When the Scan and Step Paths Diverge: A Reproducibility and Correctness Audit of a Full-History Mamba VLA (MTIL)

Anonymous authors
Paper under double-blind review

Abstract

MTIL (Zhou et al., 2025) is a full-history Mamba-2 vision–language–action (VLA) policy that advances two premises for state-space backbones in robot imitation: that encoding the *entire* interaction history improves closed-loop success, and that a recurrent state-space update deploys at constant per-step cost. We reproduce it on the ALOHA benchmark under a fixed, seed-controlled protocol and audit both premises, contributing three corrections. The first (C1) is a correctness defect: the recurrent `step()` controller shipped in the released code is broken. It collapses the canonical two-stream residual of its blocks and diverges from the sequence-parallel training path by ≈18 in action units (max$|\Delta|$ 17.9–19.7 across 8 sampled episodes), driving closed-loop success, on our sequence-parallel-trained reproduction, to **0.7%** (1/150) when it is used. (Our 97.3% reproduction is trained via the repository's parallel training script, `forward_sequence`; the repository separately ships an older training script, present since its initial commit, that trains *through* this exact `step()` frame-by-frame, so `step()` is self-consistent with, and does not corrupt, that older recipe's own numbers. What is broken is `step()` relative to canonical Mamba-2 block semantics and relative to any model trained via the newer, parallel recipe, the deployment scenario the advertised *constant-cost recurrent* controller, the efficiency claim's deliverable, is meant to serve.) A ∼15-line fix restores a controller that matches the parallel path to $\leq 1.2 \times 10^{-3}$ (in fp32 and fp16, below the model's 0.012 ground-truth floor) and recovers rollout success to **96.7%** (3×50 seeded, Wilson-95% [92.4, 98.6]), statistically indistinguishable from the 97.3% sequence-parallel reproduction. We verify the defect on a *pristine* clone of the published repository.

The verification gap is not unique to MTIL. By source inspection plus one empirical second-repo kill-test, among the four released SSM robot policies we examined, none ships a recurrent rollout path that is verified equivalent to its parallel scan (only two ship a recurrent path at all; RoboSSM's is unimplemented dead code that raises `AttributeError` on first use). Across this line, the advertised linear-time recurrent inference is either unrealized or unverified.

The second correction (C2) concerns efficiency: with a faithful controller in hand, the constant-per-step advantage is real at the block level (∼1.4× on the Mamba blocks) but invisible end-to-end (∼1.014×, flat in T), because MTIL's per-frame, uncached DINOv2 vision frontend dominates the SSM scan by more than an order of magnitude (≈27× the $O(T)$ step block, ≈19× the $O(T^2)$ re-encode). This is a property of *this* encoder, not of VLAs in general.

The third correction (C3) is narrower in scope: on MTIL's near-Markovian headline benchmark, a 16-frame context recovers most of the full-history performance. A 16-frame-context-trained policy solves transfer_cube at 90% (a Markovian control, 3×50 seeded, Wilson-95% [84.2, 93.8]) against the full-history model's 96.7%; a benchmark solvable at that level by a short-context policy cannot demonstrate that unbounded history is required. Two further probes target genuinely memory-dependent settings but remain inconclusive rather than extending this claim: an exploratory LIBERO-Mem comparison (n=20) shows no *detectable* memory gain, though both policies floor on basic manipulation before memory becomes relevant; and a delayed-cue probe we could not make conclusive (the cued task floors at 25%). We do not conclude that full history is generally unimportant for memory-dependent

manipulation, only that a short context already recovers most of the achievable success on the benchmark MTIL's headline number comes from (cf. DSSP, which shows history helping on tasks designed to require it). A same-protocol ACT baseline reproduces at only ∼25%, which we report as an under-tuned outlier and from which we draw no vision-attribution. None of these findings refute MTIL's utility; they correct *why* it works, and point SSM-VLA efficiency effort, for heavy-encoder designs, at the visual frontend rather than the sequence model. We release a one-command artifact for the central C1 correctness finding (C2/C3 are backed by committed result-dumps and scripts) on a pristine clone of the published code.

1 Introduction

Vision–language–action (VLA) policies trained by imitation now span transformer, diffusion, and state-space backbones. The state-space (Mamba) family is promoted with a specific dual promise. The first half is *accuracy*: because a selective SSM can summarize an unbounded past into a fixed-size recurrent state, a "full-history" policy is argued to capture temporal dependencies that fixed-context or single-frame policies miss. The second half is *efficiency*: that same recurrence deploys at O(1) cost per control step (a constant-size state advanced once per timestep), in contrast to attention policies whose per-step cost grows with the horizon.

MTIL (Zhou et al., 2025) is a prominent realization of this dual promise: a full-history Mamba-2 policy with a frozen DINOv2 (Oquab et al., 2024) visual frontend, image–state cross-attention, and elastic-weight-consolidation lifelong learning, reporting near-perfect success on ALOHA `transfer_cube` and a recurrent controller (its Algorithm 2) intended for constant-cost deployment. MTIL is a natural object of study precisely because it states both halves of the promise explicitly and ships code.

This paper is a reproducibility-and-correctness audit, not a new method. We train no new architecture: all training reproduces the released MTIL recipe or produces small, controlled baselines (ACT, windowed-trained variants) for comparison. The project is student-scale, conducted on shared GPU hosts (a second, independent host is used only for the C2 cross-hardware replication of § 7). We ask whether MTIL's two premises hold *as released and as benchmarked*, and we answer conservatively: where the evidence supports only a weaker claim than a tempting headline, we make the weaker claim. The result is three corrections.

The paper's contribution is evidence quality rather than a new method. The load-bearing claims, namely the reproduction (Table 1, #1) and the C1 closed-loop results (#3–4), are seed-controlled and reported with a Wilson interval (Wilson, 1927); the C1 numeric kill-test (#5) is a deterministic, code-level measurement (per-frame divergence against a fixed reference) rather than a statistical claim, so it carries no seed or interval by construction; the central finding reproduces from a one-command artifact on a *pristine* clone of the published code; and the cross-repository pattern (§ 6.2) gives the result audience interest beyond the single repository. Supporting and exploratory results (C2, C3) are held to a visibly lower bar and labelled as such in Table 1, so a reader can weight each claim by the evidence behind it.

The corrections are not fully independent, but only one of them depends on the other. C1's fix, that the released recurrent controller is broken, is the prerequisite for C2: one cannot honestly profile an $O(T)$ inference path on a controller that does not match training. C3 does not share this dependency: both the windowed and full-history conditions of § 8 are evaluated on `forward_sequence`, the sequence-parallel path that C1's fix never touches and that was correct throughout (Table 8). We include C3 in the same audit because it examines the same reproduced, trusted pipeline (§ 4), not because the C1 fix enables it.

Contributions.

1. A certified reproduction of MTIL on ALOHA (`transfer_cube` 97.3%±2.3% and bimanual `insertion` 40.7%±6.1%, sample SD) under a seed-controlled, Wilson-CI protocol with reward-ladder reporting, plus a same-protocol ACT baseline.
2. **C1**: a diagnosis and fix of MTIL's broken recurrent controller, verified on a *pristine* clone of the published code and reported with a seeded confidence interval (0.7% → 96.7%); generalized, by source inspection plus an empirical second-repo kill-test, into a cross-repository pattern: no released

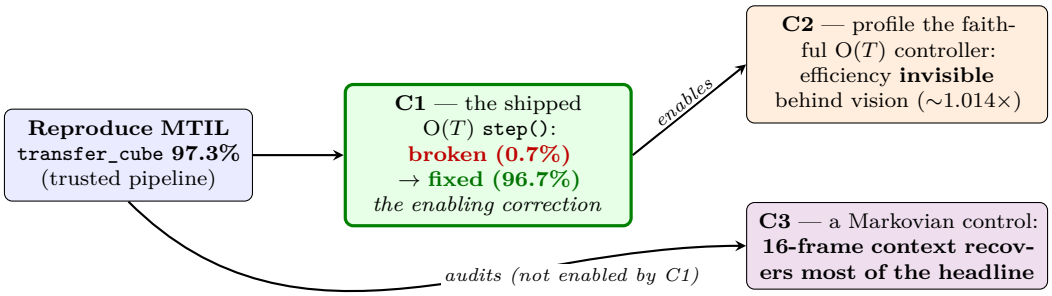

Figure 1: **The audit spine.** The C1 box's 0.7%→96.7% is on *our* `forward_sequence`-trained checkpoint (§ 3, § 6.1), not a claim about MTIL's own published numbers. C1 is the *prerequisite* for C2 only: one cannot honestly profile a controller (the O(T) `step()`) that does not reproduce the trained policy. C3 does not depend on the C1 fix: both its windowed and full-history conditions are evaluated on `forward_sequence`, the sequence-parallel path that was never broken (Table 8); it audits the pipeline the reproduction already certified, not a product of the C1 fix. The reproduction certifies the pipeline; the C1 fix makes the O(T) controller faithful for C2's profiling; C3 separately audits the full-history premise on the pipeline § 4 certified.

