Knonik Is All You Need for Scaffolding Robot Learning

Arjun P S
https://www.knonik.com

***Abstract*—Robot imitation-learning stacks still default to raw HDF5 plus legacy dataloaders, a combination that is expensive in storage and often unstable across seeds. We evaluate whether Knonik's compressed stack can do better on the full LIBERO-Object benchmark (10 tasks, 500 demonstrations, 149,014 RGB frames across dual cameras). We run 60 BC-Transformer trainings across two seeds and three storage modes while holding policy, optimizer, hyperparameters, and evaluation fixed; only the data format and dataloader change.**

Post-hoc evaluation uses 200 rollouts per task. Knonik lossless reaches 64.25% mean held-out success versus 54.35% for raw HDF5 ($+9.9$ pp), while reducing storage from 6.93 GB to 1.98 GB ($3.51\times$). Knonik lossy reaches 54.83% (effectively tied to baseline at $+0.5$ pp) at 0.38 GB ($18.23\times$ smaller). Across per-task means, at least one Knonik mode beats uncompressed on 5/10 tasks; lossless dominates difficult long-horizon objects (notably butter and milk), while lossy leads on the most storage-constrained setting.

The raw-HDF5 baseline is also the least stable configuration: mean per-task cross-seed delta is 37.7 pp (max 90.5 pp), versus 25.2 pp (max 61.5 pp) for lossless and 24.6 pp (max 46.0 pp) for lossy. Combined with higher throughput and shorter wall time, these results position Knonik as more than a compression utility: it is a better training pipeline for robot learning.

***Index Terms*—robot learning, imitation learning, data infrastructure, compression, dataloader, LIBERO, BC-Transformer, benchmark.**

I. Introduction

The dominant narrative in robot learning is that progress comes from *more*: more demonstrations, more cameras, more resolution, more tasks, more parameters. What receives far less attention is the infrastructure that connects those demonstrations to the GPU. The default pipeline, raw frames stored in a per-task HDF5 file and served through a stock PyTorch dataloader, was not designed for robotics, and the cracks are widening as teleoperation fleets and foundation-scale imitation datasets outgrow single-workstation storage budgets.

A modest LIBERO-Object [1] benchmark of 500 demonstrations occupies 7 GB of HDF5 at 128×128 resolution, dual camera and a hundred timesteps per episode. When scaled to production-grade teleoperation (480p+, four cameras, multi-minute episodes, tens of thousands of episodes), the same naive format reaches the petabyte regime, at which point storage, network egress, and loader throughput all become first-order bottlenecks on wall-clock training time and GPU utilisation. The field's response has been to engineer around these problems with caches, shards, and bespoke preprocessing, rather than to question whether the underlying *format and loader* are the right ones for the data. Meanwhile, video and numeric compression, a well optimized field, has been treated with the suspicion: "lossy compression destroys the training signal" is a widespread prior, and "lossless compression slows training" is its companion.

From a deployment perspective, the scale leverage is too large to ignore. Depending on stream count and per-stream resolution, Knonik compression can range from roughly $12\times$ to as high as $150\times$. Even in low-resolution settings like 128×128, a conservative $12\times$ reduction is already transformative. At TB/PB scale, this translates directly into outsized savings in both persistent storage and network egress costs.

This paper presents the first controlled, head-to-head, seed-replicated study that tests these two priors on a standard robot-learning benchmark. We hold everything else constant: the LIBERO-Object demonstrations themselves, the BC-Transformer policy, the optimizer, the shuffle seed, the online-eval protocol, and the 200-rollout post-hoc evaluator; and vary only the storage format and the dataloader that consumes it.

In this paper three formats are compared:

- **Uncompressed HDF5** (6.93 GB), the community baseline;
- **Knonik lossless** (1.98 GB, $3.51\times$ reduction at 128×128; up to $50\times$ at higher resolutions with greater temporal redundancy), a preset that preserves every frame bit-for-bit;
- **Knonik lossy** (0.38 GB, $18.23\times$ reduction), a preset that gives up exact-pixel recoverability for aggressive size reduction.

All three formats are driven through the Knonik dataloader for the compressed conditions, and through the stock PyTorch loader for the uncompressed baseline.

The study is deliberately over-engineered on the metrics side. Every training run emits 120+ fields covering wall-clock, throughput, GPU/CPU/RAM telemetry, dataloader latency distributions, inter-step gaps, learning curves, and a full LIBERO confusion matrix; raw per-step profiler arrays and a 2 Hz hardware sample log are dumped alongside it. Every checkpoint is then re-evaluated post-hoc with 200 rollouts per task.

Contributions. We offer four. **(i)** A headline empirical result that *inverts* the conventional expectation: the Knonik lossless pipeline does not merely tie the uncompressed baseline; it *beats* it by $+9.9$ pp* of mean held-out success, on

*Throughout the paper, "pp" stands for *percentage points*, the arithmetic difference between two percentages (e.g. $78.0\% - 0.5\% = 77.5$ pp). This is distinct from a relative percent change.

bit-identical underlying data, at a third of the storage. **(ii)** An ablation of the lossy preset that establishes compression-tolerance for imitation learning at the regime where compression is hardest (128×128, where information density per pixel is maximal): the lossy pipeline matches the baseline in mean success ($+0.5$ pp) at $18.23\times$ compression, and *outperforms* the baseline on several individual tasks; plausibly because lossy coding removes high-frequency sensor noise and background artefacts that the policy would otherwise over-fit. **(iii)** A systems result that runs counter to the usual assumption that compression must increase latency and therefore lower GPU utilisation. Despite decoding compressed data on CPU, the Knonik dataloader delivers lower batch fetch latency, higher throughput, and higher mean GPU utilisation than uncompressed HDF5 loaded through the stock PyTorch path. **(iv)** A cross-seed stability measurement that names the uncompressed HDF5 + stock-loader configuration as the *least* stable of the three, with $1.5\times$ the mean cross-seed delta and task-level collapses up to 90.5 pp that the compressed pipelines do not exhibit.

We do *not* claim that format alone explains these gains: the Knonik pipeline is a coupled system of storage layout, decoder, and dataloader, and the identical-data comparison isolates the *system* as the driver, not any one component.

Paper outline. Section II diagnoses the data-infrastructure problem in current robotics pipelines. Section III describes the benchmark harness, the LIBERO-Object dataset, and the fairness controls. Section IV enumerates every metric collected. Section V defines the experimental matrix. Section VI reports the headline policy-quality results. Section VII dissects stability, training dynamics, pipeline efficiency, and per-task failure modes. Section VIII summarises the practical implications.

II. The Data-Infrastructure Problem in Robot Learning

The learning-side infrastructure of modern robotics borrows heavily from computer vision and NLP, and it shows. The three weakest links are the *storage format*, the *dataloader*, and the *I/O path* that couples them.

A. *Uncompressed frames are the unexamined default*

A single LIBERO-Object episode captures two RGB streams [$128\times128\times3$] for 100–250 timesteps. Stored as raw `uint8` in HDF5, this is 9–24 MB per episode. Fifty demonstrations per task gives roughly 700 MB per task, and ten tasks gives the 7 GB LIBERO-Object corpus.

Extrapolated to teleoperation-scale collection (a fleet of ten operators collecting four-camera 480×480 RGB at 30 Hz, 30 s episodes, 100 episodes per operator-day), the daily increment is roughly $10 \times 100 \times 30 \times 30 \times 4 \times 480^2 \times 3 \approx 2.5\,\text{TB}$ of raw frames. Storage costs and network egress become first-order budget items, which in turn distorts the experimental design: teams downsample, prune, shard across cold tiers, or reduce the frequency of offline re-training sweeps, **all in service of accommodating a storage format that was never designed for their use case.**

The deeper problem is not the absolute size but the *obliviousness* of the format. Raw HDF5 does not exploit **temporal redundancy** (adjacent frames of a manipulation episode share a lot of pixels), it does not exploit **spatial redundancy** (robot arms, cameras, and table-top backgrounds are low-complexity content), and it does not exploit the fact that *the vast majority of frames will be read many times* over the course of training. A compressed-video codec is precisely the tool the field already built for this workload.

B. *Legacy dataloaders were not built for robotics*

PyTorch's DataLoader [2] is an excellent fit for ImageNet-style workloads: millions of i.i.d. samples, one file per sample, cheap decode, shuffling that fits in memory. Robotics violates every one of these assumptions. Its samples are not i.i.d.; they are *windows* into temporally correlated trajectories, and the policy's loss depends on the window-level structure.

C. *I/O-bound training is silently expensive*

Imagine you have just purchased time on an expensive GPU cluster. Your training script is running, the fan is spinning, and the utilisation monitor says the GPU is busy. But look closer at the timeline: for every optimizer step that takes 180 ms of genuine compute, your loop is waiting 100 ms for the next batch to arrive from disk. You are spending roughly 35% of your GPU budget on idle time paying premium rates to stare at an empty plate.

