A chunk with more than one fragment is simply marked fragmented — to be tidied later.

Fragment record = zone · offset-in-zone · first block · block count (12 bytes). Reads follow it; the drive never sees a seek.

Mechanism 3 — garbage-collect and reclaim

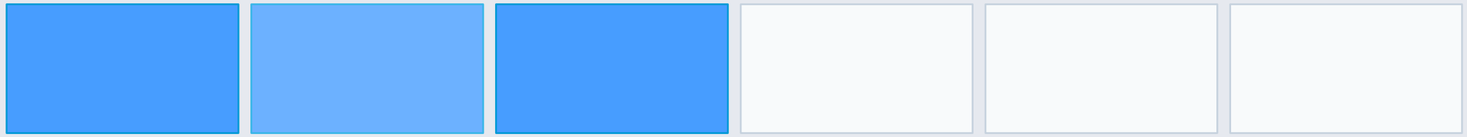
Fragmentation trades clean layout for sequential writes. GC pays it back: compact live data into a fresh zone, then reset the zones left holding nothing live.

BEFORE — dirty zones, holes between live fragments



grey = reclaimable holes

AFTER — live data compacted, empty zone reset and returned



LeilFS always keeps at least one zone empty, so reclaim can always make progress — the invariant that keeps a petabyte-scale drive from wedging.

Reclaiming space in minutes, not hours

Single node, 60 SMR drives, LeilFS 5.11, EC 6+2. Delete 10% of an 88%-full node and reclaim.

128 TB

Freed on one node

Node usage fell 88.46% → 79.62% between — space back in service immediately.

< 6 min

To reclaim

~21.3 TB/min per node when aged data is deleted; refills afterwards at ~9.9 GB/s, near full write speed.

1.1 PB

At 8 nodes, same window

Every node reclaims in parallel, so 480 drives clear ~1.1 PB in the same under-6-minute window.

The one variable that decides reclaim speed

SMR drives can only rewrite a whole zone at a time — so what you delete matters more than how much.

AGED / BATCHED DELETION

Whole zones empty at once

- Retire whole containers or datasets written around the same time
- Entire zones free up with almost no data movement
- ~140 TB reclaimed in under 6 minutes — the comfortable case

RANDOM SCATTERED DELETION

Survivors must be rewritten

- Partly-empty zones force read-and-rewrite of the live data
- ~9 units moved for every 1 reclaimed → hours, not minutes
- Handle by design: delete at the batch / container level

Native, not slow

LeilFS on 102 × WD Data102 SMR (SAS3, EC-ready), measured against the drives' own raw ceiling.

99.7%

Read efficiency

14.2 GB/s sustained vs 14.24 GB/s raw at 102 FIO jobs.

95.4%

Write efficiency

8.28 GB/s sustained vs 8.68 GB/s raw at 102 jobs.

Linear

Scaling

Throughput grows with every drive added — no software ceiling below the hardware.

SECTION 03

Boosting writes

The single biggest gain we measured this year came from software — and it lands exactly where compression needs it.

Zone assignment: one choice, one trade-off

Which zone LeilFS writes next decides whether a chunk's appends stay sequential at the drive. Two strategies, opposite ends of the same trade-off.

LeilOS <=5.10 · “SPREAD”

Optimised for reads

- Each new chunk goes to a different zone
- Fragmentation stays near zero → excellent sequential reads
- Cost: a non-appending write jumps the drive to a new region

LeilOS >=5.11 · “FILL ONE ZONE”

Optimised for writes

- Fill one zone at a time; a client flush hint batches larger writes
- Single-writer ordering keeps writes perfectly sequential
- +57% write throughput for a ~7% read trade-off

+57% on writes, for 7% on reads

Same 8 × 30 TB WD ePMR drives, EC 6+2, sequential sweep.
Only the zone-assignment strategy changed.

1.52 GB/s

Experimental writes

vs 0.965 GB/s on the stable “spread” branch — a +57% gain from software alone.

–7%

Read trade-off

1.52 vs 1.63 GB/s reads — a favourable trade for write-heavy ingest.

