



Lustre File Data Resiliency in 2.18: Erasure Coding With Friends

Marc-André Vef, DDN

Patrick Farrell, The Lustre Collective

Andreas Dilger, The Lustre Collective

Lustre Erasure Coding: It Takes a Village

- Started development over 8 years ago:
- 2017 – Intel – Zhenyu (Bobijam) Xu ([LAD2018](#))
- 2021 – ORNL – Adam Disney ([LUG2021](#))
- 2024 – ORNL ([LAD2024](#))
 - James Simmons, Chris Brumgard
- 2025 – DDN/Whamcloud/TLC ([LUG2026](#))
 - Marc Vef, Ronnie Sahlberg, Max Dilger, Keguang Xu, Zhenyu (Bobijam) Xu
 - Patrick Farrell, Andreas Dilger, Hiroshi Nishida
- Many thanks to everyone involved in bringing this important feature into the 2.18 release!



Lustre: Increased Scale Meets Hardware Limitations

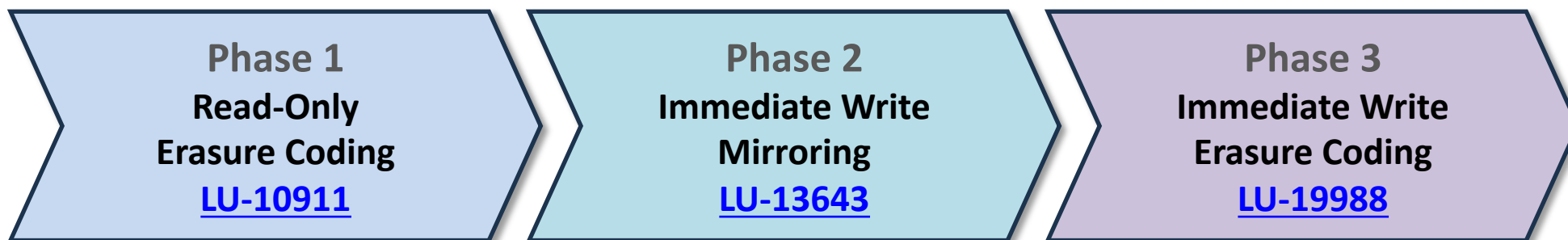
- Extreme scale means hardware failures are routine
 - A single filesystem may have 10K storage devices, 1K servers, 4K network cables/ports
 - Most nodes and storage are on OSTs, so they fail most frequently
- Mitigating in hardware keeps getting more expensive
 - OSTs have become larger over the years
(100 TiB was originally a whole Lustre filesystem, now 100-1000 TiB per OST)
 - Added multiple network interfaces for better connectivity
 - Flash devices more reliable than spindles
- Device scaling/resilience only gets you so far, and what about flaky networks?

Lustre Data Redundancy: Today

- Tolerate OST failure in software – break assumption that hardware is always available
 - Lose up to m OST objects and still be able to access file data
 - Per-file layouts: overhead and resilience configurable per file/directory
- File Level Redundancy (FLR): read-only, delayed mirroring since 2.11
 - Clients write only to a *primary* mirror; all others are made **STALE**
 - `lfs mirror resync` syncs data to secondary mirrors
 - Once a mirror is in sync, it can be read from any mirror
- Mirroring (RAID-1): $m+1$ full copies of the data, tolerate m disk failures
- Erasure coding (RAID-4/5/6): m redundancy (parity) chunks for m failures; no full copies

Lustre Resiliency Project

Staged plan to improve data resiliency



- Delayed/Async parity resync
- Immediate benefits for users
- Write-time fault tolerance
- Mirror space usage reduced after delayed EC parity resync
- Write-time, space-efficient fault tolerance

Each stage is useful independently, further increases data protection

Read-only Erasure Coding

[LU-10911](#)