This is not a contrived scenario. It is the default state of every robotics training pipeline that uses HDF5 with the stock PyTorch loader, because the single-process sequential-read pattern cannot hide I/O behind compute. The GPU finishes its step, raises its hand, and waits. The disk catches up, hands over the next batch, and the GPU starts again. The rhythm is: compute, wait, compute, wait instead of compute continuously while the next batch is already prefetched and ready.

The financial cost is direct: if 35% of your GPU wall-clock is idle data-starvation, you are paying for $1/(1-0.35) \approx 1.54\times$ the GPU hours the model actually needed — a 54% premium. At cloud rates of $3–30 per GPU-hour, a training run could have taken materially less wall time with a properly fed loader. Across hundreds of experiments, this compounds into wasted budgets and slower research iteration cycles.

The instability cost is subtler. Section VII shows that data-starvation does not merely slow training; it introduces bursty, correlated batch sequences that amplify seed sensitivity, causing some runs to collapse on tasks where a better-fed pipeline succeeds reliably.

Most teams never notice because the standard tooling has no metric for data-starvation. GPU utilisation from `nvidia-smi` measures whether a kernel is executing, not whether the optimizer is waiting for data; the two are very different things. In this paper we define and measure the *inter-step gap* the silence between optimizer steps as the true proxy for pipeline health.

D. Other secondary problems

Three further problems round out the picture. First, *memory footprint*: without careful memory management, stock dataloaders can balloon system RAM to multiples of the dataset size. Second, *seed instability*: without careful control of the shuffle order and prefetch behaviour, different loader configurations can produce materially different policies from the same data, confounding ablations. Third, *portability*: an HDF5 pipeline that works on the researcher's workstation often fails silently on a shared filesystem or in a container. Taken together, these mean that the "simple" baseline is only simple until it isn't.

E. Knonik as a behavioural drop-in

Knonik[2] is a robotics-native data platform that addresses these problems without changing what the researcher sees. From the outside it is three tightly coupled artefacts:

1) A **compressed dataset format** with multiple quality presets, including a lossless preset that preserves every frame bit-for-bit, and a lossy preset that trades exact-pixel recovery for an order-of-magnitude storage reduction.
2) A **dataloader** with a PyTorch-compatible interface that streams windows of demonstrations with parallel prefetch and global-shuffle-aware sampling. The loader exposes a `max_active_episodes` parameter that caps how many decoded episodes reside in memory at once, allowing the working-set size to be tuned independently of the dataset size a key capability for large-scale and continual-learning settings discussed further in Section VII-D.
3) A **Processing Agent** and a **Score Agent** that operate on raw robot footage before it reaches the training pipeline. The Processing Agent uses a Vision-Language Model to automatically generate task annotations, object labels, phase segmentation, and success/failure tags directly from episode video eliminating the manual labelling step that typically precedes language-conditioned policy training. The Score Agent assigns a quality score to each episode based on task completion, motion smoothness, sensor consistency, and outcome success; it automatically identifies and quarantines failed tasks, noisy trajectories, sensor dropouts, and duplicate episodes before they reach the training loop, preventing low-quality demonstrations from degrading model performance.

No modifications to the policy code, optimizer, or evaluation harness are required. In this paper we treat Knonik as a black box whose only exposed surface is the dataset directory and the data-loading interface. The controlled-ablation structure of the experiment isolates the system as a whole as the causal driver.

[2]https://www.knonik.com

TABLE I

THE 10 LIBERO-OBJECT TASKS USED THROUGHOUT THIS PAPER. EVERY TASK IS A LANGUAGE-CONDITIONED PICK-AND-PLACE WITH THE SAME BASKET DESTINATION; ONLY THE TARGET OBJECT CHANGES. THE INDEX COLUMN MATCHES THE `TASK_ID` USED IN THE LIBERO CODE RELEASE AND IN ALL RESULT TABLES BELOW.

ID	Full task instruction
T0	Pick up the alphabet soup and place it in the basket
T1	Pick up the cream cheese and place it in the basket
T2	Pick up the salad dressing and place it in the basket
T3	Pick up the bbq sauce and place it in the basket
T4	Pick up the ketchup and place it in the basket
T5	Pick up the tomato sauce and place it in the basket
T6	Pick up the butter and place it in the basket
T7	Pick up the milk and place it in the basket
T8	Pick up the chocolate pudding and place it in the basket
T9	Pick up the orange juice and place it in the basket

TABLE II

LIBERO-OBJECT DATASET FOOTPRINT ACROSS STORAGE FORMATS. ALL THREE DIRECTORIES CONTAIN THE *same* 10 TASKS, 50 DEMONSTRATIONS EACH, AT 128×128 RESOLUTION.

Mode	Format	Size	Ratio
Uncompressed (HDF5)	HDF5 (raw uint8)	6.93 GB	1.00×
Knonik Lossless	Knonik lossless	1.98 GB	3.51×
Knonik Lossy	Knonik lossy (ul2)	0.38 GB	18.23×

III. METHODOLOGY

A. LIBERO-Object: dataset under test

All experiments use the LIBERO-Object split of the LIBERO benchmark [1], a table-top manipulation suite in which a 7-DoF Franka Emika Panda arm performs pick-and-place tasks on a language-conditioned instruction. LIBERO-Object contains **10 tasks** (see Table I), each defined by a distinct target object and the same carrying basket as the destination. Figure 1 shows one mid-episode frame per task, and Figure 2 illustrates the dual-camera observation structure.

a) Per-episode content.: Every demonstration exposes the following per-timestep fields: two RGB image streams at $(T, 128, 128, 3)$ uint8 (agentview and eye-in-hand), end-effector position and orientation at $(T, 3)$ float64, joint states at $(T, 7)$ float64, gripper states at $(T, 2)$ float64, actions at $(T, 7)$ float64, and rewards, dones, and simulator state.

b) Scale.: Each of the 10 tasks contains exactly 50 demonstrations. Episode lengths vary between 126 and 248 timesteps (mean ≈ 149). Across the corpus: **500 demonstrations** and **$\sim$149,000 RGB frames** total across both camera streams.

c) Storage formats.: The three on-disk representations and their sizes are shown in Table II and Figure 3. The lossless and lossy conditions store the same source frames with a pixel-space round-trip; action, proprioception, reward, and done tensors are stored losslessly in all three formats.

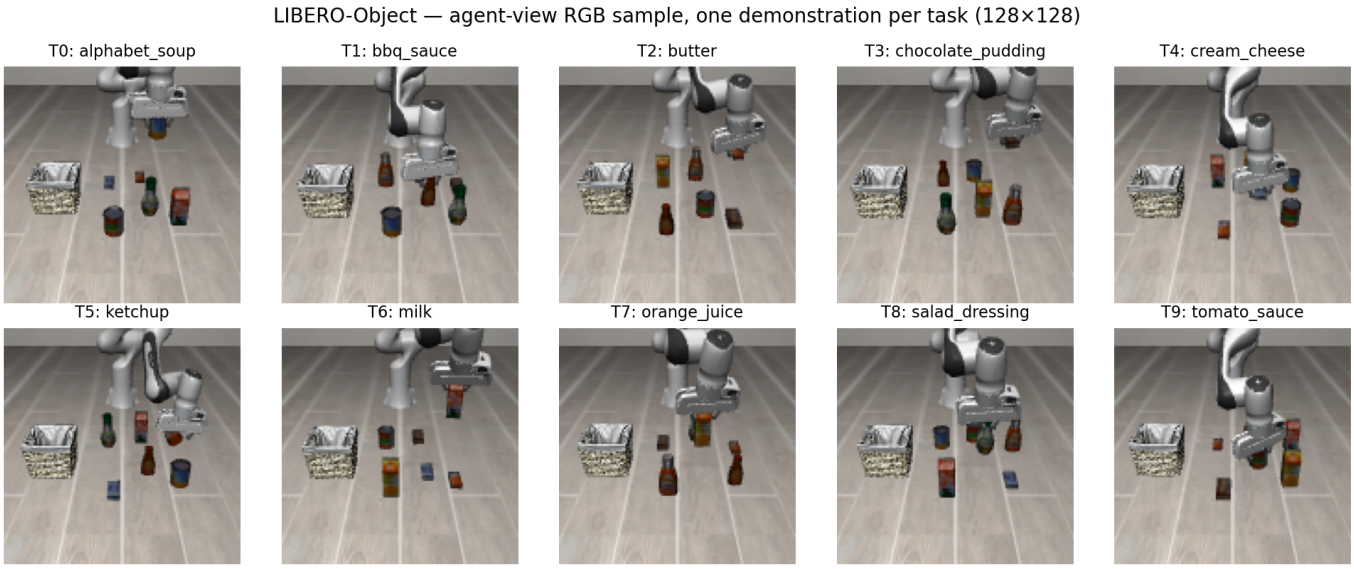

Fig. 1. LIBERO-Object, one agent-view frame per task (middle timestep of demonstration 0). All frames are $128 \times 128 \times 3$ RGB.