0 hw

Hardware changed

Identical drives, HBA and fabric. The filesystem is the lever.

SECTION 04

Compression on the fly

Pluggable LZ4 or Zstd, per chunk, on the write path — so it composes with everything in Section 03.

How LeilOS compresses

Compression is transparent and native to the SMR chunk format — not a layer bolted on top. Four design choices make it safe at scale:

- One frame per 64 KiB block — a random read decompresses only the block it asked for.
- Pluggable per chunkserver — LZ4 for line-rate writes or Zstd for maximum ratio; LZ4 keeps no per-chunk dictionary, so it adds almost no metadata.
- Per-block raw fallback — a block that won't shrink is stored raw, so a compressed chunk is never larger than a legacy one.
- Format fixed at chunk creation — re-derived from the chunk's own metadata, so a config change never reinterprets data already on disk.

LZ4: faster than uncompressed

50% compressible data, LZ4 level 1, EC 6+2, chunkserver tuned. Same rig as the Zstd runs.

13.1 GB/s

LZ4 writes

At 50% compressible data — faster than the 10.1 GB/s uncompressed baseline.

18.0 GB/s

LZ4 reads

vs 10.5 GB/s uncompressed; reads are client- and network-bound, so both algorithms match.

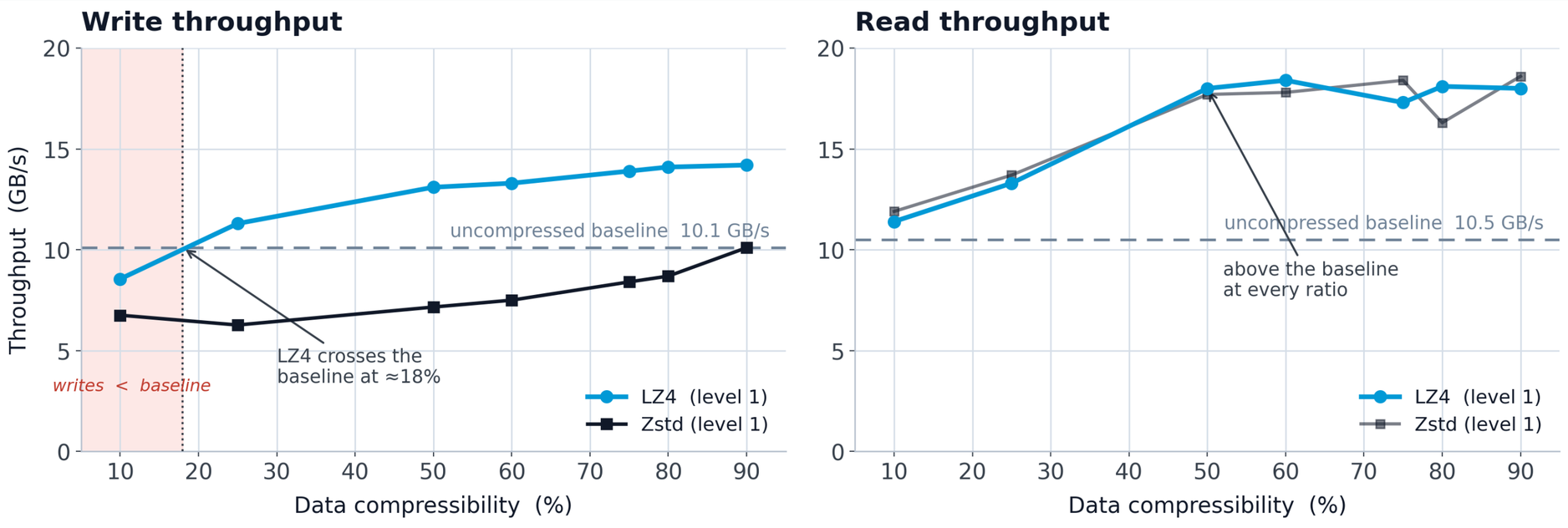
1.8×

vs Zstd on writes

13.1 vs 7.2 GB/s at level 1 — Zstd saturates the CPU, LZ4 leaves headroom.

