UNIVERSITY OF CALIFORNIA
RIVERSIDE

Memory-Augmented Vision-Language-Action Policies for Robot Manipulation

A Thesis submitted in partial satisfaction
of the requirements for the degree of

Master of Science

in

Electrical Engineering

by

Litian Gong

June 2026

Thesis Committee:

Dr. Jiachen Li, Chairperson
Dr. Hang Qiu
Dr. Mingyu Cai

The Thesis of Litian Gong is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

I am deeply grateful to my advisor, Dr. Jiachen Li, for his guidance, patience, and support throughout this work, and to my committee members, Dr. Hang Qiu and Dr. Mingyu Cai, for their time and thoughtful feedback. I thank my labmates and collaborators in the Trustworthy Autonomous Systems Laboratory for many helpful discussions, and the UCR High-Performance Computing Center for the computational resources on which the experiments in this thesis were run. Finally, I thank my family for their unconditional love and support.

To my family.

ABSTRACT OF THE THESIS

Memory-Augmented Vision-Language-Action Policies for Robot Manipulation

by

Litian Gong

Master of Science, Graduate Program in Electrical Engineering
University of California, Riverside, June 2026
Dr. Jiachen Li, Chairperson

Vision–Language–Action (VLA) models have become a dominant paradigm for generalist robot manipulation, yet most are effectively memoryless: they map the current observation and a language instruction to an action without an explicit mechanism for retaining and reusing information across long-horizon, partially observable episodes.

This thesis makes three contributions toward memory-augmented VLA policies. First, we unify three heterogeneous VLA backbones—$\pi_{0.5}$, Cosmos Policy, and DreamZero—under a single `BasePolicy` interface, so that the same evaluation and rollout infrastructure applies uniformly to all three. Second, we introduce *DynaMem*, an input-level dynamic memory mechanism that stores frozen video-tokenizer latents at physically detected subtask boundaries and injects them as prefix tokens, without modifying backbone internals. DynaMem is constructed so that training-time memories and inference-time memories are bitwise identical, eliminating the train–inference mismatch present in prior memory-augmented VLAs. Third, we conduct a controlled evaluation on the RoboMemArena benchmark with a memory-zeroing diagnostic that directly measures whether the policy uses its memory.

Our experiments yield a careful negative result: when DynaMem is fine-tuned on top of a converged behavior-cloned $\pi_{0.5}$ policy, task performance is statistically indistinguishable from a no-memory control, and zeroing the memory at inference time does not degrade performance—the policy learns to ignore the injected tokens. We trace this to a structural property of behavior cloning from plateaued checkpoints: the training loss can be minimized without consulting memory, so the zero-initialized memory pathway receives no gradient pressure. The injection overhead itself is negligible (under 10 ms per memory write and 12 μs per control step). We distill these findings into design guidelines for future memory-augmented VLA training.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Recent advances in large-scale pretraining have produced Vision–Language–Action (VLA) models that can follow natural language instructions and manipulate objects in unstructured environments [3, 1]. These models learn broad manipulation skills from diverse demonstration data and generalize across tasks and object configurations. Despite their promise, most deployed VLA policies share a fundamental limitation: they are effectively *memoryless*. At each control step the policy receives only the current RGB observation and a language goal; no information from earlier in the episode is explicitly retained.

Real-world manipulation tasks, however, are often long-horizon and partially observable. An agent may need to recall which objects it has already placed, track the progress of a multi-step assembly, or recover from an occlusion by referencing what it observed before the occlusion occurred. Without memory, VLA policies are forced to re-infer all relevant context from the current frame alone, which limits their reliability on such tasks.

1.2 Problem Statement

The central question addressed in this thesis is: *how can we equip VLA policies with an explicit memory mechanism without rewriting each backbone's internal architecture?* A secondary engineering challenge is that the research community now maintains several competing VLA backbones with incompatible inference interfaces; developing and evaluating memory augmentation for each one independently would be prohibitively expensive. We therefore also ask: *how can a common software interface reduce the per-backbone cost of adding and evaluating memory?*

1.3 Contributions

This thesis makes the following contributions:

- **Unified backbone interface.** We design and implement a `BasePolicy` abstraction that places three heterogeneous VLA backbones—$\pi_{0.5}$, Cosmos Policy, and DreamZero—behind a single interface for rollout and evaluation. Thin adapter layers bridge each backbone's native API to the shared contract without hand-rolling model logic.