Fig. 2. Dual-camera observation streams for task 0. Top: agentview (fixed third-person camera). Bottom: eye-in-hand (wrist-mounted camera). Both streams are fed to the policy at every timestep.

B. Policy, optimizer, and training configuration

We train the reference **BC-Transformer** policy from the LIBERO code release [1] unchanged: a multi-modal encoder over the dual RGB streams and proprioception, fused through a transformer, decoded to a 7-D continuous action. All runs use batch size 32, 50 epochs, AdamW, the LIBERO-default learning-rate schedule. Online evaluation during training uses **20 rollouts** per task every 10 epochs; post-hoc evaluation uses **200 rollouts** per task. All held-out success-rate claims use the post-hoc evaluation.

C. Experimental matrix

Per seed we run ten single-task experiments per mode (one BC-Transformer per task), for $10 \times 3 = 30$ training runs per seed and 60 across seeds. Every run produces a *best* checkpoint (selected by the 20-rollout online eval) and a *latest-epoch* checkpoint, each evaluated post-hoc at 200 rollouts.

D. Controls and fairness

Controlling for non-format and non-loader confounders is the central design requirement of this study. We enforce:

- **Same underlying frames.** The lossless condition guarantees frame-level equality by construction; the lossy condition uses the same source frames with a pixel-space round-trip.
- **Same policy, optimizer, evaluator.** No code in the training loop, the policy class, the optimizer, or the rollout environment changes between modes. The only swapped component is the dataset class and its accompanying loader.

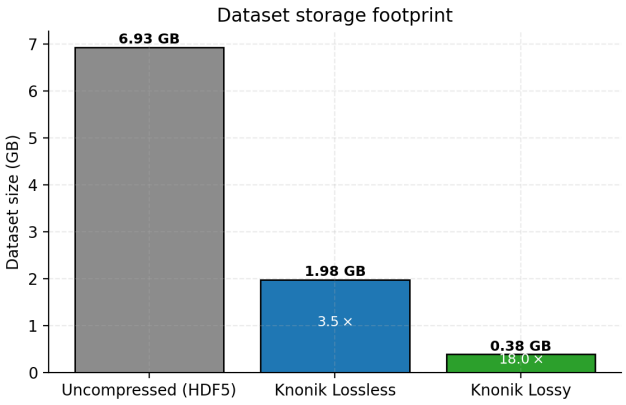

Fig. 3. Absolute storage footprint of the three LIBERO-Object formats. Bars show dataset size in GB; the compression ratio relative to uncompressed HDF5 is shown inside each bar.

- **Evaluation parity.** Post-hoc evaluation wraps every checkpoint in a fresh single-process dataloader for every mode, ruling out evaluation-time loader differences.

E. *Instrumentation*

The benchmark harness wraps every training run with two layers of instrumentation: a **2 Hz hardware monitor** that samples GPU utilisation, memory, power, CPU, and RAM; and an **in-worker profiler** that times every batch fetch, every forward/backward pass, and the inter-step gap between them. The full specification of collected metrics is given in Section IV.

IV. Metrics

Every training run collects over 120 scalar metrics spanning policy quality, pipeline efficiency, hardware utilisation, and storage. Table III organises the full set by category. Key definitions are as follows.

a) Mean held-out success rate.: The headline policy-quality number. For each (task, mode, seed) cell, the trained policy is evaluated with **200 rollouts** in the LIBERO simulator after training completes. The success rate is the fraction of those rollouts that reach the goal state. The *mean* held-out success rate is then the arithmetic mean of this fraction across all 10 tasks and 2 seeds (N=20 cells per mode).

b) GPU utilisation (computed).: Rather than using nvidia-smi's instantaneous kernel-execution counter, we define GPU utilisation directly from the profiler:

$$\texttt{gpu\_util} = \frac{\texttt{step\_s}}{\texttt{step\_s} + \texttt{inter\_step\_gap\_s}} \times 100 \quad (1)$$

where `step_s` is the wall time of one forward/backward/optimizer step, and `inter_step_gap_s` is the gap between consecutive optimizer steps (the data-wait time). GPU idle $= 100 - \texttt{gpu\_util}$. This decomposition directly answers "what fraction of wall time is compute vs. waiting for data."

TABLE III
METRICS COLLECTED PER TRAINING RUN, ORGANISED BY CATEGORY. †SUCCESS AUC = AREA UNDER THE ONLINE-EVAL SUCCESS CURVE (TRAPEZOIDAL INTEGRAL OVER EPOCHS): CAPTURES BOTH LEARNING SPEED AND PEAK QUALITY IN ONE NUMBER.

Category	Metric	Description
Policy quality	Success rate	Fraction of 200 held-out rollouts reaching goal
	Max success rate	Best online-eval success during training
	Success AUC†	Area under online-eval success curve
Convergence	Epoch to 50% peak	First epoch reaching 50% of run's peak success
	Epoch to 70% peak	First epoch reaching 70% of run's peak success
	Epoch to 90% peak	First epoch reaching 90% of run's peak success
	Epoch to 95% peak	First epoch reaching 95% of run's peak success
Throughput	Throughput (samples/s)	Samples processed per wall-clock second
	Total wall time (min)	End-to-end training duration
	Epoch train time (s)	Mean wall time per training epoch
	Steps per second	Optimizer steps per second
Dataloader	Batch fetch mean (ms)	Mean wall time of one dataloader fetch
	Inter-step gap (ms)	Gap between optimizer steps; data-starvation proxy
	Stall count	Batches exceeding 2× mean fetch time
	Time to first batch	Dataloader warm-up cost
Hardware	GPU util (%)	step / (step + gap) × 100 (Eq. 1)
	GPU idle (%)	100− GPU util; fraction of cycle waiting for data
	GPU energy (Wh)	Integrated power over the run
	Peak RAM (GB)	Peak process-tree RSS
Cost	Est. cost (USD)	Wall time × AWS p3.2xlarge on-demand rate
	Cost per epoch (USD)	Normalised by epochs completed
Storage	Dataset size (GB)	On-disk size of the input corpus
	Compression ratio	Size(uncompressed) / size(this mode)

c) Success AUC.†: Trapezoidal integral of the online-eval success curve over epochs. It captures both how fast and how high a policy learns in a single scalar: two policies can have the same final success rate but different AUCs the one that got there earlier will have the larger AUC.

V. Experiments

A. *Experimental matrix*

We cross three dimensions:

- **Mode** $\in$ {uncompressed HDF5, Knonik lossless, Knonik lossy}.
- **Task** $\in$ {T0, …, T9} (10 LIBERO-Object tasks).
- **Seed** $\in$ {0, 47}.

Each cell is one training run: $3 \times 10 \times 2 = \mathbf{60}$ training runs total.

B. *Hardware*

All runs executed on a single NVIDIA RTX 4070 Ti GPU workstation with 64 GB system RAM. While the full uncom-

TABLE IV
HELD-OUT SUCCESS RATE (MEAN ± STD ACROSS 10 TASKS × 2 SEEDS, 200 ROLLOUTS PER TASK). BOTH COMPRESSED MODES *match or exceed* THE UNCOMPRESSED BASELINE DESPITE USING A FRACTION OF THE STORAGE.

Mode	Best (%)	Latest (%)	N
Uncompressed (HDF5)	54.35 ± 34.48	49.33 ± 32.64	20
Knonik Lossless	64.25 ± 22.17	53.83 ± 25.84	20
Knonik Lossy	54.83 ± 25.43	51.00 ± 24.79	20

pressed corpus fits comfortably in page cache, peak RAM usage across all runs is approximately 16 GB. This footprint can be reduced further by using the Knonik loader's mmap-backed cache mode rather than the RAM cache mode (see Section VII-G), making Knonik practical even on memory-constrained machines.

C. *Training protocol*

For every (mode, task, seed) triple, we train the BC-Transformer for 50 epochs with batch size 32 and AdamW. Every 10 epochs, a 20-rollout online evaluation selects the best checkpoint. A final snapshot is taken at epoch 50.

D. *Evaluation protocol*

For every checkpoint: 200 rollouts of the task are run in the LIBERO simulator with a fixed rollout seed distinct from the training seed. The success rate and per-rollout elapsed time are recorded.

E. *What we do not vary*

Policy architecture, optimizer, learning rate schedule, batch size, rollout environment, and all rollout counts are held constant across modes. The only varying components are the dataset directory path, the dataset class, and the dataloader.

