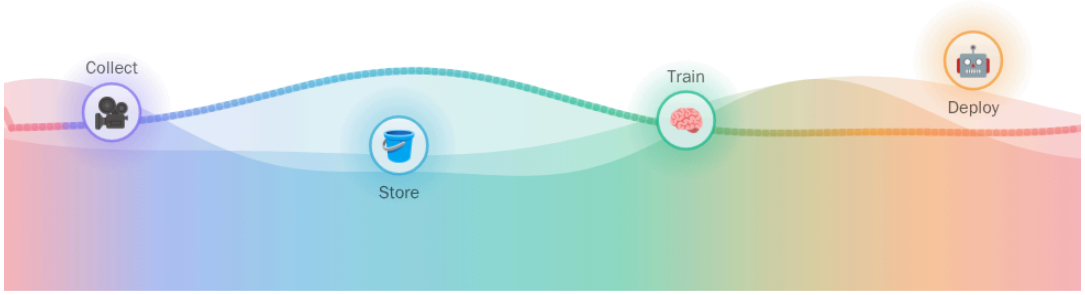
The physical-AI data loop: LeRobot and Hugging Face Buckets



How LeRobot and Hugging Face Storage Buckets keep robot data cheap to store and quick to move, from the first recording to the deployed policy, with no cloud infrastructure to run.

AUTHOR

Steven Palma

AFFILIATION

Hugging Face

PUBLISHED

Jun. 22, 2026

Table of Contents

1 Phase 1: collect and ingest

2 Phase 2: aggregate and deduplicate

3 Phase 3: train at scale

4 Phase 4: deploy and iterate

5 Closing the loop

You already know the `LeRobot` format: state and action tensors in Parquet, multi-camera video in MP4, a `meta/` folder tying them together. The question this article answers is where all that data should live once you outgrow a laptop, and what it costs you when you don't think about it.

Robotics is one of the largest dataset category on the Hugging Face Hub, up from 1,000 datasets in early 2025 to 60,000 in mid 2026 [1]. Every one is heavy, append-only data: a typical robotics setup records data at 140 MB/s, and that has to be stored, moved to GPUs, and shipped back to hardware. That is where the storage bill hides.

Storage Buckets are a new S3-like, Xet-backed repository type, generally available to every user and org since early 2026 [2]. While `LeRobot` does not need them to function, they blend nicely with the physical-AI data loop. A `LeRobot` user gets the benefit of this setup simply by pointing to `hf://buckets/...` paths.

We follow the data through four phases, each pairing one problem with one fix:

Phase 1: collect and ingest

Collection setups record new episodes all day, each a continuous run of camera frames and state-action telemetry, and the `LeRobot` format appends per episode. But where do those bytes land?

A `LeRobotDataset` keeps three things: state and action tensors in Parquet, one MP4 stream per camera, and a small `meta/` folder with the schema, stats, and per-episode index. Many episodes

are packed into a few large shards, so an episode is a slice of a shard located through offset metadata, not a standalone file.

The practical consequence is that a dataset is a few set of large, mutable files that grow as you record. Push them straight to a versioned dataset repository and you fight Git-LFS history: every append is a commit, and every revision is retained forever. During collection you don't want that. You want a fast place to dump bytes and overwrite freely.