SSM robot policy we examined both deploys *and* verifies a recurrent rollout path (RoboSSM's is unimplemented dead code that raises `AttributeError` on first use; MTIL's is the only one deployed, and it is broken).

3. **C2**: a measurement that the O(T) advantage is real at the block level but practically invisible behind the vision frontend, with a precise, encoder-specific framing that reconciles two contradictory prior profiling results.

4. **C3**: on MTIL's near-Markovian headline benchmark, a 16-frame context recovers most of the full-history performance (90.0% vs. 96.7%, a Markovian negative control, § 8.1); a benchmark solvable at that level by a short-context policy cannot demonstrate that unbounded history is required. Two further probes at genuinely memory-dependent settings, an exploratory non-Markovian comparison and a controlled delayed-cue probe (§ 8.2–8.3), remain inconclusive and do not extend this claim to memory-dependent manipulation generally.

5. A one-command C1 reproduction artifact (pristine clone, numeric kill-test, closed-loop test, fix patch); C2/C3 are backed by committed result-dumps + scripts, not one-command from the base bundle.

2 Related Work

Imitation policies for ALOHA. ACT (Zhao et al., 2023) is the reference behaviour-cloning policy for the ALOHA bimanual platform: a conditional-VAE transformer that predicts an action *chunk* and executes with temporal ensembling, using a ResNet-18 visual backbone trained from scratch. It is the baseline against which MTIL positions itself, and our same-protocol control (§ 5). MTIL replaces ACT's transformer with a full-history Mamba-2 backbone and a frozen, *pretrained* DINOv2 ViT-L/14 frontend.

State-space models and the scan–step duality. Mamba (Gu & Dao, 2023) and the Mamba-2 structured state-space duality (Dao & Gu, 2024) establish that a selective SSM admits *both* a chunked parallel scan (used in training) and an exact sequential recurrence (used at inference), and that the two are mathematically equivalent. This duality is the theoretical foundation of C1: an exact O(T) recurrent equivalent of the trained model is *guaranteed to exist*, though the duality does not guarantee that any given implementation realizes it. The generic SSM-versus-attention efficiency crossover (linear versus quadratic in sequence length) is textbook from these works; we therefore do not claim it.

Efficient SSM robot policies. A line of work (MaIL (Jia et al., 2024), RoboSSM (Yoo et al., 2025), RoboMamba (Liu et al., 2024)) advertises linear-time SSM inference for robot control. Of these, only RoboSSM ships a recurrent decode path at all (MaIL re-stacks a fixed window each step; RoboMamba's action head is a single parallel forward, its recurrence serving text autoregression only). As § 6.2 documents, none deploys *and* verifies a recurrent rollout path; our efficiency finding (C2) is positioned against this line rather than as an addition to it.

Concurrent full-history-Mamba work (DSSP). DSSP (Guan et al., 2026) is the closest concurrent work: it independently reports a full-history-Mamba windowed-versus-full history ablation and an $O(T)$ efficiency story. Its history ablation (Mamba encoder) shows a monotone gain: short window (Th=10) 53.0%, medium (Th=20) 60.0%, full history 71.33%. Its inference is efficient (full-history Mamba 1.97 ms p95, 238 MB peak; 45% faster and 59% smaller than a full-history Transformer), with per-step latency *nearly independent of accumulated history length*. We treat DSSP as corroboration on two axes, showing that an SSM scan does not blow up with history and that full history can help on tasks designed for it, while noting three things it does not do: audit MTIL's released code, break down vision-versus-SSM cost (its encoder is light, so its $O(T)$ win is *visible*, the inverse of our encoder-gated result), or supply a Markovian negative control of the kind our § 8.1 provides.

VLA inference profiling. VLA latency profiling is configuration-specific and, across papers, contradictory. VLA-Perf (Jiang et al., 2026) measures, for a π_0-class transformer VLA (Black et al., 2024), that the language backbone dominates (≈39–64% of latency across datacenter, consumer, and edge GPUs) while the visual encoder is the *smallest* component (≈11–13%). This is the *opposite* ordering to ours, because π_0 amortizes a lightweight tokenizer behind a heavy language head. Our C2 (§ 7) is therefore not a generic "vision dominates" claim and does not rest on prior profiling as corroboration; it is specific to MTIL's frontend, which runs a frozen DINOv2 ViT-L/14 *per frame, uncached*. We add an encoder-specific data point to a landscape where the dominant term depends on the architecture.

Memory benchmarks and causal confusion. LIBERO-Mem (Liu et al., 2023) is our non-Markovian testbed (§ 8.2): its tasks enforce object aliasing and occlusion, and its published baselines place plain short-context policies near the floor. The interpretation of our Markovian control (§ 8.1), that a full-history policy's collapse under context truncation reflects *learned dependence* rather than task necessity, is an instance of causal confusion in imitation learning.

Positioning. The 2026 SSM/memory-VLA literature has moved quickly, and most would-be novelty is pre-empted (DSSP, the efficient-SSM-policy line, VLA-Perf). The contribution that survives this crowding is a correctness audit no prior work performs: MTIL's released recurrent inference is broken (C1), and, across the four released SSM robot policies we examined, none both deploys and verifies a recurrent rollout path (§ 6.2).

3 Setup and Protocol

Architecture (reproduced unchanged). We use the released MTIL pipeline: each frame's RGB image is encoded by a frozen DINOv2 ViT-L/14 (layer −4, feature map 1024×35×46), reduced by a learned spatial adapter, fused with the 14-dimensional proprioceptive state by image–state cross-attention, and processed by a stack of four Mamba-2 blocks over the whole episode (`forward_sequence`); a head emits a 16-step action chunk per frame, executed with ACT-style temporal aggregation. Training uses 50 scripted demonstrations per task (40 train / 10 held out), DINOv2 features cached once, batch size 1, AdamW with cosine schedule, `precision=16-mixed`, 150 epochs. The repository ships *two* training scripts: an older `train.py`, present since the repository's initial code-release commit, whose training loop calls `step()` directly (`train.py:180`, also :244 for validation), frame-by-frame, with truncated backpropagation (hidden states detached after each call, `train.py:183-184`); and a newer `train_par.py`, added later per the repository's own changelog, whose training loop calls `forward_sequence` (`train_par.py:158`), the chunked-parallel path. We train via `train_par.py`: our reproduction is fit to `forward_sequence`, not to `step()` (see § 6.1 for why this matters for C1). Two environment notes matter for faithful reproduction: the chunked-scan kernel requires fp16 on Turing GPUs (it raises an indexing error in fp32), and DINOv2 is frozen and dropped from the checkpoint

after training, so evaluation rebuilds it from a model hub. The single correctness change we make to the released code, the `step()` fix of § 6, does not affect the `forward_sequence` numbers in § 4.