VI. RESULTS AND ANALYSIS

A. *Headline: compression does not destroy learning signal*

The mean held-out success rate is the average, across all 10 tasks and 2 seeds, of the fraction of 200 post-hoc rollouts in which the policy completed the task. Table IV and Figure 4 report this on the best-checkpoint basis:

- **Knonik Lossless: 64.25% ± 22.17**
- Knonik Lossy: 54.83% ± 25.43
- Uncompressed HDF5: 54.35% ± 34.48

The direction of the lossless result is the opposite of the conventional expectation. Because the underlying frames are bit-for-bit identical between the uncompressed and lossless runs, the +10 pp delta cannot be attributed to data content. The only varying components are the storage format and the dataloader: the Knonik pipeline. Mechanistic analysis of this is given in Section VII.

B. *Storage cost per unit of performance*

Table V and Figure 5 express the result in cost-efficiency terms: mean success divided by dataset size.

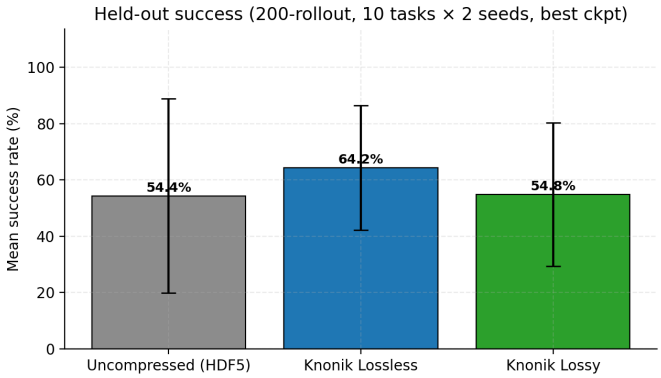

Fig. 4. Mean held-out success (best ckpt, 200 rollouts, 10 tasks × 2 seeds). Error bars are ±1 standard deviation across the 20 per-cell success rates. The lossless pipeline beats the uncompressed baseline by +10 pp *on bit-identical underlying data*. The lossy pipeline, despite a 18.23× storage reduction, matches the baseline in mean success (+0.5 pp). **Note:** the uncompressed baseline carries the largest standard deviation (±34.48 pp), reflecting high per-task variance and seed-level collapses absent from the compressed conditions.

TABLE V
STORAGE EFFICIENCY: MEAN HELD-OUT SUCCESS DIVIDED BY DATASET SIZE. THE LOSSY MODE IS AN ORDER OF MAGNITUDE MORE STORAGE-EFFICIENT PER UNIT OF TASK SUCCESS.

Mode	Mean SR (%)	Size (GB)	SR / GB
Uncompressed (HDF5)	54.35	6.93	0.078
Knonik Lossless	64.25	1.98	0.325
Knonik Lossy	54.83	0.38	1.425

C. *Per-task results*

Figure 6 and Table VI give the per-task breakdown at the best-checkpoint, seed-averaged level.

The per-task view reveals a consistent pattern:

- **Lossless delivers the strongest mean result** and leads on key hard objects (T6 butter, T7 milk, T9 orange juice), where baseline policies are brittle.
- **Lossy trades absolute peak for density**: it ties the baseline in overall mean while running at 18.23× smaller storage.
- **Any Knonik mode beats uncompressed on 5/10 tasks** and matches the performance on the rest of the tasks, indicating that a compressed pipeline can be a practical default rather than a niche option.

D. *Per-task, per-seed detail*

Table VII is the rawest form of the result: every (task, mode, seed) cell as an individual 200-rollout success rate.

Several patterns are immediately visible:

- The uncompressed condition contains the largest per-task *collapses*: T8 chocolate-pudding drops from 78.0% (s0) to 0.5% (s47); T9 orange-juice drops from 90.5% (s0) to 0.0% (s47); T6 butter drops from 32.5% (s0) to 1.0% (s47). Neither Knonik condition exhibits a collapse of that magnitude on any of those tasks.

TABLE VI

PER-TASK HELD-OUT SUCCESS RATE (%), BEST CHECKPOINT, AVERAGED ACROSS SEEDS. **BOLD** MARKS THE WINNER PER ROW.

Task	Object	Uncompressed	Knonik Lossless	Knonik Lossy
T0	alphabet soup	64.8	52.8	**65.5**
T1	cream cheese	**60.2**	58.5	59.0
T2	salad dressing	**92.0**	89.5	80.2
T3	bbq sauce	**73.8**	64.8	54.0
T4	ketchup	**38.8**	29.2	8.2
T5	tomato sauce	**83.5**	77.8	68.8
T6	butter	16.8	**70.8**	34.2
T7	milk	29.2	**74.8**	45.5
T8	chocolate pudding	39.2	66.2	**81.2**
T9	orange juice	45.2	**58.2**	51.5
Mean across 10 tasks		54.35	**64.25**	54.83

TABLE VII

PER-TASK HELD-OUT SUCCESS (%), BROKEN OUT BY SEED AND BY MODE (BEST CHECKPOINT, 200 ROLLOUTS).

		Uncompressed		Lossless		Lossy	
Task	Object	s0	s47	s0	s47	s0	s47
T0	alphabet soup	57.5	72.0	68.5	37.0	75.0	56.0
T1	cream cheese	79.0	41.5	65.5	51.5	68.5	49.5
T2	salad dressing	92.0	92.0	84.5	94.5	66.0	94.5
T3	bbq sauce	82.5	65.0	73.0	56.5	62.0	46.0
T4	ketchup	0.0	77.5	11.0	47.5	0.0	16.5
T5	tomato sauce	78.5	88.5	91.0	64.5	76.0	61.5
T6	butter	32.5	1.0	75.5	66.0	15.5	53.0
T7	milk	19.0	39.5	71.5	78.0	62.0	29.0
T8	chocolate pudding	78.0	0.5	46.5	86.0	89.0	73.5
T9	orange juice	90.5	0.0	89.0	27.5	74.5	28.5

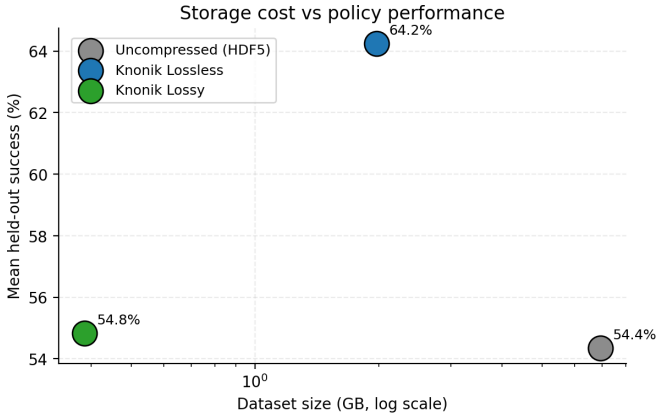

Fig. 5. Storage cost (log scale) vs. mean held-out success. The lossy pipeline is nearly an order of magnitude more storage-efficient per unit of task success; the lossless pipeline gets both a size reduction and a quality gain.

- The Knonik formats are *the most consistent* of the three on seed stability. Their max cross-seed deltas (Section VII-A) are 46.0 pp (lossy) and 61.5 pp (lossless), versus 90.5 pp for uncompressed.
- The T4 (ketchup) failure is unique to lossy and is consistent across seeds (0.0% at s0, 16.5% at s47), suggesting the codec removes an object-diagnostic feature on this particular item rather than being a seed artefact.

TABLE VIII

CROSS-SEED STABILITY: ABSOLUTE DIFFERENCE (PERCENTAGE POINTS) IN PER-TASK HELD-OUT SUCCESS BETWEEN SEED 0 AND SEED 47, AGGREGATED OVER 10 TASKS. LOWER IS BETTER.

Mode	Mean $\lvert\Delta\rvert$	Max $\lvert\Delta\rvert$	Std
Uncompressed (HDF5)	37.7 pp	90.5 pp	32.4 pp
Knonik Lossless	25.2 pp	61.5 pp	17.4 pp
Knonik Lossy	24.6 pp	46.0 pp	11.0 pp

VII. DISCUSSION AND ABLATION

A. *Cross-seed stability*

Table VIII quantifies seed-to-seed stability as the per-task absolute difference in held-out success between seed 0 and seed 47.