Throughput vs data compressibility

An FIO sweep varies data compressibility, measuring LZ4 and Zstd against the uncompressed baseline (EC 6+2, 100 jobs, 60 HM-SMRs, 1 MiB direct I/O).

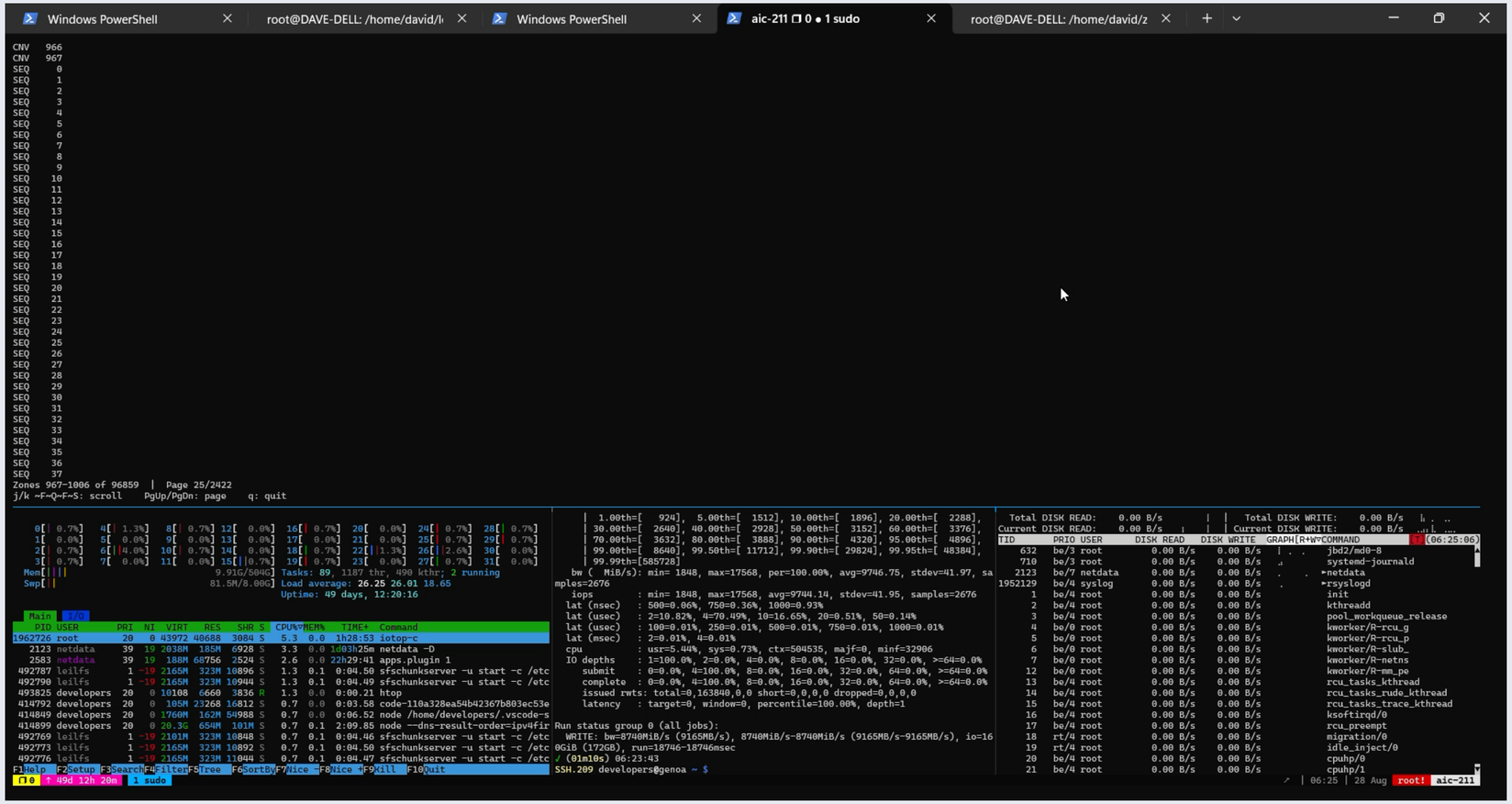