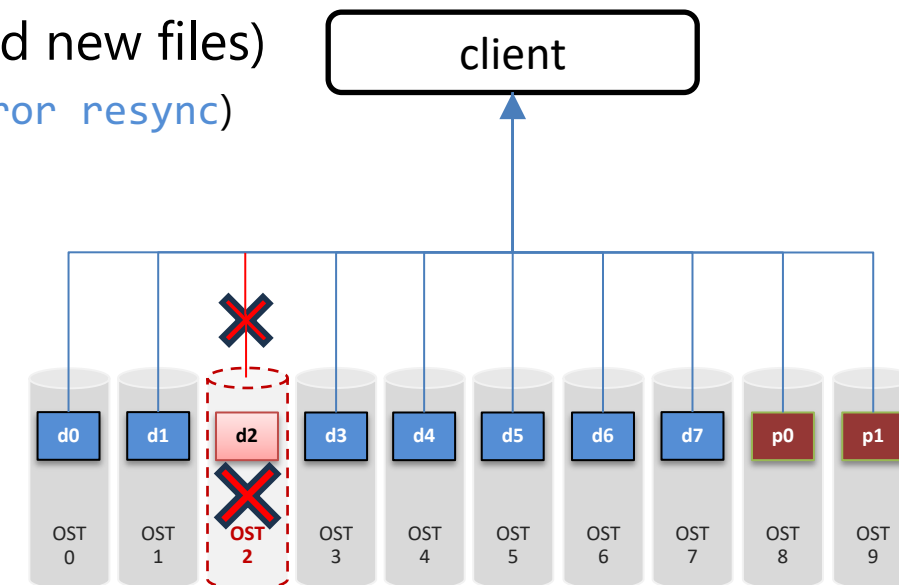
Phase 1: Read-only Erasure Coding

- Write striped file data normally; write redundant parity stripes later
- Once parity in place, reads bypass most OST and network failures
 - Missing data is transparently rebuilt by the client on the fly (**degraded read**)
 - Agnostic of storage, server, or network failures
- Resync works like read-only mirroring (for both old and new files)
 - Add parity after file is written instead of a full copy (via `lfs mirror resync`)
- Reduces impact of OST failover
 - Reads continue within seconds of failure
- Far more space-efficient and robust than mirroring

8+2 EC: handle any 2 device failures, 25% bandwidth/space overhead

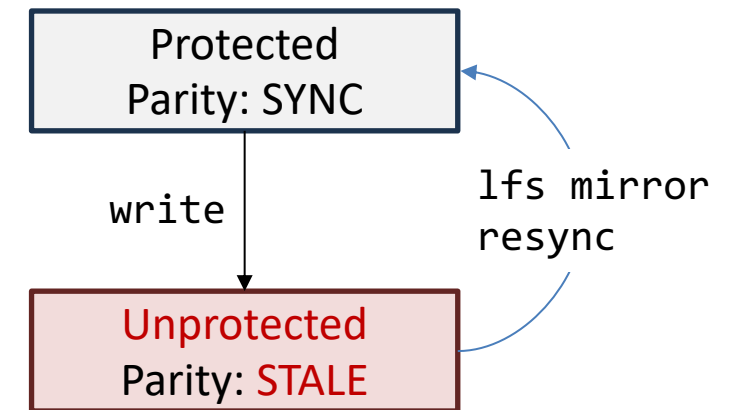
2-way mirror: handle 1 device failure, 100% write/space overhead