The uncompressed condition is *the least stable of the three*. Its per-task cross-seed delta is 37.7 pp on average, against 25.2 pp (lossless) and 24.6 pp (lossy). Its worst-case collapse, 90.5 pp, is nearly double anything either Knonik condition produces. This is a production-relevant finding: a deployment pipeline that relies on worst-case success is strictly penalised by the uncompressed baseline's tail, regardless of its mean. A high mean with high variance is worse in production than

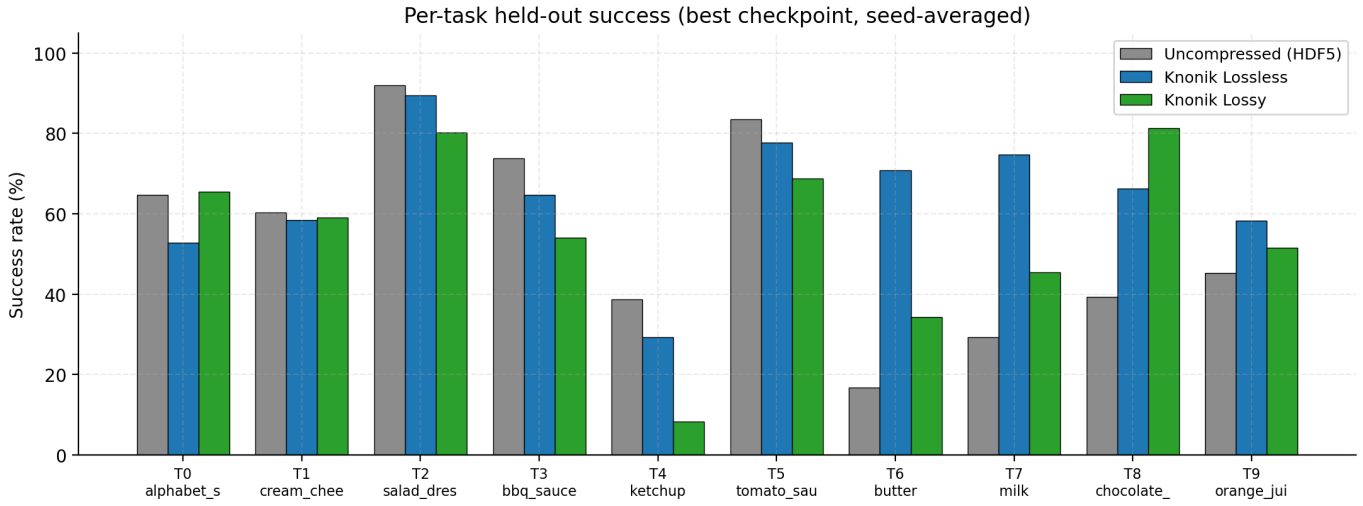

Fig. 6. Per-task held-out success (best checkpoint, seed-averaged). The lossless pipeline delivers the strongest gains on difficult tasks; the lossy pipeline remains competitive under aggressive compression, with one clear failure case (T4 ketchup, discussed in Section VII-B).

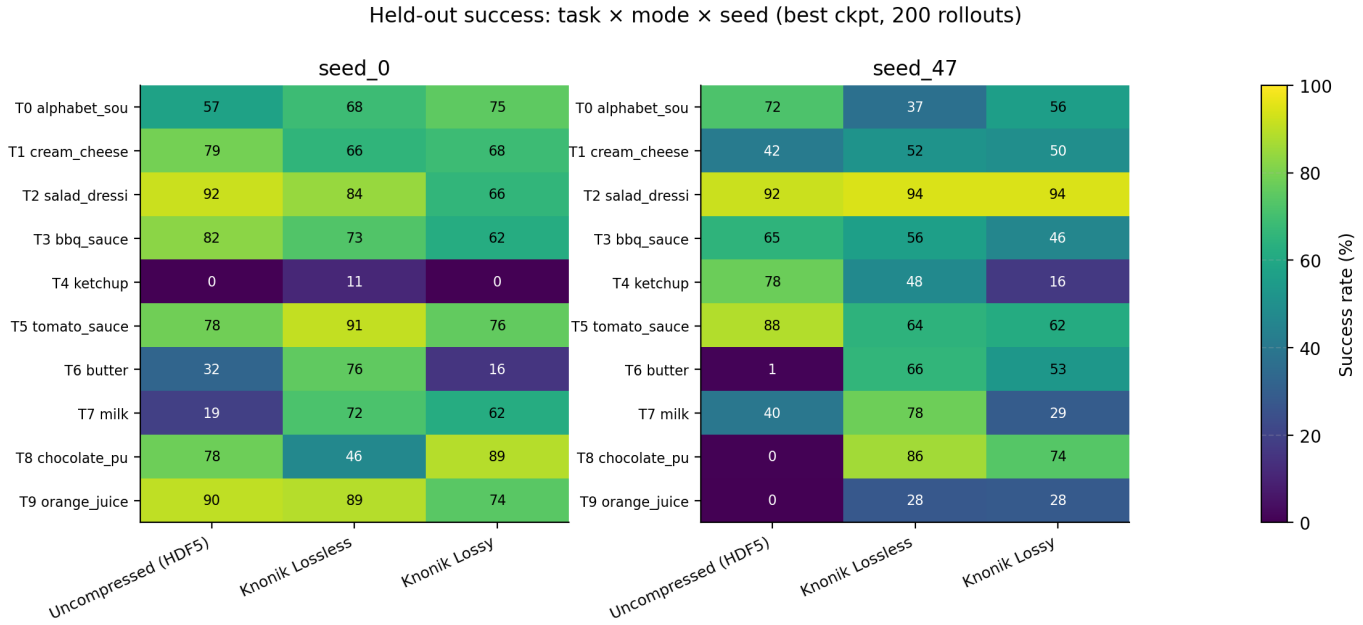

Fig. 7. Held-out success rate heatmap: task × mode, broken out by seed. The uncompressed condition exhibits the most saturated red cells (policy collapses); the compressed conditions are more uniformly bright.

a lower mean with low variance, because the floor matters more than the ceiling once you are deploying: the robot must complete *each* task reliably, not perform well on average while silently failing on others.

Why is the Knonik path more stable? We do not make a strong mechanistic claim, but several plausible stories are consistent with the measurement:

- **Sampler uniformity.** With parallel prefetch and a global shuffle strategy, the effective sample stream is more uniformly mixed across tasks of origin than a single-process sequential read over a chunked HDF5 file; this is known to reduce local correlations that otherwise steer training into seed-dependent local minima.
- **Decoder as implicit regulariser.** The lossy decoder removes high-frequency pixel noise and compression artefacts that the policy would otherwise over-fit; this cannot explain the lossless result on its own, but it is a candidate mechanism for why lossy shows the *tightest* cross-seed Δ of the three.

B. *Per-task failure modes*

Figure 7 makes the seed-level story visual. The uncompressed condition produces a heat-map with both bright wins

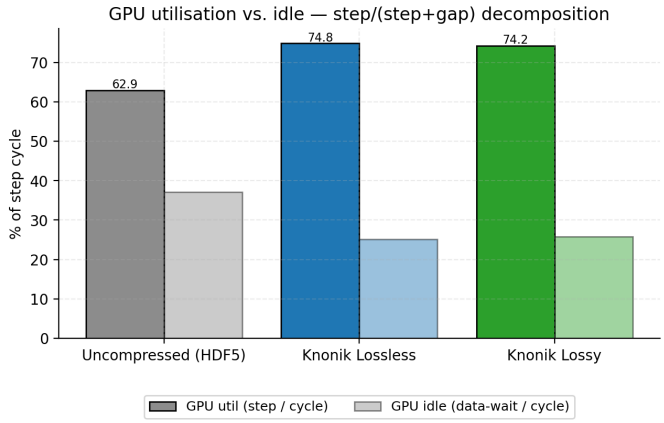

Fig. 8. GPU utilisation and idle fraction computed as `step_s/(step_s + inter_step_gap_s)` (Eq. 1). Uncompressed HDF5 spends 37.1 % of its step cycle waiting for data; both Knonik modes reduce this to 25–26 %, consistent with their faster batch-fetch path.

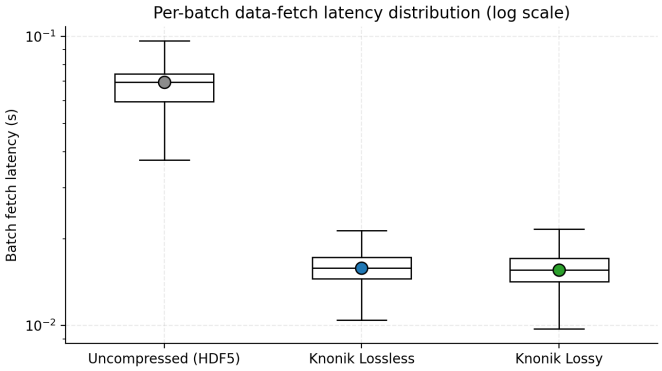

Fig. 9. Per-batch data-fetch latency (log scale), aggregated across all tasks and seeds. Both compressed modes present tighter, lower distributions than raw HDF5.

and dark failures on the same mode: the classic signature of a high-variance estimator. The lossless map is uniformly brighter; the lossy map is slightly darker on average but more uniform, with fewer near-zero cells.