Performance gain: the HDD was the bottleneck

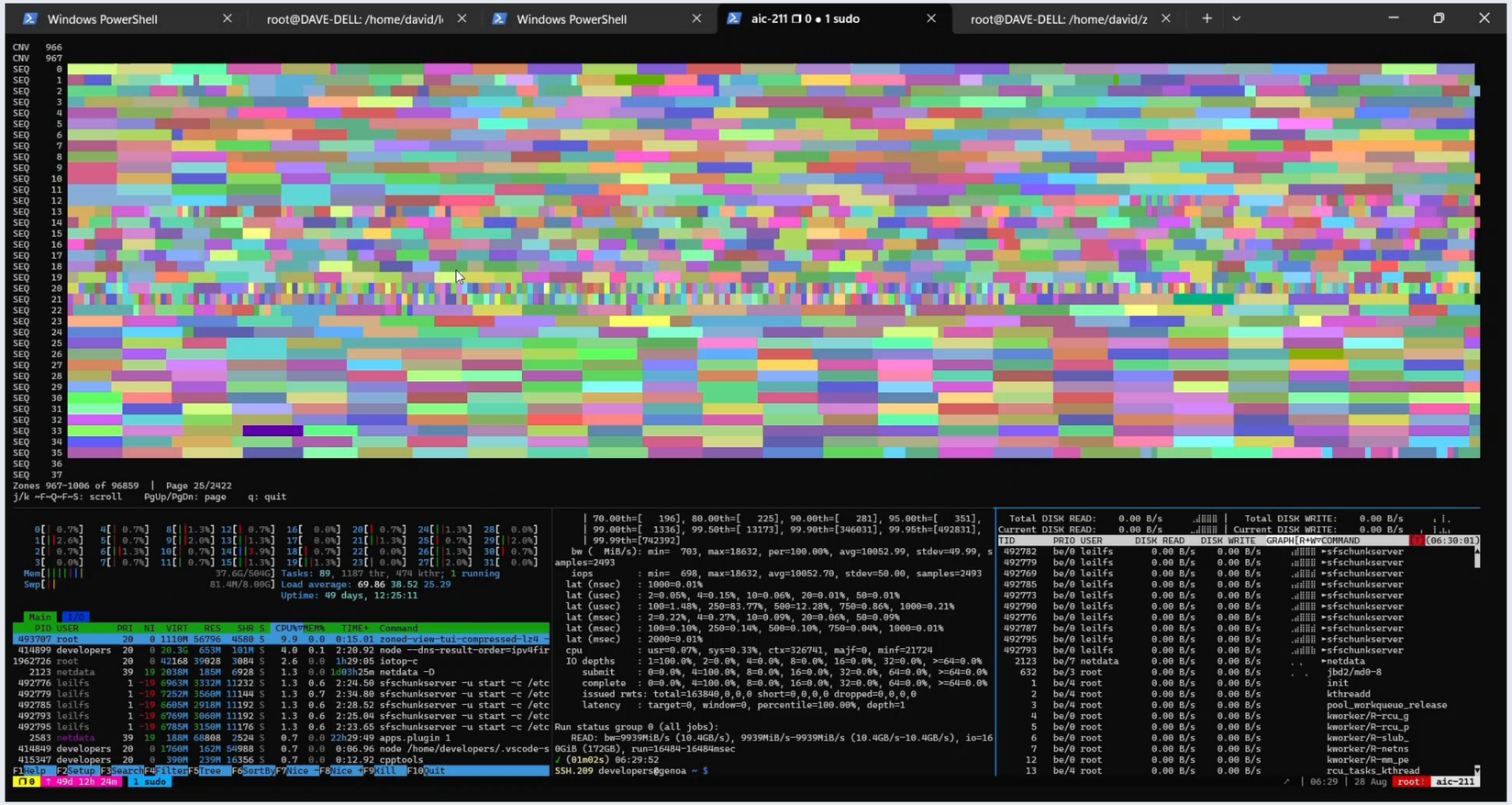
Writes cross the baseline; reads never dip below it — because compression (writes) costs far more CPU than decompression (reads).