- **DynaMem.** We introduce an input-level dynamic memory injection mechanism that is backbone-agnostic: frozen video-tokenizer latents are written at physically detected subtask anchors into an append-only bank and injected as backbone-prefix tokens through a zero-initialized projector. The design guarantees that training-time and inference-time memory contents are bitwise identical, eliminating the train–inference mismatch of prior memory-augmented VLAs.

- **A controlled empirical study on RoboMemArena.** We evaluate the memory-augmented system in closed loop on the RoboMemArena (RMA) benchmark [5] with two safeguards—a continued-training control arm and a memory-zeroing diagnostic—that isolate the true contribution of memory. The study yields a carefully established negative result: behavior-cloning fine-tuning from a converged checkpoint does not adopt the injected memory. We provide a mechanistic explanation (gradient starvation of the zero-initialized memory pathway under a near-zero imitation loss) and distill design guidelines for future memory-augmented VLA training.

1.4 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 reviews relevant background on VLA models, memory in embodied agents, and the benchmarks used for evaluation. Chapter 3 describes the unified `BasePolicy` interface and the per-backbone adapters. Chapter 4 presents DynaMem in detail, including the anchor-based write policy and the train–inference exactness contract. Chapter 5 reports the controlled experimental study on RoboMemArena, the mechanistic analysis of the resulting negative finding, and overhead profiling. Chapter 6 summarizes the work, discusses limitations, and suggests directions for future research.

Chapter 2

Background and Related Work

2.1 Vision–Language–Action Models

VLA models formulate robot control as a sequence-to-sequence problem over multimodal tokens. Given an RGB image observation o_t and a language instruction l, the policy produces an action a_t or a chunk of future actions. Modern VLAs are typically initialized from large vision–language pretrained backbones and then fine-tuned on robot demonstration data [3, 1].

2.1.1 $\pi_{0.5}$

$\pi_{0.5}$ [10] is a flow-matching VLA built on the PaliGemma vision–language backbone. The model processes a *prefix* of image and language tokens with the PaliGemma VLM, and a separate flow-matching *action expert* generates an action chunk by iteratively denoising a noise-corrupted action sequence as the *suffix*, with a block-causal attention boundary between the two. The model is trained via behavior cloning on large-scale demon-

strations and supports closed-loop control with action chunking. $\pi_{0.5}$ is the first backbone instantiated in this thesis.

2.1.2 Cosmos Policy

Cosmos Policy [4] fine-tunes a pretrained video generation model from the Cosmos world-foundation-model platform [8] for visuomotor control and planning. Conditioning action generation on predicted future visual states provides implicit temporal consistency. Unlike $\pi_{0.5}$, Cosmos Policy performs diffusion-based generation in a video latent space for both visual prediction and action generation.

2.1.3 DreamZero

DreamZero [14] is a world-action model that learns action generation jointly with future video prediction, and can act as a zero-shot policy. Because the backbone explicitly models latent dynamics alongside the policy head, it is naturally suited for partially observable settings.

2.2 Memory in Embodied Agents

Memory mechanisms for embodied agents can be broadly categorized along two axes: *implicit* versus *explicit*, and *architectural* versus *input-level*. Implicit recurrent state (e.g., LSTM hidden states) is embedded in the model weights and updated automatically during the forward pass [13]. Explicit external memory stores representations in a separate buffer that the policy reads from and writes to. From an architectural perspective, some

approaches insert memory tokens into the attention layers of a transformer [2], while others augment the model's input sequence with retrieved context. The latter class, to which DynaMem belongs, requires no modification to backbone internals and is therefore directly applicable to any backbone behind a standard input interface.