3-way mirror: handle 2 device failures, 200% write/space overhead



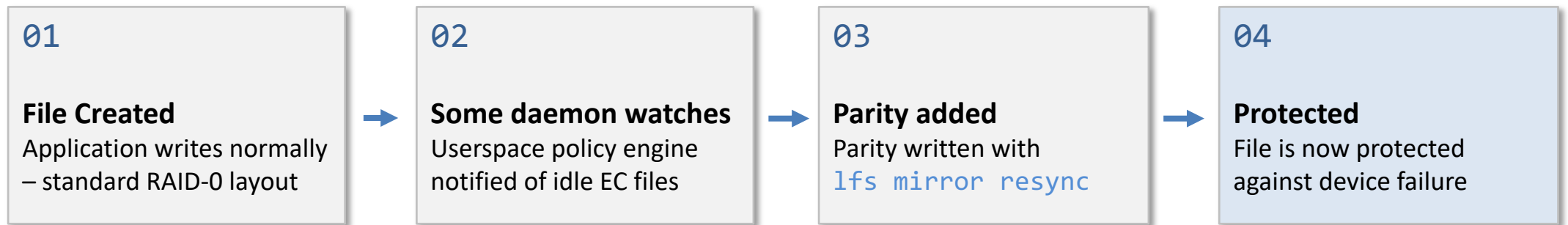
EC: Writes and Parity Lifecycle

- Writes go **to the primary data mirror** (RAID-0 striped I/O)
 - No bandwidth overhead during initial write
 - Parity mirror is automatically marked STALE on write (delayed)
- After write, parity is out of date until resync
 - lfs mirror resync computes parity from data using ISA-L
 - Uses [SEEK_DATA/SEEK_HOLE](#) to only resync regions with actual data
- ChangeLog records stale transitions
 - Policy-engines can trigger resync without scanning
- [lfs mirror verify](#) validates stored parity matches computed parity



Read-only EC: how it flows

- Idle files with EC layout get protection eventually (within ~minutes)
- Not ideal for every application, but has real world uses and benefits
- Protects data for the 99.9% of its lifetime after write has completed



Use Case: Generative AI Training Data

- Write once, read forever
 - Written once during dataset preparation
 - Read thousands of times during training across many nodes
 - Datasets only grow; existing data is never modified
- A read failure during a training run can force a costly restart
 - GPU hours are expensive and continuous usage is key
 - Multi-day runs interrupted by storage waste real money, even with checkpoints
- Read-only EC protects this data with minimal storage overhead
 - 8+2: any 2 device failures, only 25 % extra storage
 - Mirroring: 100 % storage overhead for single-failure protection

Space Efficiency: EC vs Mirroring

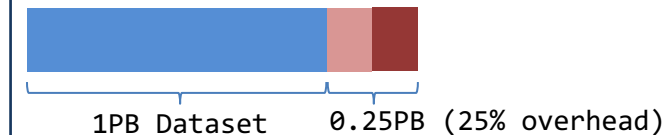
Scheme	Overhead	Failures
1 mirror	100%	1
2 mirrors	200%	2
4+1 EC	25%	1
4+2 EC	50%	2
8+2 EC	25%	2
16+2 EC	12%	2
16+3 EC	18%	3
24+3 EC	12%	3
32+3 EC	10%	3
32+4 EC	12%	4

Scenario: 1 PB Dataset

Mirroring – 2 failure protection



8+2 EC – 2 failure protection



8+2 EC provides 1.75PB savings
At the same protection level

Read-Only EC: Limitations

- No coverage for newly written data
 - Parity exists only after **explicit** resync
 - Need to opt-in pre-existing files to add EC mirror component
 - Single-stripe data components are still mirrored (at start of file, usually 5-10% of total capacity)
- No resilience for files which are continually modified (AI training checkpoints, DBs, ...)
 - Only safe after the file is idle and resynced
 - Cannot survive OST loss during write

Read-Only EC: Some Technical Details

- All parity stored in separate FLR parity objects (RAID-4, **not** RAID-5/6)
 - Parity naturally distributed over OSTs by different files to avoid reconstruction hot spots
- Resync tool computes parity from data using Intel ISA-L library
 - ISA-L base implementation (x86, ARM, RISC-V)
 - 60–80 GB/s per core for 5+1 with vectorized EC kernels (AVX2/GFNI, in review)
- On-disk format changes minimally affect existing file layouts:
 - New lcme_flags: `LCME_FL_PARITY`
 - `lcme_dstripe_count` (k , data stripes), `lcme_cstripe_count` (m , parity stripes)
 - `lcme_mirror_link_id`: bidirectional link between data and parity mirror
- Standard limits $k \leq 32$, $m \leq 4$; expert mode (`--ec-expert`) $k \leq 255$, $m \leq 15$

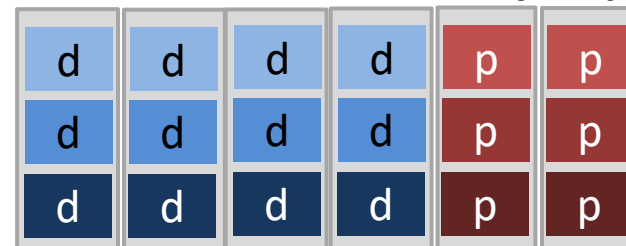
EC: Why RAID-4?