- For no compression case the platter sets the ceiling — a chunkserver moves a fixed number of physical bytes per second; compression only changes how many logical bytes that represents.
- Writes trade CPU for bytes — below ≈18% compressibility the byte savings can’t cover LZ4’s overhead, so writes fall under the baseline (10% → 8.6 vs 10.1 GB/s).
- Past the crossover the drive wins — +12% at 25%, +40% at 90%; Zstd stays CPU-bound and below the baseline far longer.
- Reads have no crossover — decompression is cheap and reads bind downstream, so fewer physical bytes read means +9% to +71% at every ratio.

Demo: Chunkserver write throughput



Demo: Chunkserver read throughput



Compression is a performance feature, not a tax

The instinct is that compression must cost you performance. For reads it's the opposite — and we engineered the write path so it holds there too.

- Reads get faster by default — 18.0 vs 10.5 GB/s. Fewer physical bytes cross the platter and the bus, so client throughput can exceed the raw disk rate.
- Writes are where the cost lands — Zstd is CPU-heavy, and on the write path the CPU, not the disk, is the bottleneck.
- Two improvements, both shipping — fill-one-zone ordering (Section 03) plus LZ4's ~1.8× lower CPU cost push compressed writes to 13.1 GB/s, past the 10.1 baseline.
- Next: FPGA offload — hardware headroom for Zstd-level ratios at LZ4-level speed.

Where compression is today — and next

Compression is in the SMR data path now; CMR support comes first, then selective and retroactive control.

TODAY

On-the-fly, global per chunkserver

- Pick the algorithm and level (LZ4 or Zstd); reloadable, applies to new chunks
- Fully integrated with fragmentation, GC and erasure coding

ROADMAP

CMR first, then goal-based on demand

- Extend compression from SMR to CMR chunkservers as the first step
- Per-goal policy — compress what benefits, recompress in place, no migration

SECTION 05

The economics

Two independent efficiency gains that multiply — and why, on migration, compression is not optional.

Two gains that multiply

SMR raises how much a drive holds; compression raises how much of that is your data. They stack.

+20–25%

From SMR density

Shingled areal density plus mixed CMR/SMR and variable capacity (roughly +6 TB usable on a 30 TB-class drive).

+20–50%

From compression

Effective-capacity gain on typical data, e.g. LZ4 stored 8000 GiB of 90%-compressible data in 846 GiB, about 89% saved.

~35–45%

Lower cost / TB

Density × ratio compounds to well over 1.5× effective capacity — a materially lower cost per usable TB.

Migrating from CMR + compression?

The trap: SMR looks denser, so teams assume any move to SMR lowers cost per TB. If you already run compression on CMR, that isn't automatic.

SMR WITHOUT COMPRESSION

Cost per TB can go up

- You gain ~20–25% raw density from shingling
- But you lose the 20–50% you were getting from compression
- Net effective capacity can fall — the migration costs you money

NATIVE SMR + COMPRESSION

Cost per TB drops — as intended

- Keep the compression ratio you already depend on
- With LZ4, keeping compression costs nothing on writes — it now runs faster than not compressing.
- For a CMR + compression shop, native compression is a must-have