Evaluation protocol. We evaluate closed-loop in the dm_control ALOHA environment with headless EGL rendering. The decisive metric is task success, defined as reaching the environment's maximum reward (4: the cube is transferred / the peg is inserted); we never use action MSE, which we show repeatedly fails to predict closed-loop success. For each task we run three seeds × 50 rollouts = 150, reporting the per-seed rate, the seed mean ± sample standard deviation, the pooled Wilson 95% confidence interval, and the reward ladder (the fraction reaching each intermediate reward). (Pooling 150 rollouts as i.i.d. for the Wilson interval is mildly anti-conservative under between-seed clustering, since rollouts share a checkpoint and training seed and are not fully independent replicates. The true unit of replication is the seed, of which we have only three per condition. As a companion, we report a seed-level 95% interval from a t-distribution on the three per-seed rates ($t_{0.975,\,\mathrm{df}=2} = 4.30$); this is a minimal-assumption estimate of between-seed variability, not a claim of normality at $n = 3$, and it is intentionally much wider than the pooled Wilson interval. It is a poor approximation when a metric sits at or near 0% or 100% (as when two of three seeds are identical), which we flag where it applies; where the raw interval falls outside [0,100]%, we report it intersected with that range. We report both views wherever a seeded comparison is load-bearing, rather than relying on the pooled statistic alone.) Checkpoints are selected by lowest held-out loss. We treat the reward ladder, not the binary endpoint alone, as the informative signal about *where* a policy fails. A rollout in which a policy's actions drive the simulator into an invalid physics state is counted as a failure (this occurs only for the broken `step()`; see § 6.1).

Evidence-strength summary. Table 1 is the results-at-a-glance ledger: every load-bearing claim is either held to the full seeded-Wilson bar or, where the measurement is deterministic rather than statistical (the C1 numeric kill-test, #5), reported as such explicitly; supporting and exploratory results are labelled explicitly so their weight is transparent rather than implied.

The two corrections that carry the certification, the reproduction (#1) and C1's closed-loop results (#3–4), are seeded and interval-reported; C1's numeric kill-test (#5) is a separate, deterministic, non-seeded measurement (§ 3); the C3 evidence (#9–11) is supporting and is scoped as such throughout.

4 A Certified Reproduction

4.1 `transfer_cube`: reproduced

Cube transfer is reproduced: 97.3% against the paper's ≈100%, with a tight interval and a 100% touch rate. Every rollout reaches the cube; the rare misses fail only at the final lift/transfer. This certifies that the feature extractor, fusion, Mamba-2 backbone, action-chunk head, and temporal aggregation are all working as intended, and gives us a trusted pipeline to audit in § 6–§ 8.

4.2 `insertion`: a partial reproduction on a protocol-sensitive benchmark

Bimanual insertion reaches 40.7% against the paper's 84%, a ∼40-point gap. We report it as a partial reproduction on a high-variance, protocol-sensitive benchmark, for three reasons. First, the number is mid-pack, not broken: published insertion success spans 9–90% with no canonical value across methods and reproductions, and our value sits at the median of the honest-reproduction cluster, 3–5× above the floor. Second, the reward ladder (pooled over the same seeded 150 rollouts) shows the largest single loss is at alignment, not the final insertion step (touch 90.0 → lift 79.3 → align 51.3 → insert 40.7: a 28-point drop at align vs 11 at touch→lift and 11 at align→insert); about a fifth of rollouts (31/150) never reach lift, so the failure is not confined to millimetre-scale insertion alone, but the dominant loss is still late (align and beyond), consistent with a precision deficit rather than an early perception failure; the same cube→insertion drop occurs for every method. Third, our insertion code is nearly identical to the released code (chunk 16, MSE, no CVAE), so we reproduce their *code* faithfully; the 84% is most consistent with a recipe absent from the release. As a control, retraining insertion on all 50 demonstrations rather than 40 left success unchanged (∼44%), ruling out demonstration count as the gap's cause.

Table 1: **Results at a glance.** Point estimate, sample size, whether seeded, the 95% interval, and the evidence strength of each claim. Rows 3–4 (C1) are on *our* `forward_sequence`-trained checkpoint (§ 3, § 6.1); MTIL's own published numbers are not shown to have used this defective path.

#	Claim (section)	Result (point estimate)	n	Seeded	Wilson-95% / spread	Strength
1	Reproduction `transfer_cube` (§ 4.1)	**97.3%** success	3×50	yes	[93.3, 99.0]	**load-bearing**
2	Reproduction `insertion` (§ 4.2)	**40.7%** success	3×50	yes	[33.1, 48.7]	supporting (partial repro)
3	**C1** fixed `step()` closed-loop (§ 6.1)	**96.7%** success	3×50	yes	[92.4, 98.6]	**load-bearing**
4	**C1** pristine `step()` closed-loop (§ 6.1)	**0.7%** success	3×50	yes	1/150 [0.1, 3.7]	**load-bearing**
5	**C1** numeric kill-test (§ 6.1)	pristine **17.9**–**19.7** vs fixed $\leq 1.2 \times 10^{-3}$ max$\lvert\Delta\rvert$	8 episodes	—	deterministic (8 episodes tested)	**load-bearing**
6	**§ 6.2** RoboSSM recurrent decode	`AttributeError` (unimplemented dead code)	code + exec	—	reproduced (exec, Part A)	supporting (cross-repo)
7	ACT baseline (§ 5)	∼**25%** success (under-tuned)	6 ckpts×50	—	range 14–30%	supporting (control)
8	**C2** latency / speedup (§ 7)	vision **189 ms** vs block **7 ms** (O(T) step) / **9.8 ms** (O(T^2) re-encode); speedup ∼**1.014**×	20/pt × 7 T	no	flat in T	supporting
9	**C3** k=16 windowed-trained (§ 8.1)	**90.0%** success	3×50	yes	[84.2, 93.8]	supporting
10	**C3** LIBERO-Mem (§ 8.2)	full ≤ windowed (no memory gain)	1×20 / task	no	**exploratory**	exploratory
11	**C3** delayed-cue probe (§ 8.3)	follow **68.8%** (11/16), flip 37.5% (3/8)	8 pairs	no	**exploratory**	exploratory

Table 2: `transfer_cube` reproduction summary.

	value
Per-seed success (seeds 0/1/2)	100% / 96% / 96%
Seed mean ± SD	**97.3% ± 2.3%**
Seed-level 95% CI (t, df=2)	[91.6%, 100.0%]
Pooled (146/150), Wilson-95%	**[93.3%, 99.0%]**
Avg return	609.9
Reward ladder (pooled, 150)	touch (≥1) **100%** · **transfer (=4) 97.3%** (the few misses fail only at the final lift/transfer)

The contrast between § 4.1 and § 4.2 is itself a finding: a single insertion success rate is uninterpretable without a fixed protocol and a confidence interval, since our own per-seed rates span 34–46% on identical weights. This protocol discipline is what licenses the claims of § 6–§ 8.

Table 3: `insertion` reproduction summary.

	value
Per-seed success (seeds 0/1/2)	42% / 34% / 46%
Seed mean ± SD	**40.7% ± 6.1%**
Seed-level 95% CI (t, df=2)	[25.5%, 55.8%]
Pooled (61/150), Wilson-95%	**[33.1%, 48.7%]**
Reward ladder (pooled, seeded, n=150)	touch (≥1) **90.0%** · lift (≥2) **79.3%** · align (≥3) **51.3%** · **insert (=4) 40.7%**

5 A Same-Protocol ACT Baseline

Because a single same-method success rate is uninterpretable without a fixed protocol (§ 4.2), we run ACT (Zhao et al., 2023) as a same-protocol *control* that makes that point concrete, not to crown a head-to-head winner. We train ACT under its canonical recipe (kl-weight 10, chunk 100, hidden 512, feed-forward 3200, batch 8, lr 1e-5, 2000 epochs, temporal aggregation) on the identical `transfer_cube` demonstrations MTIL used, and evaluate it in the same simulator with the same cube distribution.

On our reproduction ACT reaches only ∼**25%** (a representative upper-cluster value; 14–30% across six checkpoints), consistently across both 40- and 50-demonstration training, so this is neither a checkpoint-selection artifact nor a demonstration-count effect. Inspecting the rollout videos, the ACT policy reaches the cube and attempts the grasp but misses it (a grasp-precision deficit), then fails to lift and transfer. We verified that the obvious confound is absent: the training and evaluation cube poses are drawn from the identical distribution (`sample_box_pose`, x∈[0,0.2], y∈[0.4,0.6]) by both the demonstration collector and the evaluator, so there is no train/test distribution shift.