- Data and parity stripes live in separate OST objects
 - Data uses the existing RAID-0 layout
 - Parities are added without rewriting any data
- Parity is an FLR mirror
 - stale handling, resync/verify are reused
- No new layout magic (reuses [LOV_MAGIC_COMP_V1](#))
- Parity component adds new layout flag for interoperation
 - Flag: [LOV_PATTERN_PARITY](#)
 - Older clients skip the unknown PARITY flag and read/write only the data mirror

```
# enable 8+2 EC on new 8-stripe file at time of creation
$ lfs setstripe -E -1 -c 8 --ec 8+2 /lustre/newfile
# enable 8+2 EC on existing 8-stripe file after creation
$ lfs mirror extend --ec 8+2 /lustre/oldfile
```

RAID-4

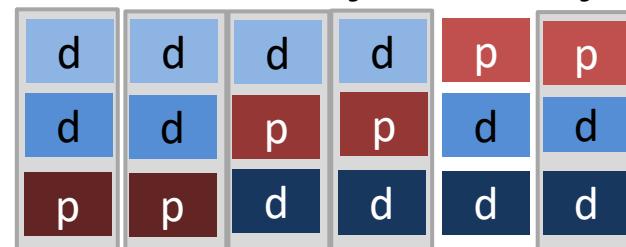
Dedicated Data and Parity Objects



Adding parity does not affect data

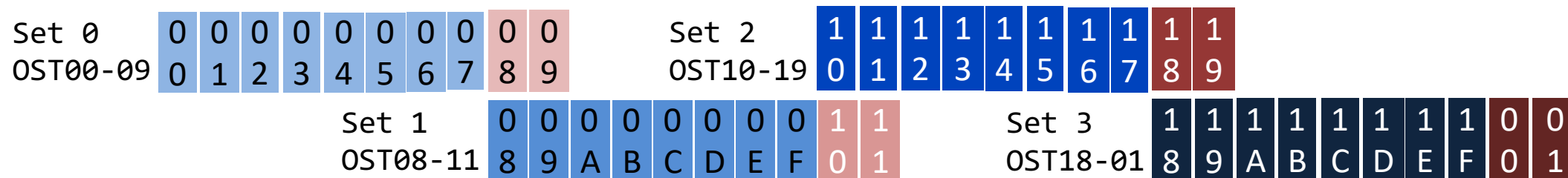
RAID-5/6

Interleaved Parity Across Objects



Adding parity needs data re-write

EC: RAID Sets



FILE: 32 data objects - 8+2 EC - 4 EC RAID sets on 32 OSTs – 8 parity objects overlapping 8 data objects

- Widely striped files are divided into RAID sets, each an independent redundancy group
 - Without RAID sets, rebuilding one stripe would read all stripes
- RAID sets solve the "every OST already holds a stripe" problem
 - 32 data stripes on 32 OSTs, 8+2: 4 RAID sets, 8 parity objects on OSTs that hold *another* set's data
 - *Parity objects* can reuse OSTs of *data objects* in another RAID set without loss of redundancy
- RAID-set-aware allocation: data and parity of one RAID set never share an OST
- Sparse files: a RAID set entirely inside a hole stores no parity

EC: Degraded Read Recovery

- Normal reads use only the primary data mirror (no parity needed)
- When an OST is unavailable, the client switches the read to the `CIT_EC_RD` IO type
 - Outer I/O: original VFS read request (application offset+count)
 - Inner I/O: expanded to whole RAID-set rows
 - Surviving data and parities are read, and the missing data is reconstructed
- Reconstructed data pages are cached until the OST is recovered
 - Those pages are discarded after recovery*
- A stale parity mirror cannot be used for recovery and must be resynced first
- Repair: `lfs migrate` rewrites the file through the degraded read