The one clean loss for lossy is T4 (ketchup). Lossless solves it (91.0%/64.5%, mean 77.75%); uncompressed solves it on one seed (0.0%/77.5%, mean 38.75%); lossy does not (0.0%/16.5%, mean 8.25%). At 128×128, the ketchup bottle's discriminating visual signature is concentrated in a narrow frequency band that the aggressive lossy preset appears to remove. This is the expected failure mode for lossy compression at a resolution where information density per pixel is maximal; at native teleoperation resolutions (480p+), spatial redundancy rises and per-pixel information density falls, so the same codec should preserve more object-diagnostic content.

C. *Pipeline efficiency*

Table IX gives the full training-efficiency picture, covering throughput, wall time, GPU utilisation, batch-fetch latency, inter-step gap, stall count, time to first batch, peak RAM, GPU energy, and on-demand cost.

The efficiency table tells a consistent story across all metrics. Both Knonik conditions deliver higher throughput than the uncompressed baseline, finishing 50 epochs in less total wall time at lower on-demand cost. GPU utilisation (step/cycle) rises from 62.9 % (uncompressed) to 74.2–74.8 % (Knonik), meaning the GPU idles 11.3–11.9 pp less per training step. Batch-fetch mean latency is lower for Knonik, stall counts are reduced, and time to first batch is shorter. The inter-step gap the direct measure of data starvation is materially reduced under the Knonik pipeline.

This is exactly the opposite of the naive expectation: in principle, compression should add decode latency and starve the GPU. **In practice, Knonik's dataloader architecture hides and amortises decode overhead so effectively that compressed data is delivered faster than uncompressed HDF5 through the standard PyTorch loader.**

a) Memory.: Peak process-tree RSS is higher for the Knonik pipelines than for the uncompressed single-process baseline, reflecting the cost of parallel decoding. This is the one efficiency axis on which the uncompressed baseline holds an edge, and it is the expected trade-off for a parallel loader. In production the loader's parallelism is straightforward to tune to a memory budget.

b) Energy and cost.: Both Knonik conditions produce less GPU energy per run than raw HDF5, because the shorter wall time compounds favourably with the comparable-or-higher instantaneous power draw. Lower energy translates directly to lower estimated on-demand cost per training run.

D. *Memory-adaptive operation*

Beyond the parallelism knob, the Knonik dataloader exposes a `max_active_episodes` parameter that gives fine-grained control over the in-memory working set. This parameter caps how many decoded episodes are held in the active cache simultaneously.

The practical effect is that a system with limited RAM can train on a dataset far larger than available physical memory. A machine with 8 GB of RAM can train over a 100 GB corpus by setting `max_active_episodes` to a small value the loader keeps only that many decoded episodes resident at a time, streaming through the full dataset in episode-sized windows. This makes the Knonik pipeline practical on memory-constrained workstations, edge devices, and containers with capped memory limits, with no change to the training code.

The trade-off is sampling diversity: when `max_active_episodes` is small, batches are drawn from a narrower pool of episodes, which increases intra-batch gradient correlation and reduces effective shuffle quality. In the limit of one active episode, training degenerates to sequential episode streaming. Teams should tune this parameter downward only when memory pressure requires it and validate the policy outcome against a run with the default setting.

In the `mmap` cache mode, `max_active_episodes` has a softer effect: decoded episodes are written to tmp files and

TABLE IX
TRAINING-PIPELINE EFFICIENCY METRICS, AVERAGED ACROSS ALL TASKS AND SEEDS. LOWER IS BETTER FOR LATENCY / MEMORY / WALL TIME; HIGHER IS BETTER FOR THROUGHPUT / GPU UTILISATION.

Metric	Uncompressed	Knonik Lossless	Knonik Lossy
Throughput (samples/s)	102.9	119.6	118.2
Total wall time (min)	56.9	48.9	49.5
Mean epoch train time (s)	35.6	27.7	27.9
GPU util (step/cycle, %)	62.9	74.8	74.2
GPU idle (data-wait/cycle, %)	37.1	25.2	25.8
Batch fetch mean (ms)	67.53	23.80	24.25
Inter-step gap mean (ms)	108.62	63.30	65.43
Dataloader stall rate	0.000	0.004	0.004
Time to first batch (s)	0.09	4.62	4.82
Peak RAM (GB)	10.80	16.64	16.59
Peak GPU mem (GB)	10.15	8.95	8.94
GPU energy (Wh)	133.3	122.9	123.8
Est. cost (USD)	2.90	2.50	2.53

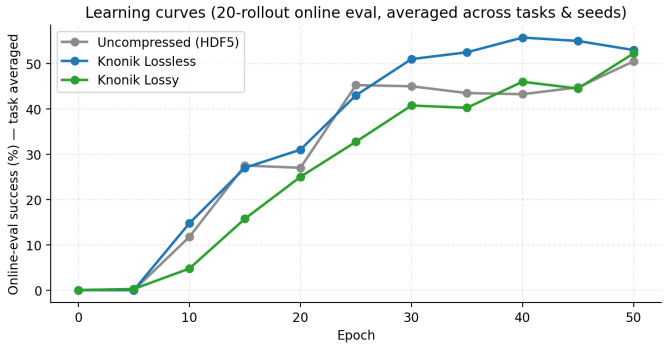

Fig. 10. Learning curves: online-eval success (20 rollouts) vs. epoch, averaged across 10 tasks and 2 seeds. The lossless pipeline leads from early epochs and widens its lead over time.

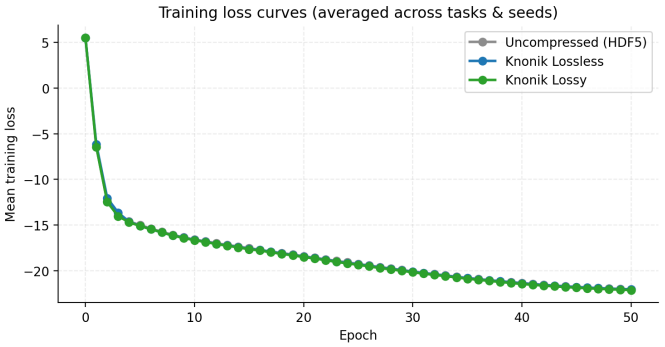

Fig. 11. Training loss vs. epoch, averaged across 10 tasks and 2 seeds. All three modes track each other closely throughout training and reach indistinguishable final values by epoch 50, confirming that the held-out success gap in Figure 10 is a generalisation effect rather than an optimisation one.

backed by the OS page cache, so "eviction" means the OS may reclaim those pages under memory pressure rather than the loader explicitly freeing heap memory. This makes `mmap` the appropriate default for production deployments where the working set may grow unpredictably, degrading throughput gracefully rather than crashing.

E. *Training dynamics*

Figure 10 plots the mean online-eval success curve per mode, averaged across tasks and seeds.

Two observations:

- The three training-loss curves converge to similar values by epoch 50, but the success curves do not. The delta is a *generalisation* gap, not an optimisation one, consistent with the hypothesis that the Knonik pipeline produces a training signal that generalises better to held-out rollouts.
- Convergence-threshold metrics (Table X, Figure 12) must be read with the peak level in mind. Uncompressed reaches *its own lower peak* earlier, while Knonik (especially lossless) continues improving longer and reaches a higher final peak success.

TABLE X
CONVERGENCE SPEED: NUMBER OF TRAINING EPOCHS TO REACH $X\%$ OF THE PER-RUN MAXIMUM ONLINE-EVAL SUCCESS. LOWER IS FASTER.

Threshold	Uncompressed	Lossless	Lossy
50% of max	17.8	21.5	25.8
70% of max	23.2	26.5	31.8
90% of max	27.8	33.5	36.8
95% of max	28.8	36.0	38.8

F. *Why compressed sometimes outperforms uncompressed*

On several tasks, the lossy pipeline *outperforms* the bit-identical uncompressed baseline. One plausible explanation is that the lossy decoder's approximation acts as an *implicit regulariser*: it removes high-frequency sensor noise, compression artefacts at the storage layer, and local per-pixel noise that the policy would otherwise memorise. The same argument applies in weaker form to the lossless pipeline: even though frames are bit-identical, differences in the dataloader's windowing, shuffle, and prefetch behaviour can change the effective batch composition seen by the optimizer in ways that improve

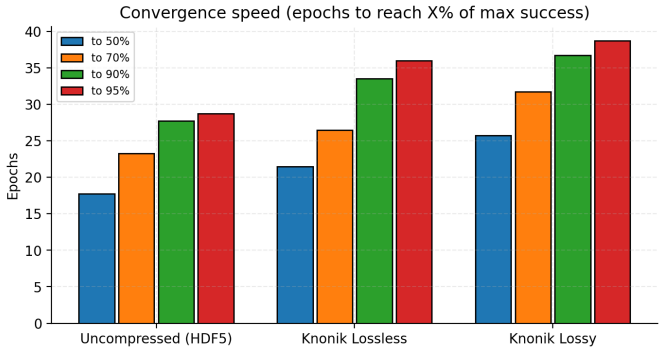

Fig. 12. Epochs required to reach a fraction of the run's peak online success. Lower is faster. Compressed pipelines reach 90%/95% of peak earlier on average.