For VLA policies specifically, MemoryVLA [12] maintains a perceptual–cognitive memory bank that is consolidated by token merging and injected into the action expert. PrediMem, the strongest published method on the RoboMemArena benchmark [5], follows a dual-system design: a high-level VLM planner summarizes history into text subgoals which are passed to a low-level policy. A shared weakness of these designs is a *train–inference mismatch*: MemoryVLA approximates its token-merged inference-time bank during training by sampling random sorted frame subsets, so merged entries are never seen by the training loss; keyframe-based methods such as PrediMem are trained with ground-truth keyframes (teacher forcing) but rely on model-predicted keyframes at inference time. DynaMem is designed so that the training-time and inference-time memory contents are *bitwise identical* (Chapter 4), removing this class of exposure bias by construction.

2.3 Benchmarks for Memory-Dependent Manipulation

RoboMemArena (RMA) [5] is a simulation benchmark designed to measure the performance of manipulation policies on tasks that cannot be solved from the current observation alone. It comprises 26 long-horizon tasks organized into four families—*Transferring*, *Occlusion*, *Counting*, and *Sequence*—implemented on the LIBERO/robosuite simulation stack with BDDL goal specifications. Examples include opening a specific drawer *again*

after interacting with several identical drawers (Occlusion of task progress), and pouring from a container exactly twice (Counting): in both cases the correct action depends on history that is not visible in the current frame. RMA reports two metrics: *task success rate* (TSR), the fraction of episodes in which the full BDDL goal is achieved, and *checkpoint success rate* (CSR), a partial-credit measure over annotated task stages. The RMA paper reports that vanilla $\pi_{0.5}$ achieves 21.5 TSR / 38.7 CSR, and its best memory method, PrediMem, achieves 38.5 TSR / 55.2 CSR; notably, MemoryVLA scores 15.0 TSR, *below* the memoryless $\pi_{0.5}$ baseline.

Other long-horizon manipulation benchmarks, such as LIBERO [6] and CALVIN [7], also feature multi-step task sequences but do not explicitly control for memory dependency. RMA is used in this thesis because it provides targeted evaluation of the memory contribution.

2.4 Positioning

Figure 2.1 summarizes the design space that motivates DynaMem. The published RMA results suggest that *where* memory enters the architecture matters more than *whether* it is present: memory delivered only to the motor-control suffix can even hurt, while memory that reaches the decision-making pathway helps. DynaMem therefore injects memory at the VLM-backbone prefix—upstream of decision-making—while treating the backbone itself as a black box behind the unified interface of Chapter 3. The empirical study in Chapter 5 tests whether this placement alone is sufficient when the backbone is fine-tuned with standard behavior cloning; the answer is more subtle than placement and motivates the analysis of