EC: Failure Domains

- Parity on the same failure domain as its data has limited protection
- Admins can set numeric labels per OST ([LU-19066](#) - landed)
- Same number = same domain; no further meaning
- Used by the stripe allocator ([LU-19823](#)) for EC and non-EC files (with mult. FLR mirrors)
 - FLR mirrors of the same extent never share a domain
 - For EC: within a RAID set, data and parity stripes are placed on distinct domains
 - Compatible with `-C` (overstriping)
 - Currently in review

```
# 0 = unset  
oss $ lctl set_param obdfilter.fs-OST0001.failure_domain=4  
oss $ lctl set_param obdfilter.fs-OST0002.failure_domain=4
```

Read-only Erasure Coding: Status & Usage

- Current status on master 2.17.58-62: EC is currently guard-protected (removed by 2.18)
- Usage (see [lustre/tests/sanity-ec.sh](#) for further examples):

```
# Create file: 8 data + 2 parity stripes  
client $ lfs setstripe --ec 8+2 /lustre/file1
```

Read-only Erasure Coding: Status & Usage

- Current status on master 2.17.58-62: EC is currently guard-protected (removed by 2.18)
- Usage (see [lustre/tests/sanity-ec.sh](#) for further examples):

```
# Create file: 8 data + 2 parity stripes
client $ lfs setstripe --ec 8+2 /lustre/file1
# Create file: One --ec for all components: [0,1M) 1 stripe -> 1+2, [1M,EOF) -> 8+2
client $ lfs setstripe -E 1M -c 1 -E eof -c 8 --ec 8+2 /lustre/file2
```

Read-only Erasure Coding: Status & Usage

- Current status on master 2.17.58-62: EC is currently guard-protected (removed by 2.18)
- Usage (see [lustre/tests/sanity-ec.sh](#) for further examples):

```
# Create file: 8 data + 2 parity stripes
client $ lfs setstripe --ec 8+2 /lustre/file1
# Create file: One --ec for all components: [0,1M) 1 stripe -> 1+2, [1M,EOF) -> 8+2
client $ lfs setstripe -E 1M -c 1 -E eof -c 8 --ec 8+2 /lustre/file2
# Directory default: every new file inherits it
#      [0,1M) 1+2, [1M,1G) 8+2, [1G,EOF) 32 stripes -> 4 RAID sets of 8+2
client $ lfs setstripe -E 1M -c 1 -E 1G -c 8 -E eof -c 32 --ec 8+2 /lustre/dir
```

Read-only Erasure Coding: Status & Usage

- Current status on master 2.17.58-62: EC is currently guard-protected (removed by 2.18)
- Usage (see [lustre/tests/sanity-ec.sh](#) for further examples):

```
# Create file: 8 data + 2 parity stripes
client $ lfs setstripe --ec 8+2 /lustre/file1
# Create file: One --ec for all components: [0,1M) 1 stripe -> 1+2, [1M,EOF) -> 8+2
client $ lfs setstripe -E 1M -c 1 -E eof -c 8 --ec 8+2 /lustre/file2
# Directory default: every new file inherits it
#      [0,1M) 1+2, [1M,1G) 8+2, [1G,EOF) 32 stripes -> 4 RAID sets of 8+2
client $ lfs setstripe -E 1M -c 1 -E 1G -c 8 -E eof -c 32 --ec 8+2 /lustre/dir

# [... write some data ...]
# Compute & verify parities
client $ lfs mirror resync /lustre/file1
client $ lfs mirror verify /lustre/file1
# Print file layout + EC information
client $ lfs getstripe [--ec-map] /lustre/file1
```