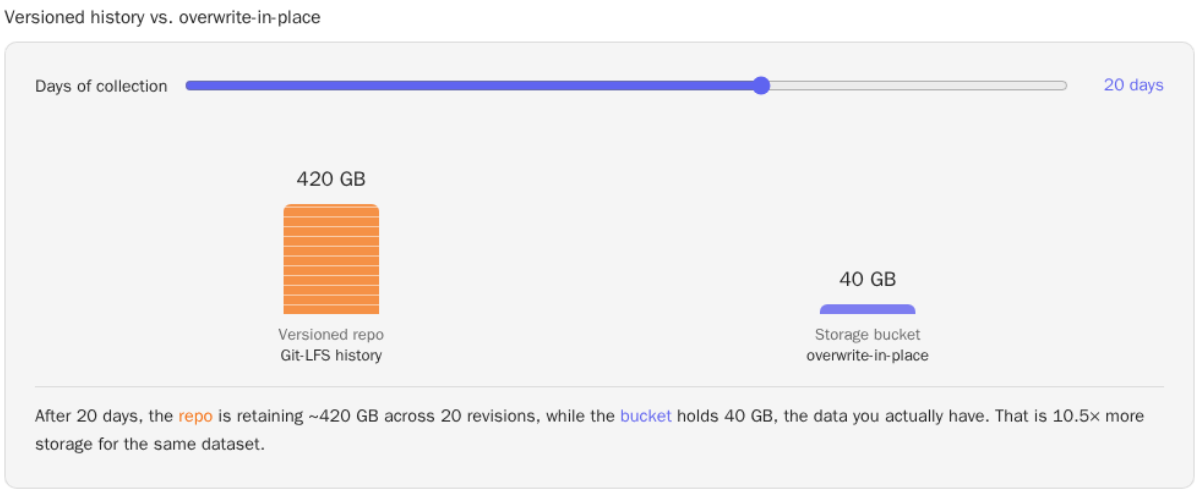


Figure 2 · Each day adds a few hours of data recordings. As you keep re-pushing the growing dataset, a Git-LFS repo retains every revision while a bucket keeps only the live set. Drag to add days.

That is a Storage Bucket: a repository type on the Hub that behaves like an S3 bucket, mutable and overwrite-in-place, inside your Hugging Face workspace with your existing permissions [3]. No IAM roles, no CORS, no upload service to maintain.

> 🔌 Are buckets part of the LeRobot ecosystem?
>
> `LeRobot` records to the `LeRobotDataset` format and `push_to_hub()` targets a dataset repo; there is no bucket-specific code path. What makes buckets usable from a `LeRobot` project is the shared stack: the same `hf://` namespace, Xet backend, `huggingface_hub` client, and `fsspec`/mount tooling. A bucket slots in as the mutable working layer next to your datasets, with nothing new to install.

Collection runs append into one mutable place, published repos stay clean, and you wrote no cloud-storage plumbing. The next three phases are where that backend pays you back.

Phase 2: aggregate and deduplicate

Robot data is extraordinarily redundant. When a robotics arm spends eight hours clearing the same table in front of two 1080×720p cameras, the lighting, the chassis, and most of the background stay identical across thousands of episodes. Naive storage pays for all of it every time, and Git LFS makes it worse: change one frame in a multi-gigabyte video shard and it re-uploads the whole file.

Buckets are backed by Xet, which deduplicates at the byte level using content-defined chunking [4, 5]. Chunk boundaries follow the content, so inserting a few bytes changes only the chunk it lands in instead of shifting every boundary after it. Edit the stream below to see it.