Our ∼25% is an under-tuned outlier, not ACT's true performance, and we report it as such. The MTIL paper's own Table 1 reports ACT at 90% on cube transfer and 50% on insertion (vs MTIL 100% / 84%), and the ACT literature places transfer_cube near ≈86%, all with the *same* from-scratch ResNet-18 vision. We used ACT's canonical recipe untuned and did not chase its number; the ≈65-point shortfall against the published ACT figure is most likely a recipe/optimization gap in our run. Two consequences follow. First, the authoritative head-to-head is the MTIL paper's own: MTIL 100% vs ACT 90% on cube transfer, a *modest* ∼10-point advantage, far smaller than our reproduction's 97-vs-25. Second, and importantly, this means we cannot attribute MTIL's edge to its pretrained vision frontend from the success comparison: a *from-scratch* ResNet-18 (ACT) reaches 90% on this task in the published comparison, so vision pretraining is not the success bottleneck here. We therefore make no success-attribution claim from the ACT gap; the only frontend claim we defend is about *latency* (§ 7), which is independent of success. The value of our ACT run is methodological: it re-confirms (§ 4.2) that a single same-method success rate is uninterpretable without a fixed protocol, here spanning 25% (our untuned reproduction) to 90% (the MTIL paper's own ACT) on the same task, a cross-source span rather than a within-run one.

6 C1 — The Released Recurrent Controller Is Broken

6.1 The bug, the fix, and a pristine-clone verification

MTIL's efficiency claim rests entirely on its recurrent `step()` (Algorithm 2). We find the *released* `step()` does not reproduce `forward_sequence`. Each of the four blocks is a canonical pre-norm two-stream block carrying a hidden stream `h` and a residual stream `r`; with an MLP sublayer a single block computes `r` ← `m + r` (post-mixer residual), then `h` ← $\texttt{MLP(LayerNorm}_2\texttt{(r))}$, returning the two streams *separately*. The released decode loop instead collapses both streams: it sets `hidden_out` ← $\texttt{MLP(LayerNorm}_2\texttt{(m+r)) + (m+r)}$ and then `hidden = residual = hidden_out`, so the next block computes `r` ← `r + h = 2(m+r)`, double-

Table 4: Pristine vs fixed `step()`: numeric and closed-loop comparison, on our `forward_sequence`-trained checkpoint (§ 3).

Level	Pristine / published `step()`	Fixed `step()` (ours)
Numeric — per-frame max|Δ| vs `forward_sequence` (8 sampled train episodes, N=24 frames/episode)	**17.9–19.7** (mean 18.7), fp32 **and** fp16 nearly identical	$\leq 1.2 \times 10^{-3}$ across the same 8 episodes ($<$ the model's 0.012 ground-truth error floor)
Closed-loop — `transfer_cube`, 3×50 seeded, Wilson-95%	**0.7%** (1/150; per-seed successes 1/0/0 of 50)	**96.7%** (145/150, [92.4, 98.6]; per-seed 48/49/48 of 50)

counting the residual; the error compounds across the four blocks, and the final projection is applied to the merged tensor rather than the bare hidden stream. The SSM mixer itself is untouched and correct; the defect is purely the inter-block residual bookkeeping.

This defect is not self-inconsistent for every released recipe. The older `train.py` (§ 3) trains *through* this exact `step()`, frame-by-frame with truncated backpropagation, so the forward recurrent states a model trained that way sees during training are exactly those `step()` produces at evaluation; detaching hidden states between steps changes only the gradient path, not those forward values. This is narrower than a claim of literal train/eval identity (training additionally injects proprioceptive noise, `train.py:101`, and dropout differs between train and eval mode), but it is enough to establish that this recipe has no `forward_sequence`-versus-`step()` architectural mismatch of the kind C1 describes. The two `mamba_policy` implementations' `step()` functions are themselves byte-identical (verified by direct comparison of `train/mamba_policy.py` and `train/mamba_policy_par.py`), so this is not a difference introduced by the later, parallel-training refactor; the same code has been present since the repository's initial code-release commit. What the defect breaks is `step()`'s use against a model trained via the *other* released recipe, `forward_sequence` (`train_par.py`), which is the recipe we use (§ 3). We therefore do not claim that MTIL's own published headline numbers are corrupted by this defect; they may well derive from the older, `step()`-consistent recipe, for which no such corruption would occur. Our claim is narrower and, we think, still consequential: the repository's two released training recipes are not currently interchangeable at deployment through the shared `step()` controller, and this specific incompatibility is what drives closed-loop success to 0.7% for *our* `forward_sequence`-trained checkpoint (§ 3), the newer recipe the repository advertises as faster, though its own usage instructions still direct a user to `train.py`.

The fix restores the two-stream semantics (`r` $\leftarrow$ `m + r`; `h` $\leftarrow$ `MLP(LayerNorm`$_2$`(r))`, kept separate). It is about 15 lines, adds no new parameters, and leaves the parameter names unchanged, so any checkpoint's weights load directly without a name-mapping shim; this loadability is mechanical, not a claim that behavior is preserved for a checkpoint trained via the older, `step()`-consistent recipe (§ 6.1), for which the fix would change, not restore, the deployed behavior. We verify on the unmodified published code: our audit harness loads the *pristine* `MambaPolicy` (a clean clone of the upstream repository, byte-identical to its published HEAD) and our *fixed* one side-by-side from one checkpoint.

The error is *structural*, not an exposure or state-accumulation effect: it is large from the very first frame, where no history is yet involved (per-episode mean 14.2–17.3, worst-frame max 17.9–19.7, across the same 8 episodes), not only at some later, history-dependent frame. The fixed controller's seeded 96.7% is statistically indistinguishable from the 97.3% sequence-parallel reproduction (§ 4.1), confirming that the corrected `step()` is a faithful $O(T)$ realization of the trained policy, while the broken one is unusable. The single pristine "success" (1/150) is a fluke (garbage actions that happened to nudge the cube home), and on 3/150 poses those actions drove the simulator into an *invalid physics state* (`mjWARN_BADQACC`), which we count as failures; the controller is non-functional either way. As a seed-level companion (§ 3): the fixed controller's three seeds (48/49/48 of 50) give a t-interval of [93.8%, 99.5%]. The pristine controller's seeds (1/0/0 of 50) are a

degenerate case for this approximation, two of three seeds are exactly zero, so we do not read a t-interval on it as informative; the pooled count (1/150) and the numeric kill-test are the more reliable evidence there.

A reproduction pitfall. The repository ships two copies of the policy module (training imports one, evaluation the other); patching only one leaves the controller broken at evaluation time. We flag this because silently duplicated source is an easily-overlooked hazard when fixing released code. (Independently, we note that *shared* environment assets are the dual hazard: an unrelated experiment that edits a benchmark's scene file can silently degrade an otherwise-correct evaluation, a lesson our own seeded re-run surfaced and one that motivates the seed-controlled protocol of § 3.)