Read-only Erasure Coding: Status & Usage

- Current status on master 2.17.58-62: EC is currently guard-protected (removed by 2.18)
- Usage (see [lustre/tests/sanity-ec.sh](#) for further examples):

```
# Create file: 8 data + 2 parity stripes
client $ lfs setstripe --ec 8+2 /lustre/file1

# Print EC information
client $ lfs getstripe --ec-map /lustre/file1
components:
  - lcme_id:          131074
    lcme_ec:          8+2
    lcme_ec_raidset_count: 1
    lcme_ec_raidsets:
      - 0: { ec_data_count: 8, data_stripes: "0-7", parity_stripes: "0-1" }
```

Read-only Erasure Coding in 2.18: Status

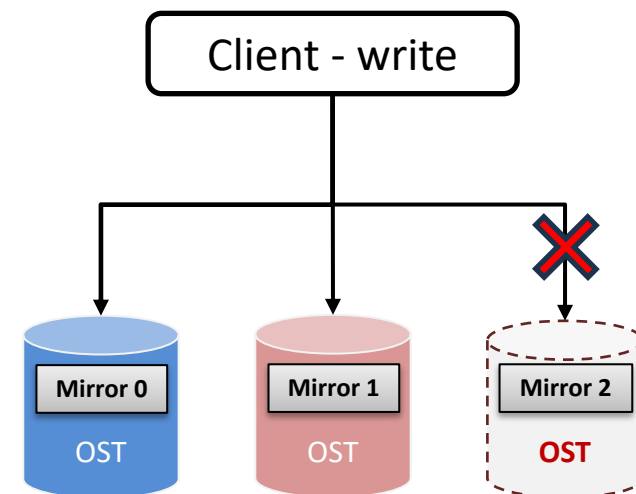
- As of September 2026 (2.17.58): Code complete with further fixes in review
- The core of the feature is landed (40+ patches landed in 2026); a few highlights:
 - Layout and creation: `--ec k+m`, parity mirror bound to its data mirror ([LU-12187](#), [LU-19545](#))
 - Parity lifecycle: resync and verify, block-granular, holes skipped ([LU-12668](#))
 - RAID-set-aware OST allocation ([LU-19852](#))
 - Degraded read: parity reconstruction on the client ([LU-12669](#))
 - Tooling and more
- Missing patches in-line for landing (in review):
 - Ifs mirror extend ([LU-19548](#))
 - Failure domains integration into stripe allocator ([LU-19823](#))
 - Single stripe replacement ([LU-20534](#))
- Further fixes in review before 2.18 code freeze and guards removal

The Road Ahead:

Immediate Write Mirroring & Immediate Write Erasure Coding

Phase 2: Immediate Write Mirroring (IWM)

- Clients send every write to all mirrors immediately
 - A write epoch is opened and closed with the MDT; the MDT picks a write leader
 - A mirror that fails a write is marked stale; continues on others
- Higher space overhead, but can be used for critical files
 - Checkpoints, databases, other commonly modified files
- Space overhead can be reclaimed after a file becomes idle
 - Mirror at creation and migrate to read-only EC when idle
- IWM builds infrastructure used for immediate write EC
- Design mostly complete, starting implementation planning
 - [Immediate Write Mirroring Design](#)
 - Development will begin while read-only EC is finalizing landing in 2.18
 - Expected development time ~6-8 months (comparable to RO-EC)