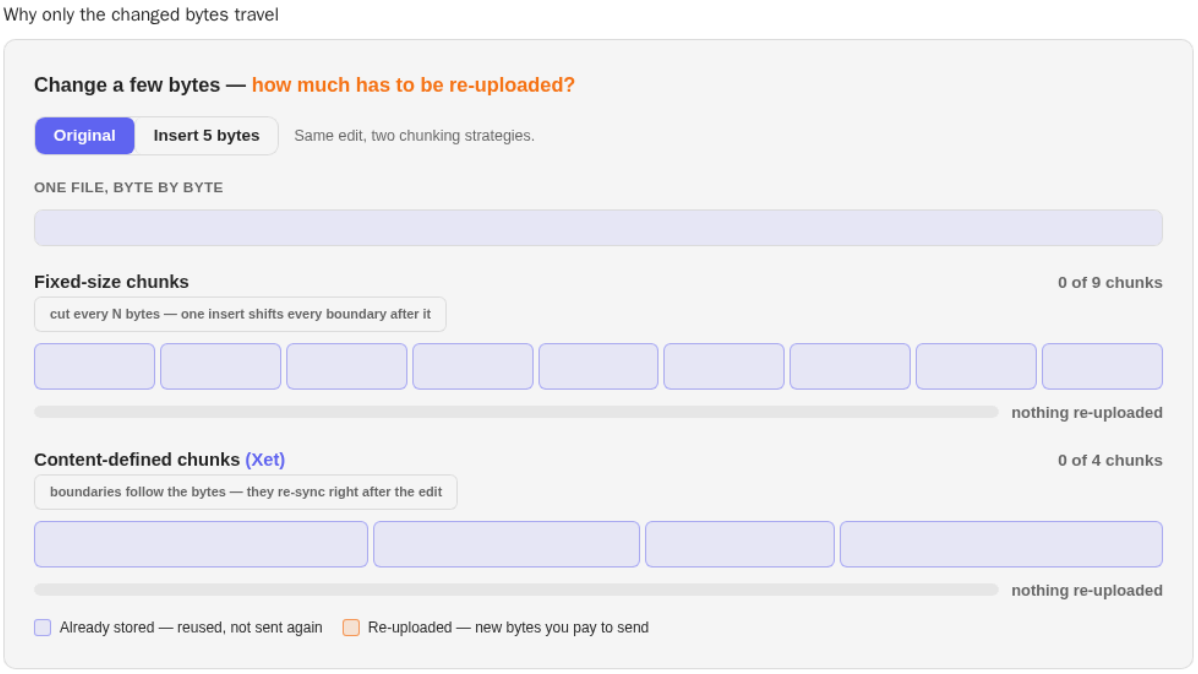


Figure 3 · Fixed-size chunking re-uploads everything after an edit; content-defined chunking only uploads the changed piece. Click to insert a byte and see the difference.

For an append-heavy robot dataset, a daily push uploads roughly the new material plus whatever genuinely changed. Successive π0.5 or GR00T checkpoints with mostly-frozen weights behave the same way [6]. A server-side `hf buckets cp` can even copy a terabyte-scale dataset near-instantly, since it migrates content hashes rather than bytes.

> 🔍 How big is the win, really?
>
> About 4 times less data per upload. 53% on GPT-2 `safetensors`, 30 to 85% across PyTorch checkpoints, and a `gemma-2-9b` GGUF repo that went from 191 GB to 97 GB [7, 5]. On Enterprise plans, billing is on the deduplicated footprint, so the saving shows up on the invoice [3].

Phase 3: train at scale

Training a VLA like NVIDIA's GR00t or a world-action-model like Fast-WAM means pointing GPUs at your datasets. The expensive failure mode is GPUs sitting idle while hundreds of gigabytes download. Pick a mode below to see where the bottleneck moves.

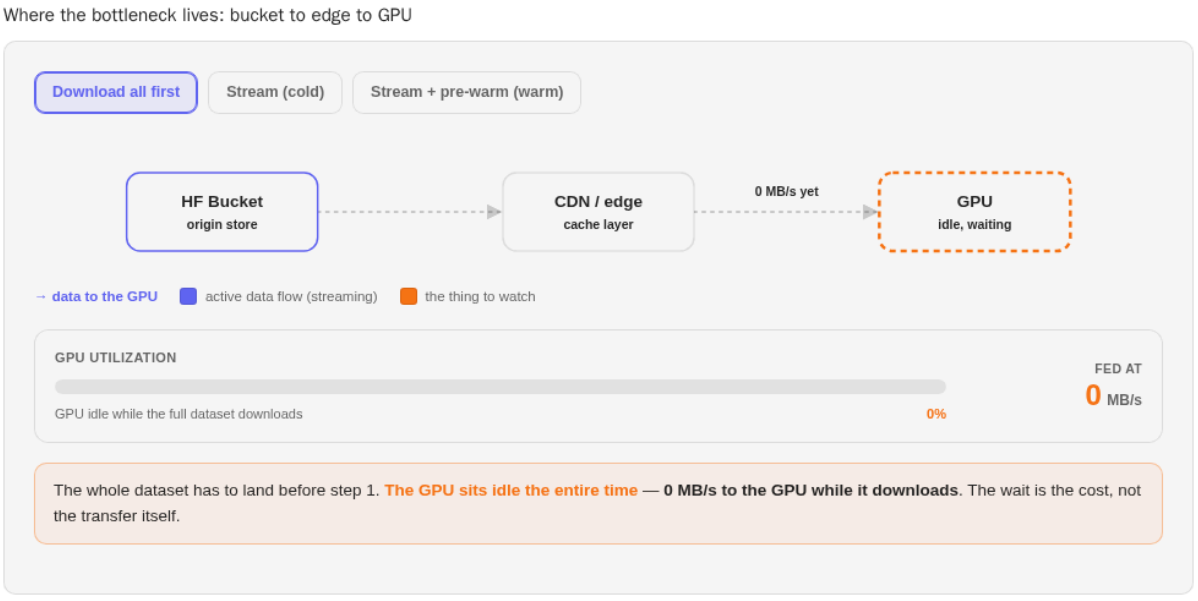


Figure 4 · Naive (download-everything-first) stalls the GPUs on a cold read. Streaming reads byte ranges on the fly; pre-warming the CDN pushes the bottleneck off the origin. Pick a mode to see the limiting hop and its effective throughput.