Responsible disclosure and repository provenance. All C1 *experimental* claims (§ 4–§ 8) are pinned to `9bea607` (2026-04-02), the commit we cloned; the remainder of this paragraph instead describes the live repository's state as of a separate later check and is not a claim about the audited code. On whether the defect belongs to the code contemporaneous with MTIL's original publication or only to a later refactor: the identical residual-collapse `step()` is also present, byte-for-byte, in `train/mamba_policy.py` (the original, non-parallelized policy file), unmodified in the public history since commit `77eb0c9` (2025-05-12), its only edit to date; the parallel-training variant we audited (`train/mamba_policy_par.py`) was added later, the same day as `9bea607`, but reproduces the identical bug. The defect therefore predates, and does not depend on, that later refactor. We have not filed a public issue or pull request. We did privately disclose this finding to the original authors (without revealing our identity to reviewers, so as to preserve double-blind review) and received a substantive reply: they confirmed the residual-collapse defect, and clarified that the older `train.py` recipe trains through this exact `step()` (making it self-consistent for that recipe, per the discussion above), while stating that the parallel-training (`_par`) files were intended to correct this at the time they were introduced. Our own reading of the `_par` files as released, however, does not support that: `train/mamba_policy_par.py`'s `step()` is byte-identical to the older file's (above), unfixed as of the commit we cloned and, per the check below, still unfixed in `train/` as of the most recent commit we inspected. We offered our fix and will continue this exchange, and de-anonymize to them in full, at the point of de-anonymization. As of our latest check (2026-07-22), none of the repository's three open issues describe this defect. In the course of that check we also found that upstream has since partially addressed it, in two commits on 2026-06-24: `fe605a9` adds a new file with a corrected `step()` that keeps the two-stream residual separate, matching our fix (its docstring reads, our translation, "corrected single-step inference function, strictly aligned with the Mamba block's residual logic"; an inline comment on the fixed line, since removed, read "add only here! the previous code added it again at the end of the layer, causing a double explosion", an independent restatement of the exact defect above); minutes later, `ca07ddc` renames that file to `test/mamba_policy_par.py`, the path `test/inference_par.py` imports at deployment, and cleans up the comments. Neither commit message mentions the defect (both read as routine file/comment edits); the diagnostic language we quote is source-code commentary, not a commit message. No issue or pull request is linked to either commit, so we cannot attribute the fix to our audit or to independent discovery, and we do not speculate further on its provenance. The file the repository's own parallel evaluator actually imports was therefore fixed first. `train/mamba_policy_par.py`, which defines the class `train_par.py` trains (though `train_par.py` itself never calls that file's `step()`, only `forward_sequence`), and the legacy pair `train/mamba_policy.py` plus its test-side copy, remained unmodified at that point and still carried the original residual-collapse `step()`: the fix reached only one of (at least) four then-existing copies of this class, precisely the duplicated-source hazard flagged above. We raised this gap with the original authors as part of our disclosure; they confirmed it and, on 2026-07-23, pushed a further commit replacing `train/mamba_policy_par.py`'s `step()` with the corrected two-stream logic (which we verified, by re-fetching the file, now matches the canonical semantics), and added a comment clarifying that the legacy, `step()`-trained recipe and the parallel, `forward_sequence`-trained recipe use incompatible checkpoints and must not be mixed. We report this as a concrete instance of the audit's actionable value; it changes nothing in this paper's own results, which remain pinned to the commit we originally cloned, `9bea607` (§ 3). A reader consulting the live repository after our writing should still recheck its state, since we do not track it further.

By the structured state-space duality, the *existence* of an exact recurrent equivalent is guaranteed and MTIL's Algorithm 2 already *specifies* the intended controller; our contribution is therefore not a new inference algorithm but the correct realization and verification of MTIL's intended $O(T)$ controller, which the released

Table 5: Cross-repository audit of recurrent rollout paths in released SSM robot policies.

Repo	recurrent `step()` present	used at rollout	verified ≡ parallel
MTIL	yes	**yes** (the only one)	**no — broken** (§ 6.1)
RoboSSM	branch present, but the `Longhorn` mixer defines no `step`/ `_get_states_from_cache`/ `allocate_inference_cache` — **non-functional**	no (parallel scan)	no — `AttributeError` on first use
MaIL	no (re-stacks the window, no state across steps)	no	n/a
RoboMamba	text-autoregression only (action head is one parallel forward)	no	n/a

code does not provide and which § 7 requires to profile honestly (§ 8's history experiments run on the never-broken `forward_sequence` path and do not require it).

We are precise about the *operational* stakes, to avoid overselling. The $O(T^2)$ re-encode path was already correct and deployable at manipulation horizons (it is the path behind our 96–97% reproduction), so the practical cost of the shipped bug is bounded: it bites at deployment only where the $O(T)$ controller is actually needed (long horizons or tight per-step compute budgets) and, more importantly here, wherever a controller *faithful to the trained policy* is required to study the model (the profiling of § 7 demands it; the history experiments of § 8, by contrast, run on the never-broken `forward_sequence` path and do not). The bug's significance is thus less "MTIL is undeployable" than "MTIL's headline recurrent controller does not exist in a form faithful to `forward_sequence` in the release, and the one experiment that would scrutinize its efficiency claim on that (§ 7) cannot be run on the path it ships."

6.2 Is this systemic? A cross-repository audit

We asked whether a broken or unverified recurrent rollout path is unique to MTIL. We selected the four released, named SSM robot/imitation policies with public code (a fifth, DSSP, has no public code and is excluded), and classified each by reading its deployment entry point and searching its tree for an equivalence/consistency test.

We define these classes precisely and restrict every claim to the exact commit cited (Table 6); we did not track subsequent upstream changes. **Deployed**: the released evaluation/rollout entry point calls a method that carries state across successive control-loop calls. **Absent**: no such method exists anywhere in the mixer or agent for the manipulation/control path specifically (a repository may still have unrelated recurrence, e.g. for language autoregression), so the model is necessarily re-forwarded over the accumulated window, or is a single parallel forward. **Unimplemented**: the mixer class exposes a branch that would call a recurrent method were it ever invoked, but the call target does not exist on the class; the branch is not taken at deployment (deployment always takes the parallel path), and invoking it directly raises an exception. **Broken**: the recurrent call exists, executes, and returns, but its output diverges materially from the parallel/training path. **Verified**: a test asserting recurrent≡parallel equivalence exists in the repository.

We went beyond reading and probed RoboSSM's recurrent path empirically, on a verbatim, standalone copy of its `Longhorn` mixer (`longhorn.py:416`; the class `create_block` instantiates for *every* block, line 593), with random weights. The result is sharper than "unused": the recurrent-decode branch is dead code that cannot execute. `Longhorn.forward` routes any call carrying `inference_params` (i.e. any incremental/recurrent decode) through `self._get_states_from_cache(...)`, and for `seqlen_offset>0` through `self.step(...)`. But the mixer inherits only `nn.Module` and defines *none* of `step`, `_get_states_from_cache`, or `allocate_inference_cache` (the surrounding `Block`/`MixerModel` even wire an `allocate_inference_cache` path straight into the absent mixer method). Invoking the path raises `AttributeError: 'Longhorn' object has no attribute '_get_states_from_cache'` on first use; we reproduce this on the standalone class (`test/robossm_killtest.py`, Part A). So RoboSSM's advertised constant-per-step inference is not merely unused, it is *unimplemented*, and deployment necessarily falls back to the O(L) parallel scan

Table 6: Provenance for the cross-repository audit: exact commit, deployment/definition site, and classification per Table 5.

Repo	Commit	Deployment / definition site	Classification
MTIL	`9bea607`	`test/inference_M.py:81` calls `policy.step()` every control step; the bug is in `train/mamba_policy_par.py:641` (`step()`), residual collapse at lines 708–717	Deployed, broken (§ 6.1)
RoboSSM	`07df6df`	`libero/lifelong/models/longhorn.py:478` (`Longhorn.forward`), undefined call at :488, `Block.allocate_inference_cache` at :573–574	Unimplemented (raises `AttributeError`)
MaIL	`a8012a0`	`agents/models/bc/goal_bc_agent.py:279` (`BC_Agent.predict()`), window re-stack at lines 294–306; config-mapped to a Mamba backbone via `config/agents/goal_bc_mamba_encdec.yaml`	Absent (bounded-window re-forward, no persisted state)
RoboMamba	`c2f52c4`	`src/model/manip.py:68` (`LinearManip.forward`, `cache_params=None` default); no manipulation rollout script is released	Absent for the manipulation head (`cache_params` threading exists only for the language-autoregressive path, `src/model/llm.py:32-54`)

Full commit hashes (the table shows the first 7 characters): MTIL `9bea607875e20366bbb6a8cff4a935ab84f5ba57`; RoboSSM `07df6df3ca4839cfef9d0be82108aaa217a36a32`; MaIL `a8012a0018ce2e5e26adff3bb3336190be2595ea`; RoboMamba `c2f52c49217a0e383eadb71a5751567fb05b452c`.