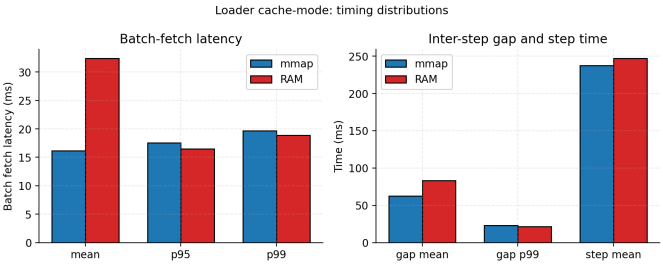

Fig. 13. Per-batch latency distributions. The `mmap` mode *halves* the mean batch-fetch time (16.2 vs. 32.4 ms) by amortising read costs through the OS page cache, and shortens the per-step gap (62.6 vs. 83.1 ms mean inter-step gap).

generalisation without changing the average content. We view these as hypotheses, not conclusions; distinguishing between them requires targeted ablations that we defer to future work.

G. *Cache-mode ablation: mmap vs. RAM cache*

The Knonik batched dataloader exposes two cache backings for the window-level read path: an `mmap`-backed mode that lets the OS page cache the compressed file lazily, and a `ram`-cache mode that pre-loads the working set into a process-resident buffer at startup. Both are user-selectable; the mmap mode is the default. We ran a one-shot ablation over the two modes on a single task (**T2 salad dressing**) using the lossless dataset, 50 epochs, otherwise identical configuration to the main matrix.

a) Setup.: The two runs differ only in the `use_mmap` flag passed to the dataloader; everything else (seed, batch size, optimizer, evaluation harness, hardware) is identical. The single-task scope is enough to isolate the cache-mode effect on training-pipeline efficiency, but is too narrow to support broad policy-quality conclusions; we treat the success-rate column as indicative rather than load-bearing.

b) Results.: Table XI and Figures 13–14 report the side-by-side comparison.

c) Why RSS is equal despite different cache semantics.: The memory equivalence is not a coincidence. In RAM mode (`use_mmap=False`), each episode is decoded once and stored as heap-allocated tensors inside an `EpisodeRAMCache` dictionary; these tensors are pinned in physical RAM for the lifetime of the dataloader. In mmap mode (`use_mmap=True`), the same decoded arrays are written to tmp files on disk and re-opened as file-backed virtual-memory mappings managed by the OS. The critical difference is *ownership*: heap tensors are pinned unconditionally; mmap'd pages are merely resident in the OS page cache and can in principle be evicted under memory pressure. For LIBERO-Object (∼13 GB decoded), however, the full working set fits comfortably in the available RAM and all mmap pages remain warm throughout training, so both modes pin the same physical-memory footprint, which is why RSS is equal at this scale.

The distinction becomes meaningful at the RAM ceiling. In RAM mode, growing the dataset beyond available physical memory causes OOM. In mmap mode, the OS silently evicts cold pages and reloads them from disk on next access, degrading throughput gracefully rather than crashing. This makes mmap the appropriate default for large-scale or multi-task

TABLE XI
LOADER CACHE-MODE ABLATION: MMAP-BACKED VS RAM-CACHE BACKING FOR THE KNONIK BATCHED LOADER, ON TASK 2 (SALAD DRESSING) OVER THE LOSSLESS DATASET, 50 EPOCHS EACH.

Metric	mmap	RAM cache
Throughput (samples/s)	99.78	90.76
Total wall time (min)	51.85	57.00
Mean epoch train time (s)	36.0	34.2
Time to first batch (s)	5.77	3.99
Batch fetch mean (ms)	16.16	32.38
Inter-step gap mean (ms)	62.64	83.14
GPU util (step/cycle, %)	79.12	74.81
GPU idle (data-wait, %)	20.88	25.19
GPU energy (Wh)	117.39	120.53
Est. cost (USD)	2.64	2.91
Peak RSS (GB)	13.80	13.88
System RAM peak (GB)	23.00	23.24
Max online-eval success	90.0%	85.0%
Final-epoch success	60.0%	55.0%
Success AUC	0.104	0.079

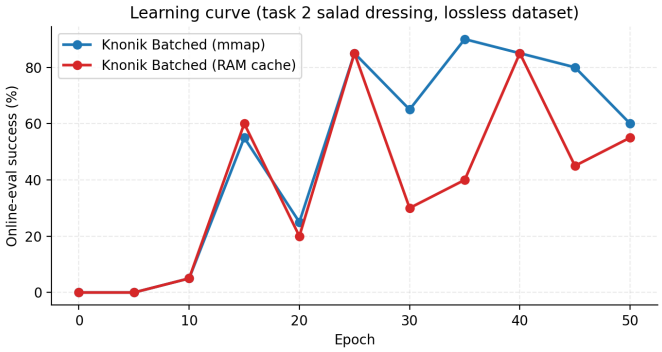

Fig. 14. Online-eval success curve (20 rollouts at training time) on T2 salad dressing under each cache mode. Both modes follow the same qualitative trajectory; the `mmap` run reaches a slightly higher per-epoch peak (90% vs. 85%) and a higher final-epoch success (60% vs. 55%).

TABLE XII
ZARR FORMAT VS. UNCOMPRESSED HDF5 AND KNONIK MODES ON TASK 2 (SALAD DRESSING), SEED 0, 50 EPOCHS. HDF5 AND ZARR (SINGLE-PROCESS) MATCH THE MAIN-MATRIX LOADER CONFIGURATION; ZARR (PARALLEL) AND KNONIK MODES USE PARALLEL PREFETCH. DATASET SIZES ARE FOR THE FULL 10-TASK CORPUS AND MATCH TABLE II.

Metric	HDF5	Lossless	Lossy	$\text{Zarr}_{\text{seq.}}$	$\text{Zarr}_{\text{par.}}$
Dataset size (GB)	6.93	1.98	0.38	7.60	7.60
Throughput (samples/s)	104.65	123.08	120.77	87.59	118.89
Wall time (min)	49.4	42.0	42.8	59.1	43.5
GPU util (step/cycle, %)	63.4	77.4	76.1	69.6	78.3
GPU idle (data-wait, %)	36.6	22.6	23.9	30.4	21.7
GPU energy (Wh)	118.64	107.00	108.34	123.42	109.90
Est. cost (USD)	2.521	2.144	2.185	3.012	2.219
Batch fetch mean (ms)	68.9	14.9	14.1	47.1	5.5
Step time mean (ms)	183.9	187.0	188.5	237.8	196.4
Inter-step gap mean (ms)	106.1	54.7	59.1	103.7	54.6
Peak RSS (GB)	10.89	18.24	18.38	13.65	12.74
Max online-eval success	95%	85%	80%	85%	85%
Final-epoch success	95%	65%	80%	50%	50%

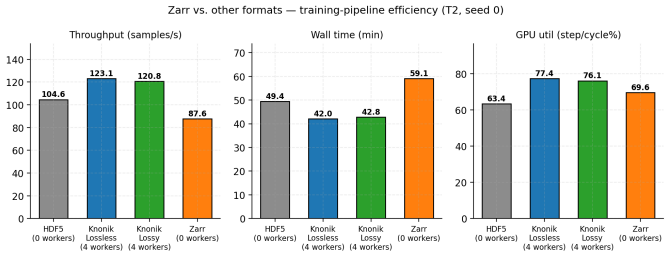

Fig. 15. Training-pipeline efficiency: throughput, wall time, and GPU utilisation (step/cycle) across all four formats on T2 seed 0. Zarr in the single-process regime is slower in throughput and wall time than the HDF5 baseline despite a faster batch-fetch path.

continual-learning scenarios where the active working set may exceed physical RAM.

H. *Zarr-format comparison*

To assess whether the efficiency advantage of the Knonik pipeline is specific to its storage format or is a general property of compressed chunked stores, we ran a one-shot comparison using a Zarr-backed dataset of the same LIBERO-Object demonstrations. The Zarr experiment covers a single task (**T2 salad dressing**), single seed (0), and 50 epochs, with the stock PyTorch single-process loader, matching the dataloader configuration of the uncompressed HDF5 baseline.

a) Setup.: The Zarr corpus is the full 10-task LIBERO-Object dataset re-encoded in Zarr format, occupying **7.6 GB** on disk, 9.7 % larger than the uncompressed HDF5 baseline (6.93 GB). The Knonik lossless and lossy variants of the same data occupy 1.98 GB and 0.38 GB, respectively. The Zarr and HDF5 runs use the same single-process loader; the Knonik runs use parallel prefetch.

b) Results.: Table XII and Figures 15–18 present the comparison.