The shard layout from Phase 1 is what makes Hub-native streaming practical: a batch is a few byte-range reads over large shards, not thousands of tiny fetches. `StreamingLeRobotDataset` turns that into a drop-in `torch` iterable that trains directly from the Hub, with nothing on local disk:

```python
from lerobot.datasets import StreamingLeRobotDataset

ds = StreamingLeRobotDataset(repo_id="my-org/pick-place")  #streaming=True
for batch in torch.utils.data.DataLoader(ds, batch_size=64):
    loss = policy(batch).loss
    loss.backward()
```

Pre-warming then caches data at edge locations near the cloud and region where your jobs run, on AWS and GCP, so the cluster reads locally [3]. In Hugging Face's benchmark a bucket served roughly 1,100 MB/s cold and up to about 1,326 MB/s warm, against a generic object store around 310 to 420 MB/s [8].

Phase 4: deploy and iterate

A finished run emits a policy checkpoint: roughly a gigabyte for SmolVLA, several for a π0.5-scale model. You push it onto physical robots to evaluate, then pull the next dataset back to improve the policy. The loop only works if moving data is cheap, because you want to do it often. The real version of this already exists: record demonstrations, push them as a `LeRobotDataset`, validate in simulation, then deploy the same code to hardware by passing a Hub checkpoint id.

```
lerobot-rollout --policy.path=my-org/pick-place-vla \
  --strategy.type=sentry \
  --robot.type=rebot_b601_follower \
```

On a self-managed store, the storage rate is the part you see coming. The surprise is egress, the data transferred out of the cloud to your robots or another region: every checkpoint shipped and every dataset pulled is metered per gigabyte, and for a busy fleet it can exceed the storage bill. Hugging Face Storage includes egress and CDN at no extra cost up to an 8:1 ratio of stored volume [9], so for a fleet that ships daily the line item largely disappears. Price your own footprint:

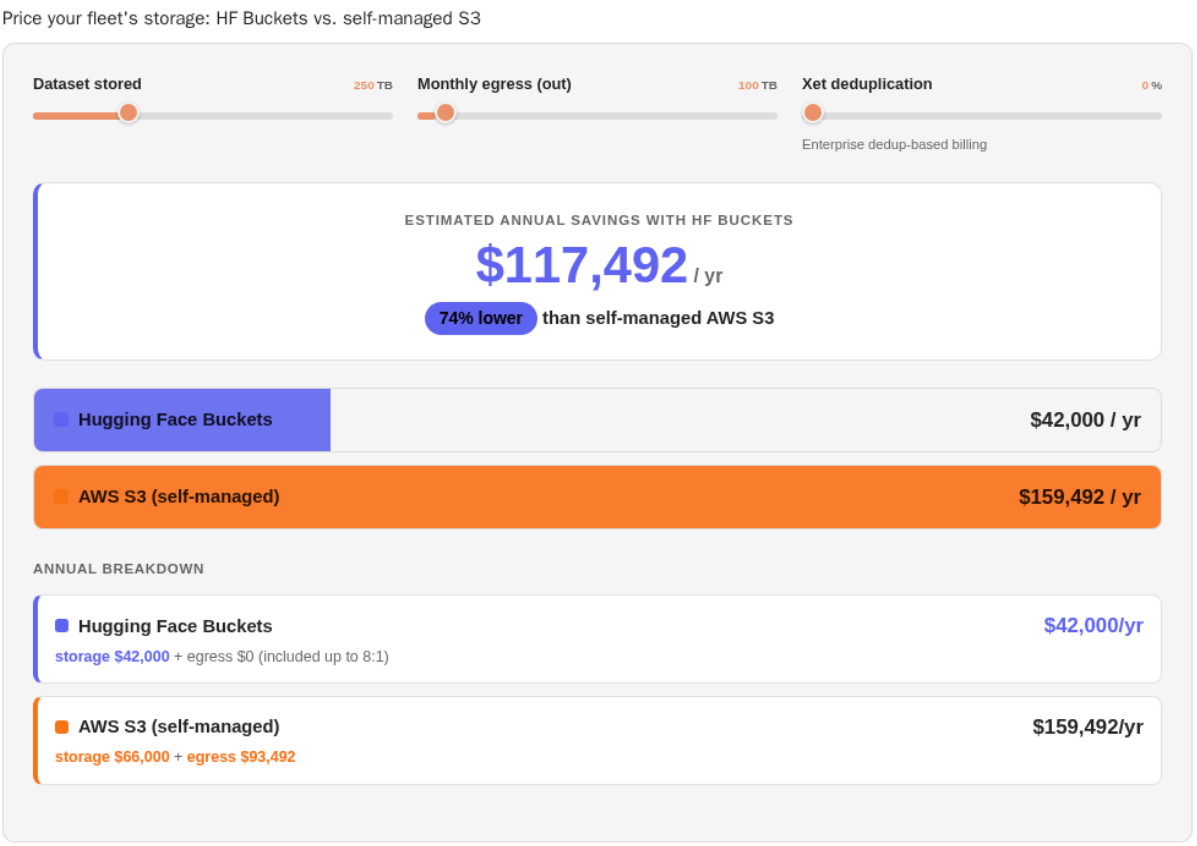


Figure 5 · HF egress is included up to 8:1 of stored volume; AWS S3 storage and per-GB egress are tiered. Defaults: 250 TB stored, 100 TB/month egress, private.

For a mid-sized fleet of YAM arms with 250 TB of recorded episodes, moving 100 TB of egress per month, the private-bucket total is about $42,000/year against roughly $160,000 on self-managed S3, where egress alone is about $93,500. That is close to a 74% reduction [9, 10], and it reproduces from the public cost calculator's own pricing.

Closing the loop

The four phases share one backend. You collect into a mutable bucket instead of bloating a git repo, Xet uploads and bills only the bytes that changed, `StreamingLeRobotDataset` and a pre-warmed CDN feed the GPUs without a download stall, and included egress keeps a fleet of robots shipping fresh VLA or World Models checkpoints without running up a transfer bill. None of it asked you to run cloud infrastructure, so your time goes to policies and hardware.

Buckets and `LeRobot` share the namespace, the Xet backend, and the tooling, which is enough to make them the storage layer the physical-AI loop wants.

🚀 Try it in five minutes

Install the CLI and sign in:

```
curl -LsSf https://hf.co/cli/install.sh | bash
hf auth login
```

Create a bucket and sync a recording into it:

```
hf buckets create my-org/robot-fave --private
```

```
hf sync ./recordings hf://buckets/my-org/robot-fave/run-021
```

Steven Palma
@imstevenpmwork

Citation

For attribution in academic contexts, please cite this work as

```
Steven Palma (2026). "The physical-AI data loop: LeRobot and Hugging Face Buckets".
```

BibTeX citation

```
@misc{palma2026_the_physical_ai_data_loop_lerobot_and_hugging_face_buckets,
  title={The physical-AI data loop: LeRobot and Hugging Face Buckets},
  author={Steven Palma},
  year={2026},

}
```

Reuse

References

1. Palma, S. (2026). *LeRobot Project Pulse: adoption dashboard*. Hugging Face Space. https://huggingface.co/spaces/imstevenpmwork/lerobot-adoption-dashboard↑

2. Hugging Face. (2026). *Introducing Storage Buckets: S3-like object storage on the Hub*. Hugging Face Blog. https://huggingface.co/blog/storage-buckets↑

3. Hugging Face. (2026). *Storage Buckets*. Hugging Face Hub Documentation. https://huggingface.co/docs/hub/en/storage-buckets ↑ back: 1, 2, 3

4. Hugging Face. (2025). *Xet: Deduplication*. Hugging Face Hub Documentation. https://huggingface.co/docs/hub/en/storage-backends ↑

5. Hugging Face Xet Team. (2025b). *From Files to Chunks: Improving Hugging Face Storage Efficiency*. Hugging Face Blog. https://huggingface.co/blog/from-files-to-chunks ↑ back: 1, 2

6. Hugging Face Xet Team. (2025c). *Xet is on the Hub*. Hugging Face Blog. https://huggingface.co/blog/xet-on-the-hub ↑

7. Hugging Face Xet Team. (2025a). *From Chunks to Blocks: Accelerating Uploads and Downloads on the Hub*. Hugging Face Blog. https://huggingface.co/blog/from-chunks-to-blocks ↑

8. h-m-t. (2026). *HF Buckets throughput benchmark*. Hugging Face Space. https://huggingface.co/spaces/h-m-t/hf-buckets-benchmark ↑

9. Hugging Face. (2026). *Hugging Face Storage: pricing and storage products*. huggingface.co/storage. https://huggingface.co/storage ↑ back: 1, 2

10. Lepers, A. (2026). *Storage cost calculator*. Hugging Face Space. https://huggingface.co/spaces/AdrianLepers/storage-cost-calculator ↑