(`longhorn_inner_fn`, `inference_params=None`). In RoboSSM's favor, its *parallel* path is structurally sound: `Block.forward` keeps the canonical two-stream residual (`residual` ← `hidden + residual; hidden` ← `mixer(LN(residual))`), so it does *not* carry MTIL's inter-block residual bug; MTIL's defect arose precisely from re-implementing that inter-block loop separately for decoding. But the linear-time recurrence RoboSSM advertises does not run.

None of the four ships a step-versus-parallel equivalence test. We state the conclusion conservatively: *among the released SSM robot policies we examined, the advertised linear-time recurrent inference is unrealized or unverified in every public rollout path.* Either the policy re-forwards the window with the parallel scan (RoboSSM/MaIL/RoboMamba), or it ships one that is broken (MTIL, the only deployed recurrent path). The point is not that equivalence testing is novel; the upstream `mamba_ssm` library does test scan↔step consistency *for a single block's kernel.* But MTIL's defect is at the *application* level: the inter-block residual loop re-implemented outside the library's decode path (§ 6.1), which a kernel-level equivalence test cannot catch and which these robot-policy repositories never add while advertising the linear-time benefit. This pre-empts the objection "re-forwarding the window is a design choice": even granting it, for RoboSSM specifically the advertised constant-per-step benefit is then measured on a parallel-scan path whose cost is *not* constant in the horizon (MaIL's bounded window and RoboMamba's single forward are each constant per step, just never recurrent). We close this with an empirical second-repo kill-test rather than leaving it to future work: RoboSSM's recurrent path does not merely go unverified; it cannot be invoked at all (above). This *strengthens* the cross-repository reading: the only released policy whose recurrent path actually executes at rollout is MTIL's, and that one is broken. In one line: of the four released SSM robot policies we examined, one deploys a recurrent rollout path (MTIL) and it is broken; one ships a recurrent branch that cannot execute (RoboSSM); two never deploy one (MaIL, RoboMamba); none verifies one against its parallel scan.

7 C2 — The Efficiency Advantage Is Real but Practically Invisible

With a faithful controller in hand (§ 6.1), we profile the two per-step inference modes of the fixed policy across horizons T∈{1,. . . ,400}: the *re-encode* used before the fix (`vision(1 frame) + blocks(history t)`, whose *per-step* block cost is $O(t)$ in the current history, i.e. $O(T^2)$ summed over an episode) and the corrected *recurrent step* (`vision(1 frame) + step_block`, O(1) per step). We name these "$O(T^2)$" and "$O(T)$" by their *per-episode* cost (the § 1 promise of "O(1) per control step" is the per-step view); in Figure 2 the per-step costs are $O(t)$ vs O(1).

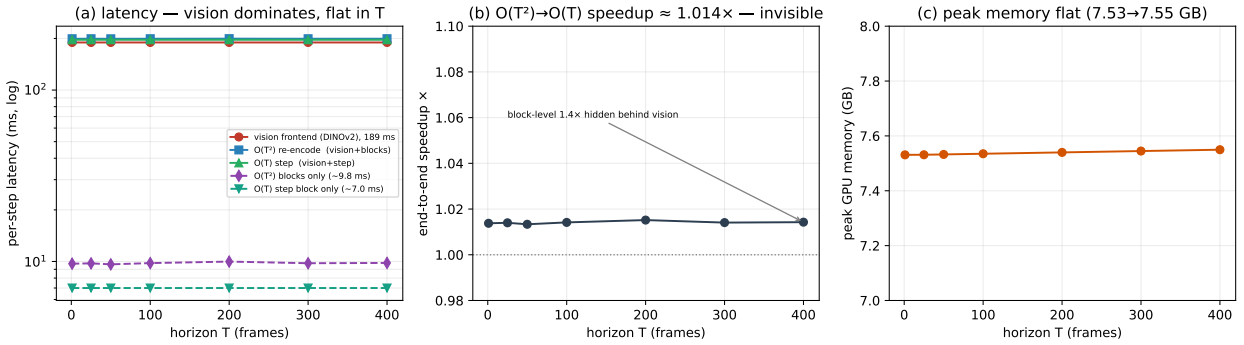

Figure 2: Per-step latency (±1 SD error bars from the genuine per-trial distribution, 20 trials per point across 7 horizons), end-to-end speedup, and peak memory vs horizon T, measured on a second, independent GPU host (Sec. 7). Here the vision frontend (148.0 ± 1.9 ms) exceeds the SSM block cost by ∼12× (vs the $O(T^2)$ re-encode block, ∼11.9 ms) to ∼19× (vs the $O(T)$ step block, 7.8 ± 0.3 ms); the block-level $O(T)$ win (∼1.5×) is swamped to an end-to-end ∼1.03×; this is a different point estimate from the first-host measurement (1.013–1.016×) reported in the main text, but both are small, few-percent effects on the same side of 1×. The absolute vision latency differs by host, as expected, but the qualitative pattern, vision-dominated and flat in T, replicates. The ordering is specific to this heavy, uncached DINOv2 ViT-L/14 frontend.

- **Vision frontend** (DINOv2 + spatial adapter, one frame): **189.4 ms** (3-run quiet-host median, [188.4–190.3]; system-load-dependent, ≈170–190 ms across runs), paid every step by *both* modes.
- **Mamba blocks**: **∼7 ms** for the $O(T)$ step; **∼9.8 ms** for the $O(T^2)$ re-encode, which, measured across T∈{1,25,...,400} (20 trials each), is essentially flat (9.6–10.0 ms): the chunked scan is kernel-launch-overhead-bound, its $O(T)$ compute negligible against fixed overhead at these sizes, so the feared quadratic re-encode cost does not materialize at manipulation horizons. Peak memory likewise flat at 7.53 → 7.55 GB.
- **Speedup**: at the block level the $O(T)$ step is **∼1.4**× cheaper (flat in T); end-to-end it is **∼1.014**×, invisible (flat in T).

So the corrected controller's per-step block win *is* real, but it is swamped end-to-end by a vision frontend more than an order of magnitude larger (≈27× the $O(T)$ step cost) and constant per step. The controller of § 6 thus *matters for correctness* (it must reproduce the trained policy) but is practically invisible for speed at manipulation horizons; it would bite only at horizons far beyond 400, where the $O(t)$ compute term (here below measurement noise) would finally exceed the ∼9.8 ms fixed block overhead, or with a much lighter, cached encoder. The actionable conclusion is that, *for this frontend* (frozen DINOv2 ViT-L/14, per frame, uncached), SSM-VLA efficiency effort should target the visual frontend, not the scan. The cost ordering is encoder-specific: in a π_0-class VLA the language head dominates and vision is the smallest term (§ 2), so we do not generalize "vision dominates" to all VLAs. DSSP's lighter encoder is precisely the regime where the $O(T)$ win *is* visible: the inverse of our encoder-gated result, and consistent with both being correct. (The absolute vision latency is system-load-dependent: across runs we observe ≈170–190 ms/frame. A 3-run quiet-host re-measurement gives t_vision = 189.4 ms [188.4–190.3], t_step ≈ 7 ms, and $O(T^2)$ re-encode blocks ≈ 9.8 ms *flat* across T∈[1,400], for an end-to-end $O(T^2)$→$O(T)$ speedup of 1.013–1.016×, flat in T. The win never exceeds a few percent across the 170–190 ms vision range, so the invisibility conclusion is insensitive to the load-dependent absolute latency. As a further, genuinely independent check, we re-ran the identical profiling script on a second GPU host (Figure 2), recording each trial's individual timing rather than only the mean, so the figure now carries real error bars (±1 SD, n=20/point). That host gives a different absolute vision latency (148.0 ± 1.9 ms vs 189.4 ms here), as expected from host and load differences, but the same qualitative pattern: flat in T, vision-dominated, block-level speedup preserved (∼1.5×), and end-to-end speedup similarly small (∼1.02–1.03× vs 1.013–1.016×). We read this as evidence that the qualitative

bottleneck, not its exact magnitude, is what is robust across hardware; we do not have the resources for a systematic multi-architecture sweep, and we do not claim one.)