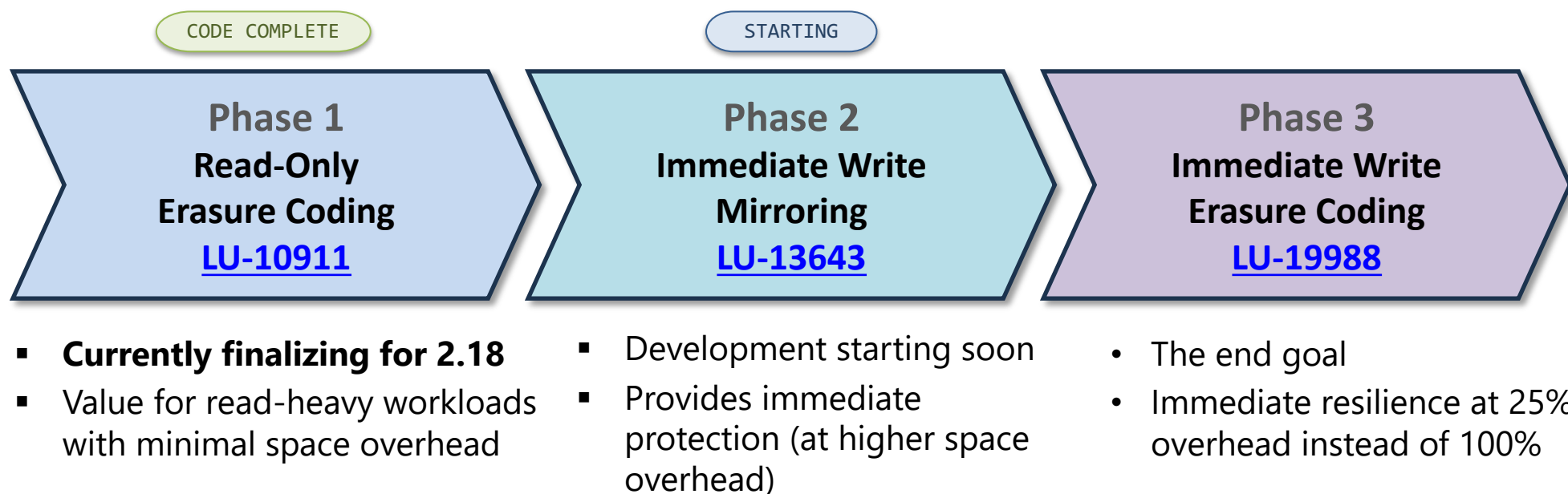


Phase 3: Immediate Write Erasure Coding (IWECC)

- Erasure-code data as it is written with full fault tolerance throughout the process
- End goal: Full protection with EC efficiency
 - vs. read-only EC: Protection during write phase
 - vs. IWM: EC efficiency and lower write overhead, no re-read when creating delayed parity
- Builds on the IWM infrastructure to update data and parity concurrently
- Status and outlook
 - Design outline complete (further design needs IWM design to be completed)
 - Expected development time ~6-8 months (comparable to RO-EC and IWM)
 - More to say soon when IWM is in motion

Conclusion and the Road Ahead

Progressive improvements that reduce impact of device failure
Increasing data protection with limited impact on capacity





Thank you! Questions?

Marc-André Vef <mvef@ddn.com>

Patrick Farrell <patrick@thelustrecollective.com>

Backup

Read-only Erasure Coding: Status & Usage

```
# File: [0,1M) 1+2, [1M,1G) 8+2, [1G,EOF) 32 stripes -> 4 RAID sets of 8+2
client $ lfs setstripe -E 1M -c 1 -E 1G -c 8 -E eof -c 32 --ec 8+2 /lustre/file
# Print file layout + EC information after stripes were allocated
client $ lfs getstripe --ec-map /lustre/file
components:
- lcme_id:          131076
  lcme_ec:          1+2
  lcme_ec_raidset_count: 1
  lcme_ec_raidsets:
    - 0: { ec_data_count: 1, data_stripes: "0-0", parity_stripes: "0-1" }
- lcme_id:          131077
  lcme_ec:          8+2
  lcme_ec_raidset_count: 1
  lcme_ec_raidsets:
    - 0: { ec_data_count: 8, data_stripes: "0-7", parity_stripes: "0-1" }
- lcme_id:          131078
  lcme_ec:          8+2
  lcme_ec_raidset_count: 4
  lcme_ec_raidsets:
    - 0: { ec_data_count: 8, data_stripes: "0-7", parity_stripes: "0-1" }
    - 1: { ec_data_count: 8, data_stripes: "8-15", parity_stripes: "2-3" }
    - 2: { ec_data_count: 8, data_stripes: "16-23", parity_stripes: "4-5" }
    - 3: { ec_data_count: 8, data_stripes: "24-31", parity_stripes: "6-7" }
```