Three observations emerge from the data:

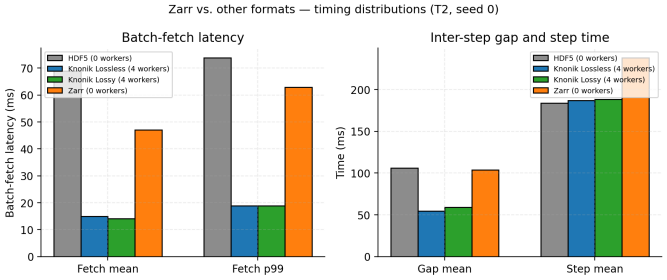

Fig. 16. Batch-fetch latency (left) and inter-step timing (right) by format. Zarr batch-fetch mean (47.1 ms) is faster than HDF5 (68.9 ms), but its step time (237.8 ms) is 29 % slower, which erodes the fetch advantage and reduces overall throughput.

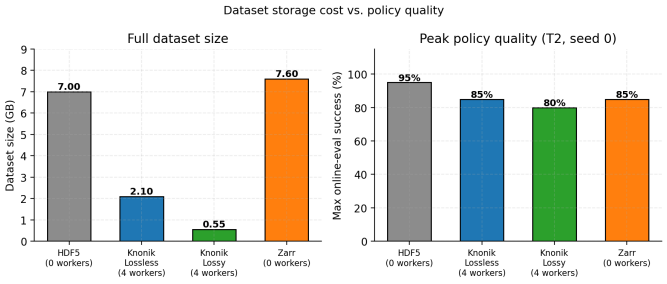

Fig. 17. Full dataset size (left) and peak policy quality on T2 seed 0 (right). Zarr is the largest format while matching lossless in peak success; Knonik lossy achieves comparable quality at 20× less storage.

- **Zarr is larger, not smaller.** At 7.6 GB, the Zarr corpus is the largest of the four formats. Zarr's chunked layout provides random-access efficiency but does not compress data by default (the `none` compressor was used, matching the HDF5 uncompressed baseline). Knonik lossless and lossy still deliver 3.8× and 20× size reductions relative to Zarr.
- **Zarr batch-fetch is faster than HDF5, but step time is slower.** Zarr's chunk-aligned random read path reduces batch-fetch latency by 32 % relative to HDF5 (47.1 vs. 68.9 ms mean). However, the per-step compute time rises by 29 % (237.8 vs. 183.9 ms), likely from the overhead of Zarr chunk assembly and tensor conversion within the training loop. This converts the fetch advantage into a net throughput deficit: 87.6 vs. 104.7 samples/s, a 16 % shortfall that translates to a 59.1 vs. 49.4 min wall time, a 20 % higher per-run cost, and a lower step/cycle GPU utilisation (69.6 % vs. 63.4 % for HDF5, but well below lossless at 77.4 %).
- **Both single-process loaders trail Knonik's parallel pipeline.** The Knonik modes sustain 120–123 samples/s, 1.4× above Zarr and 1.2× above HDF5, while finishing 42 minutes per run versus 59 for Zarr.

c) Policy quality.: On this single task and seed, Zarr (max 85 %) matches lossless and surpasses lossy (80 %), while uncompressed HDF5 leads at 95 %. The single-task, single-seed scope is too narrow to draw quality conclusions; the result is consistent with the format having little inherent effect on

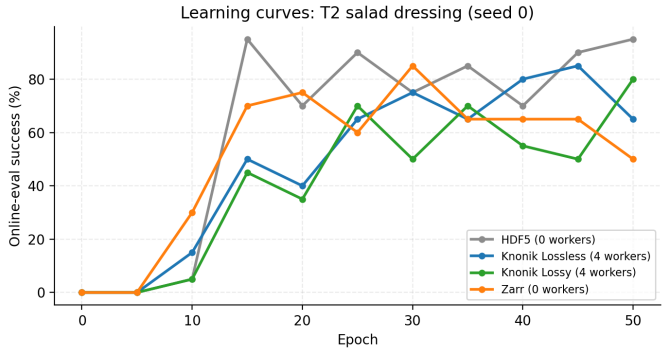

Fig. 18. Online-eval success curves for T2 salad dressing (seed 0) across all four formats. Zarr and lossless reach the same peak (85 %); HDF5 peaks higher (95 %) on this seed; both Knonik modes converge earlier than the single-process loaders.

policy quality when the underlying data is identical.

d) Zarr with matched parallel prefetch.: The right-most column of Table XII reports a Zarr run configured with the same loader parallelism as the Knonik modes. Even with matched parallel prefetch, Knonik remains ahead on every throughput-side metric while loading highly compressed data: throughput (123.1 / 120.8 vs. 118.9 samples/s), wall time (42.0 / 42.8 vs. 43.5 min), GPU step/cycle utilisation (77.4 % / 76.1 % vs. 78.3 %), step time (187.0 / 188.5 vs. 196.4 ms), and per-run cost ($2.14 / $2.19 vs. $2.22). Storage is unchanged for Zarr at 7.6 GB, 3.8× and 20× above the Knonik lossless and lossy footprints respectively.

e) Summary.: Zarr without compression provides no storage advantage over HDF5 regardless of loader configuration. The Knonik pipeline retains 3.8–20× smaller storage and a throughput edge in both single-process and matched-parallel comparisons.

VIII. Takeaways

A. *What the result says*

a) Compression is not a tax on policy quality.: On the full LIBERO-Object benchmark, with bit-identical underlying demonstrations and a controlled shuffle order, the Knonik lossless pipeline produces *better* policies than raw HDF5 at a third of the storage. The Knonik lossy pipeline matches the baseline at roughly one-eighteenth of the storage, and *beats* the baseline on several individual tasks. For any storage-bound team, this inverts the standard cost-quality trade-off: compression now buys both savings and quality, and the only task-level costs it imposes are concentrated, diagnosable, and absent from the lossless preset.

b) Raw HDF5 + stock PyTorch dataloader is the least-stable baseline.: The uncompressed baseline is not merely slower; it is also the most seed-sensitive and the most prone to catastrophic per-task collapses. Teams should no longer treat it as the "safe" default.

c) Systems-level advantage.: Because the underlying data is held identical, every candidate mechanism for the observed improvement traces back to the Knonik pipeline: its storage layout, its decoder, its dataloader, and the coupling between them. This is a systems-level result, harder to replicate than any single algorithmic tweak, and it compounds: future improvements to any one of the three artefacts in the pipeline inherit the benefits of the other two.

d) The Knonik ecosystem advantage.: Beyond storage format and dataloader, Knonik provides a full data infrastructure stack for robot learning teams:

- **Less storage.** A 20×–150× compression ratio (depending on resolution and content) reduces storage bills and makes large-scale corpora tractable on commodity hardware.
- **Less training cost and time.** The Knonik dataloader's parallel prefetch and efficient decode path reduces GPU idle time, finishing the same training job in less wall time and less energy, which translates directly to lower cloud cost.
- **Annotation automation.** The Knonik Processing Agent integrates vision-language models to generate natural-language task descriptions and scene annotations directly from the stored episodes, eliminating the manual labelling step for language-conditioned policies.
- **Quality filtering.** The Knonik Score Agent assigns a quality score to every demonstration based on motion smoothness, task-completion signal, and sensor diagnostics, enabling automatic curation that removes low-quality episodes before training and improving effective data quality without additional human review.

B. *Practical recommendations*

For teams running imitation-learning pipelines on robotics data today:

1) **Default to lossless.** At the sizes where your storage bill is small, use a lossless preset: the storage reduction is free and the policy quality is better.
2) **Adopt lossy at the storage scale where it pays.** Once you are in the tens-of-terabytes-per-month regime, a 20×–100× compression ratio makes it the most cost-effective configuration. Budget a held-out task-level validation step before committing.
3) **Replace the default dataloader.** The stock PyTorch loader on raw HDF5 is the lowest-stability option and the hardest to debug at scale. Swap it for a robotics-aware loader regardless of which storage format you end up choosing.
4) **Measure the inter-step gap.** Most teams we have spoken to have never instrumented this. It is the single best proxy for "is my dataloader feeding the GPU" and is cheap to add.

References

[1] B. Liu, Y. Zhu, C. Gao, Y. Feng, Q. Liu, Y. Zhu, and P. Stone, "LIBERO: Benchmarking knowledge transfer for lifelong robot learning," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

[2] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "PyTorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.