8 C3 — A 16-Frame Context Recovers Most of the Headline Performance

8.1 A Markovian negative control

On the near-Markovian `transfer_cube`, a windowed-*trained* k=16 policy, trained on 16-frame windows and evaluated with the matching window, reaches **90.0%** (3×50 seeded, Wilson-95% [84.2, 93.8]; per-seed 92/86/92%). This is modestly but significantly below the full-history model's 96.7% by the pooled two-proportion test (z=2.3, p=0.02; 135/150 vs 145/150), yet *vastly above* the 0% that same full-history model scores under eval-time context truncation. (The 96.7% comparator is the C1 fixed-`step()` closed-loop number of § 6.1, `evaluate_step_fs.py` (Table 8), which is numerically verified equivalent to `forward_sequence` to $\sim 10^{-3}$ (§ 6.1) but is, strictly, a distinct script. We note this only for provenance: substituting the 97.3% `forward_sequence` reproduction of § 4.1 (`evaluate_fs.py`, never subject to the C1 defect) as the full-history comparator instead changes nothing in this section's conclusion, since the two are statistically indistinguishable. The windowed condition, `evaluate_fs_windowed.py`, is built on `forward_sequence` directly and does not invoke `step()` at all.) Because each condition rests on only three seeds, we also report the seed-level companion (§ 3): a t-interval (df=2) gives [81.4%, 98.6%] for k=16 and [93.8%, 99.5%] for the full-history model, which overlap substantially over [93.8%, 98.6%]. We present both without resolving the tension: the pooled test has more nominal power but treats 150 non-independent rollouts as full replicates, while the seed-level interval is the more conservative view built from three independent evaluation replicates (rollout-pose-sampling seeds on a single trained checkpoint per condition, § 3), rather than three independently trained models. Under either reading, k=16 is far from the 0% floor the full-history model shows under truncation. The full-history model's collapse under eval-time truncation (0% at k≤16) is therefore *learned dependence*, not task necessity: a causal-confusion phenomenon in which the policy latches onto a history feature that is not required to recover most of the headline success. So `transfer_cube`, the benchmark on which MTIL's headline is reported, *cannot, by construction, demonstrate that full (unbounded) history is required.* We use windowed-*trained* baselines throughout (never a full-history model truncated at evaluation, which conflates the question with a cold-start artifact).

The two training runs are not budget-matched, and the mismatch does not favor the windowed condition. The full-history model (the checkpoint of § 4.1/§ 6.1) trained for a configured 150 epochs, completing 119 (4,760 gradient steps at 40 steps/epoch, one training trajectory per step), each step a full 400-frame trajectory with a per-frame masked loss (≈1.90M supervised frame-instances total). The k=16 windowed model trained for a configured, and fully completed, 45 epochs (1,800 gradient steps), each step drawing W=16 random 16-frame windows (≈461K supervised frame-instances total, about a quarter of the full-history run's). Both select checkpoints by the identical criterion, lowest held-out loss (`val_epoch_loss`, `ModelCheckpoint(monitor='val_epoch_loss', mode='min')`). By both gradient-step count (∼2.6×) and total supervised frame-instances (∼4×), the full-history run received the larger optimization budget; the windowed model was not compensated with extra exposure for its shorter context. If optimization budget explains part of the 90.0%-vs-96.7% gap, it should work *against* the windowed model, so 90.0% is, if anything, a conservative estimate of what a budget-matched windowed policy would reach, not an inflated one.

8.2 A non-Markovian benchmark: full shows no detectable advantage

We port the pipeline to LIBERO-Mem (dual-camera, square 128^2 inputs, the standard recipe fixes) and run full-versus-windowed-*trained* comparisons on three tasks (n=20 each; exploratory, no CI).

The reading is narrow: full history shows no detectable advantage here (n=20, both policies floor), but the one task with a real gap (T3) is driven by sample-efficiency (the windowed model draws many short windows per trajectory), and the others tie or mutually floor. Like § 8.1, then, this benchmark cannot positively *test* the memory hypothesis either. The common thread is that basic, Markovian manipulation is the binding constraint and is never solved well enough for the memory-demanding portion of the task to be exercised; the windowed policy, drawing many short windows per trajectory, learns the short-horizon skill

Table 7: LIBERO-Mem full-versus-windowed comparison (exploratory, n=20 each, no CI).

Task	full-history	windowed (k=16)	reads as
T1 (Markovian)	∼10–17%	∼10%	tie (within n=20 noise)
T9 (occlusion)	0%	0%	mutual floor — *uninformative*, neither reaches the memory step
T3 (counting)	0.10 / 3 subgoals	**0.80** / 3	windowed wins — but by *sample-efficiency*, not memory

more sample-efficiently. This is paper-confirmed: on LIBERO-Mem counting, published implicit-memory policies report 0% at *both* short and long history; the failure is the paradigm, not our implementation. We claim only "full does not help here," never "history hurts."

8.3 A controlled delayed-cue probe (exploratory, inconclusive)

To try to isolate memory from manipulation difficulty, we built a *delayed-cue* variant of `transfer_cube`: a cue shown only in the first τ=12 of 500 frames selects one of two target bins, then vanishes, so the bin choice exists *only* in early history; any window shorter than the cue→action gap cannot exceed chance by information theory. The intent was a setting where manipulation cannot floor.

That intent was not met, so we report this probe as exploratory and inconclusive rather than as a clean result. Two facts limit it. First, the cued variant does not inherit `transfer_cube`'s 96%: the trained cued policy reaches only ∼25% success (the added bins/place-down and the 500-frame horizon make it materially harder), so a manipulation-mode-collapse confound is *not* excluded and the probe cannot cleanly separate "memory not used" from "manipulation not solved." Second, the cue *is* used, but only partially and unreliably: on a counterfactual cue-swap (8 paired poses; flip only the cue, fix the pose) the converged full-history model has a follow-rate of 68.8% (11/16 single-cue trials, above the 50% chance of choosing the cued bin) and a flip-rate of 37.5% (3/8 paired poses re-route to the other bin when the cue flips). A separate run that added an auxiliary cue-prediction loss made the hidden state linearly cue-decodable (a jointly-trained `cue_head` reached val-accuracy 0.93) but did not improve cue-driven action; we do not draw an "encode-versus-use dissociation" from this, because that comparison mixes an undertrained, objective-forced model against a converged one and is confounded by the manipulation floor. The reading consistent with § 8.1–§ 8.2 is the weak one: *the full-history policy uses the delayed cue only partially, and this task is too manipulation-limited to test the memory hypothesis cleanly.* A clean test would require a cued task the policy solves at high success, a frozen-features (not jointly-trained) probe, and a seeded cue-swap with a CI over many paired poses, which we leave to future work.

9 Discussion and Limitations

Our corrections do not refute MTIL's utility; they reattribute and correct it. MTIL works, and its advantage over ACT is real but, on the paper's own authoritative numbers, *modest* (cube transfer 100% vs 90%) rather than the 97-vs-25 our under-tuned ACT suggests (§ 5), so we make no claim that pretrained vision *drives its success*. What we do show is that its $O(T)$ deployment claim is correct yet *practically invisible* behind the vision frontend at deployment (§ 7, a latency result independent of success), and that a 16-frame context recovers most of the full-history performance on the benchmark where we can measure it (§ 8); a benchmark solvable at that level by a short-context policy cannot show unbounded history is required. The one defect that genuinely matters, the broken released `step()` (§ 6), is also the most actionable, and it is the load-bearing enabler for § 7: an honest efficiency profile requires a controller faithful to the trained policy. § 8's comparisons, by contrast, run on the always-correct `forward_sequence` path and do not depend on this fix.