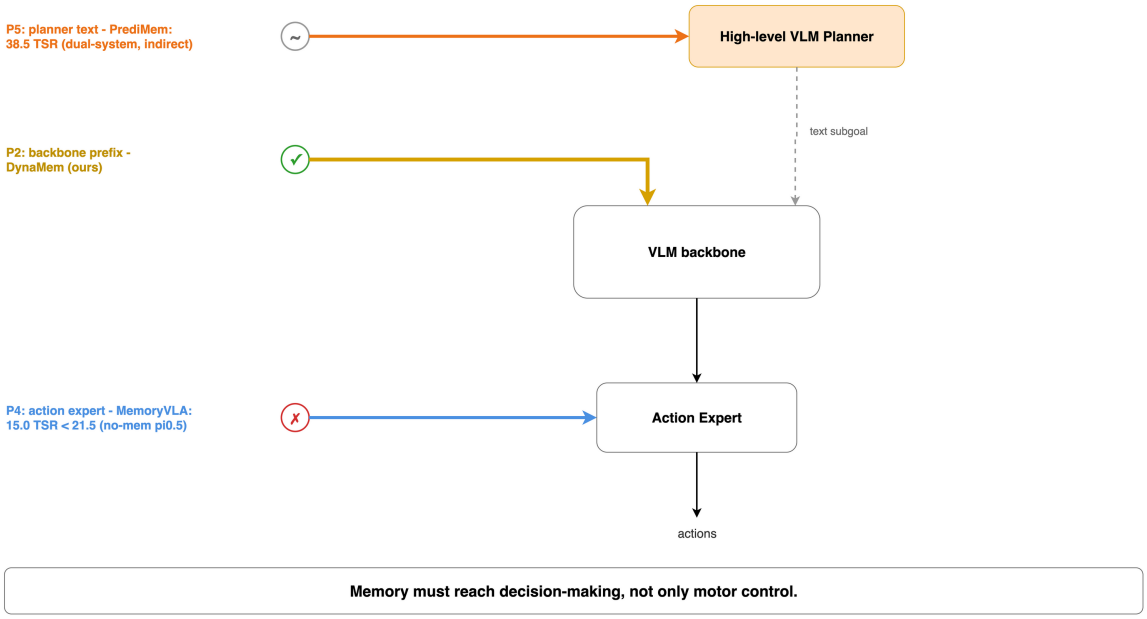

Figure 2.1: Candidate memory injection sites in a dual-system VLA architecture, annotated with published RoboMemArena evidence [5]. Injecting memory into the action expert (MemoryVLA, 15.0 TSR) underperforms the memoryless $\pi_{0.5}$ baseline (21.5 TSR), whereas routing memory through the decision-making pathway as planner text (PrediMem, 38.5 TSR) helps but is indirect and lossy. DynaMem injects memory as prefix tokens of the VLM backbone, reaching decision-making directly.

training-time gradient pressure in Section 5.4.

Chapter 3

A Unified Policy Interface across VLA Backbones

3.1 Design Goals

The primary goal of the unified interface is to allow a single evaluation and rollout pipeline to operate over multiple VLA backbones without requiring per-backbone code changes. Concretely, this means: (1) a single entry point for synchronous action inference, (2) a uniform observation and action contract, and (3) lifecycle management (episode reset, device placement, batching) handled outside of backbone-specific code.

A secondary goal is *native fidelity*: the adapter for each backbone should call the backbone's own inference or training routines directly, rather than re-implementing model logic. This guarantees that the behavior of the wrapped backbone matches its stand-alone behavior and that upstream fixes or weight updates propagate automatically.

3.2 The `BasePolicy` Interface

The `BasePolicy` abstract class defines four required methods: `act(obs) -> action`, `reset()`, `to(device)`, and `eval() / train()`. The observation contract specifies a dictionary with keys for RGB images, proprioceptive state, and language instruction. The action contract is a numpy array of joint positions or end-effector deltas, depending on the task.

All backbones are expected to implement the same interface regardless of their internal inference representation (tokens, latent codes, or noise sequences). Batching and device placement are handled by the interface wrapper so that adapters do not need to manage them.

3.3 Backbone Adapters

3.3.1 $\pi_{0.5}$ via RLinf-openpi

The $\pi_{0.5}$ adapter wraps the openpi inference client [9, 11]. Observations are preprocessed into the tokenized format expected by the $\pi_{0.5}$ action expert, the native `sample_actions` method is called, and the resulting action chunk is returned. Episode reset flushes the action chunk buffer.

3.3.2 Cosmos Adapter

The Cosmos adapter calls the Cosmos inference server via a thin HTTP client. Because Cosmos Policy operates at a lower frequency than $\pi_{0.5}$, the adapter implements an action interpolation layer to bridge the frequency mismatch with the control loop.

3.3.3 DreamZero Adapter

The DreamZero adapter interfaces with the DreamZero Python package directly in-process. It exposes the latent rollout state through the standard `reset()` hook so that the memory module can optionally condition on the latent state in addition to raw observations.

3.4 Training and Inference Parity

To verify that each adapter faithfully reproduces native backbone behavior, we run a set of deterministic regression tests: given the same observation and model weights, the adapter must produce the same action as the native inference path within numerical tolerance. This ensures that the abstraction layer introduces no behavioral divergence.

3.5 Discussion

The main trade-off of the abstraction is reduced flexibility: adapter methods that are not part of the shared contract (e.g., backbone-specific attention visualization) become inaccessible through the interface. In practice this has not been a limitation for the evaluation use cases in this thesis, but may require extension for more advanced introspection tasks.