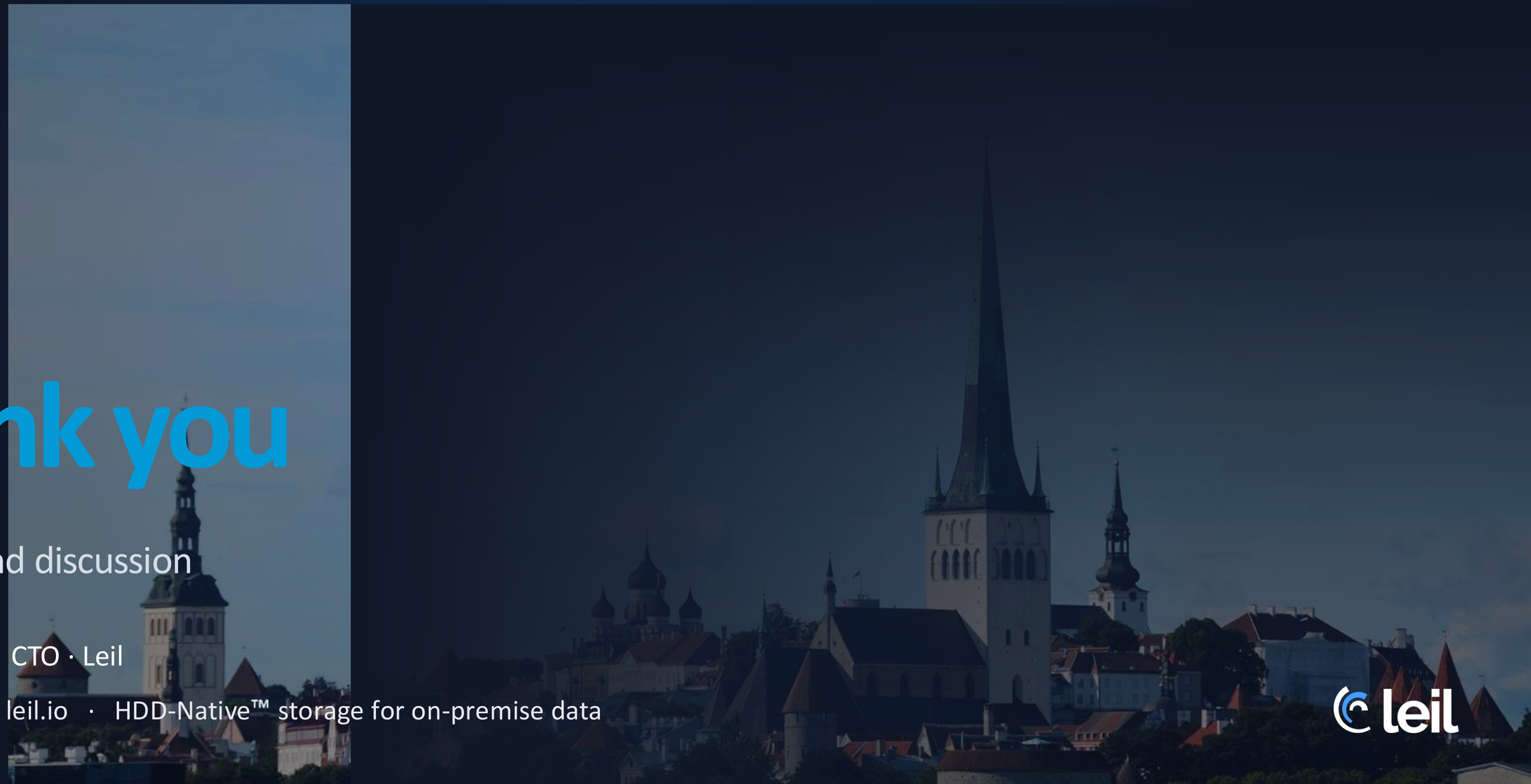
Takeaways

01 Scale-out metadata can turn an append-only workload into random I/O at the drive — so SMR support has to be native to the layer that understands the workload, not hidden beneath a general-purpose filesystem.

02 Once writes are native and sequential, the filesystem becomes the biggest lever: zone assignment alone bought +57% write throughput on identical hardware, with efficiency sitting at 95–99% of the drives' raw ceiling.

03 SMR density and on-the-fly compression are independent gains that multiply — and with LZ4 running faster than not compressing, keeping compression on is essentially free, so cost per TB keeps dropping instead of rising on migration.

Storage Days Tallinn



Thank you

Questions and discussion

David Gerstein CTO · Leil

david@leil.io · leil.io · HDD-Native™ storage for on-premise data

leil