The limitations are as follows. (i) Insertion is a partial reproduction on a protocol-sensitive benchmark. (ii) Our ACT baseline (∼25%) is below ACT's literature ≈86% under an untuned canonical recipe; we

report the matched comparison and cite the paper's own ACT number rather than over-claim ACT's ceiling. (iii) The cross-repository audit (§ 6.2) is primarily code-reading, augmented by one empirical kill-test: RoboSSM's recurrent decode path raises `AttributeError` on first use (its deployed `Longhorn` mixer implements none of `step`/`_get_states_from_cache`/`allocate_inference_cache`), which we reproduce on a verbatim standalone copy of the class. We scope this to that mixer (the one `create_block` deploys) and did not stand up RoboSSM's full LIBERO stack, so we claim nothing about its task success, only that the linear-time recurrence it advertises cannot execute. (iv) The LIBERO-Mem negative is on a single-task end-to-end model that floors on basic manipulation; it shows full history does not help *in this regime*, not that history never helps; DSSP demonstrates it helping on tasks designed for it. (v) The C2 profile is unseeded (Table 1, row 8) and measured on shared GPU hosts (two, for the cross-hardware check of § 7), so the absolute vision latency is system-load-dependent; the qualitative shape (flat in T, vision-dominated) and the $\sim$1.014$\times$ end-to-end conclusion are robust to that uncertainty. The block cost is *overhead-bound* across our T$\leq$400 range (its O(t) compute term sits below measurement noise), so the measurement contains no signal about where a recurrent O(1) step would finally overtake the re-encode; the crossover horizon is unobserved, not estimated.

10 Conclusion

We reproduce a published full-history Mamba VLA under a seed-controlled protocol, find and fix its broken O(T) controller (verified on the published code against our own `forward_sequence`-trained reproduction and reported with a confidence interval, 0.7% $\rightarrow$ 96.7%; MTIL's own published numbers are not shown to be affected, § 6.1), show by cross-repository audit that, among the four released SSM robot policies we examined, none both uses and verifies its recurrent path, measure that the corrected O(T) advantage is invisible behind the vision frontend, and show that the headline benchmark is largely short-context-solvable while our memory-targeted probes do not establish a full-history advantage (a k=16-trained policy solves transfer_cube at 90%, a LIBERO-Mem comparison shows no memory gain, and a delayed-cue probe is inconclusive). None of this diminishes the policy's utility; it corrects *why* it works, and it points SSM-VLA effort, for heavy-encoder designs, at the visual frontend rather than the sequence model. We release a one-command artifact that reproduces the central correctness finding on a pristine clone of the published code.

Appendix A — Reproduction artifact and audit method

Artifact. `artifact_c1_step_audit/` contains: a README with a $\approx$10-minute reproduction (clone upstream $\rightarrow$ numeric kill-test $\rightarrow$ closed-loop 0.7% vs 96.7%); `audit_upstream_step.py` (loads the pristine and fixed policies side-by-side from one checkpoint and prints each path's per-frame divergence from `forward_sequence`, fp32 and fp16); `evaluate_step_fs.py` (closed-loop, with a flag to drive the rollout with the *unmodified published* `step()`); `cueswap_diag.py` (the § 8.3 counterfactual probe); `robossm_killtest.py` (the § 6.2 second-repo empirical probe: instantiates RoboSSM's deployed `Longhorn` mixer from a verbatim standalone copy and shows its recurrent decode branch raises `AttributeError` on first use); the exact fix as a diff; and the pristine clone's HEAD for provenance. The checkpoint bundle (weights-only checkpoint, scaler, one episode and its cached features, the pristine policy module, SHA256 checksums) is archived on a DOI host, and the source directory and pristine clone via Software Heritage; identifiers are withheld for double-blind review (the audit code is included as supplementary material) and will appear in the camera-ready.

Cross-repository audit method (§ 6.2). For each repository we (a) located the deployment/evaluation entry point and recorded whether it calls a recurrent `step()` or re-forwards the accumulated window through the parallel scan, and (b) searched the source tree for a test asserting recurrent$\leftrightarrow$parallel equivalence (matching `equiv|consist|verify|recurr`). For RoboSSM we additionally (c) extracted its deployed `Longhorn` mixer class verbatim into a standalone module and *executed* its recurrent branch (random weights), confirming empirically that it raises `AttributeError` (`test/robossm_killtest.py`). The per-repository file:line evidence is released with the artifact. Repository selection: the named, released SSM robot/imitation policies with public code; DSSP is excluded for lacking public code.

Table 8: Claim-to-evidence map. Every load-bearing number traces to a script and a preserved result file.

Claim	Paper	Script	Evidence file	Ckpt/data
transfer 97.3%	§ 4.1 / T1#1	`evaluate_fs.py`	`eval_transfer_seed*.log`	version_6
insertion 40.7%	§ 4.2 / T1#2	`evaluate_fs_insert.py`	`eval_insert_seed*.log`	version_14
C1 numeric kill-test	§ 6.1 / T1#5	`audit_upstream_step.py`	deterministic (regenerable)	v6 + pristine `9bea607`
C1 fixed 96.7%	§ 6.1 / T1#3	`evaluate_step_fs.py`	`c1_seeded2.log`; `c1_fixed_step_3x50.json`	version_6
C1 pristine 1/150 (0.7%)	§ 6.1 / T1#4	`evaluate_step_fs.py` (PRISTINE=1)	`c1_pristine_step_3x50.json`	v6 + pristine
C2 latency / speedup	§ 7 / T1#8	`profile_oT.py`	`c2_profile_oT.json` (first host), `c2_profile_oT_host2.json` (second host, per-trial SD backs Figure 2)	version_14
C3 k=16 90%	§ 8.1 / T1#9	`evaluate_fs_windowed.py`	`c3_k16_windowed_results.json`	version_16
§ 6.2 RoboSSM dead code	§ 6.2 / T1#6	`robossm_killtest.py`	`AttributeError` (Part A, CPU)	random weights
delayed-cue (exploratory)	§ 8.3 / T1#11	`cueswap_diag.py`	`cueswap*.log`	delayed-cue ckpt

References

Kevin Black et al. π_0: A vision-language-action flow model for general robot control. *arXiv preprint arXiv:2410.24164*, 2024.

Tri Dao and Albert Gu. Transformers are ssms: Generalized models and efficient algorithms through structured state space duality. In *International Conference on Machine Learning (ICML)*, 2024.

Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces. *arXiv preprint arXiv:2312.00752*, 2023.

Zhiyuan Guan, Jianshu Hu, Han Fang, Yunpeng Jiang, Yize Huang, Shujia Li, Xiao Li, and Yutong Ban. Dssp: Diffusion state space policy with full-history encoding. *arXiv preprint arXiv:2605.14598*, 2026.

Xiaogang Jia et al. Mail: Improving imitation learning with selective state space models. In *Conference on Robot Learning (CoRL)*, 2024.

Wenqi Jiang, Jason Clemons, Karu Sankaralingam, and Christos Kozyrakis. How fast can i run my vla? demystifying vla inference performance with vla-perf. *arXiv preprint*, 2026.

Bo Liu, Yifeng Zhu, Chongkai Gao, Yihao Feng, Qiang Liu, Yuke Zhu, and Peter Stone. Libero: Benchmarking knowledge transfer for lifelong robot learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

Jiaming Liu et al. Robomamba: Multimodal state space model for efficient robot reasoning and manipulation. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

Maxime Oquab et al. Dinov2: Learning robust visual features without supervision. *Transactions on Machine Learning Research (TMLR)*, 2024.

Edwin B. Wilson. Probable inference, the law of succession, and statistical inference. *Journal of the American Statistical Association*, 22(158):209–212, 1927.

Youngju Yoo, Jiaheng Hu, Yifeng Zhu, Bo Liu, Qiang Liu, Roberto Martín-Martín, and Peter Stone. Robossm: Scalable in-context imitation learning via state-space models. In *Conference on Robot Learning (CoRL)*, 2025.

Tony Z. Zhao, Vikash Kumar, Sergey Levine, and Chelsea Finn. Learning fine-grained bimanual manipulation with low-cost hardware. In *Robotics: Science and Systems (RSS)*, 2023.

Yulin Zhou et al. Encoding full history with mamba for vision-language-action imitation learning. *IEEE Robotics and Automation Letters (RA-L)*, 2025. arXiv:2505.12410.