Chapter 4

DynaMem: Input-Level Dynamic Memory Injection

4.1 Design Principle

DynaMem is built on the observation that VLA backbones share a common input boundary: a multimodal context consisting of images and text. By treating this boundary as the single injection point, memory can be provided to any backbone without touching its internal architecture. This composability is the key property that makes DynaMem backbone-agnostic.

Formally, let o_t denote the observation at step t and $\mathcal{M}_t$ denote the memory store at step t. DynaMem constructs an augmented input $\tilde{o}_t = \text{inject}(o_t, \mathcal{M}_t)$ and passes $\tilde{o}_t$ to the backbone policy π unchanged. The backbone is never aware that its input has been augmented, beyond a small projection layer that maps memory entries into its token

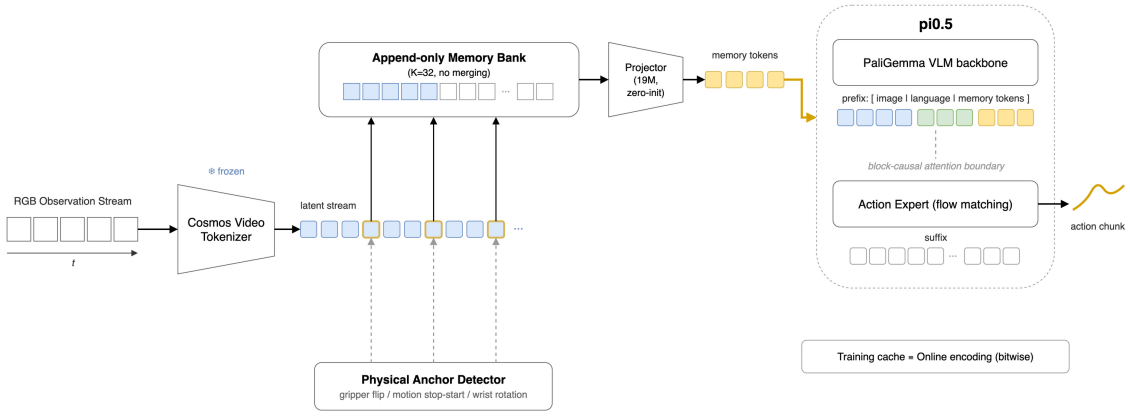

Figure 4.1: DynaMem system overview, instantiated on $\pi_{0.5}$. The RGB observation stream is encoded by a frozen Cosmos video tokenizer into a causal latent stream. A physical anchor detector (gripper-command flips, motion stop/start, wrist-rotation onsets) selects which latents to write into an append-only memory bank (K=32, no merging). A zero-initialized 19M-parameter projector maps stored latents to memory tokens, which are injected into the PaliGemma VLM prefix alongside image and language tokens, upstream of the block-causal attention boundary and the flow-matching action expert. The training-time memory cache and the online encoder produce bitwise-identical contents.

embedding space.

Three design questions then determine the mechanism, summarized in Figure 4.1: *what* to store (Section 4.2), *when* to write (Section 4.3), and *how* to inject (Section 4.4). A fourth requirement cuts across all three: the memory contents seen at training time must match those seen at inference time exactly (Section 4.5), a property that prior memory-augmented VLAs do not satisfy (Section 2.2).

4.2 Memory Substrate: Frozen Video-Tokenizer Latents

Each memory entry is a latent produced by the frozen Cosmos-Tokenizer CV4×8×8 causal video encoder [8]. The encoder compresses video by a factor of 4 in time and 8×8 in space; for 256×256 RGB input, one stored entry is a $16 \times 32 \times 32$ latent tensor. Using a

frozen, off-the-shelf video tokenizer has three advantages. First, the substrate is backbone-independent: the same latents can later be injected into video- or world-model backbones (Cosmos Policy, DreamZero) whose native representation is the same latent space. Second, freezing the encoder means the memory content distribution is stationary throughout training, which isolates the learning problem to the projection and the backbone's use of the tokens. Third, encoding is cheap (under 10 ms per write on a single GPU; Section 5.5), so memory writes do not interfere with closed-loop control.

Because the tokenizer is causal with temporal stride 4, its latent stream is defined on a frame grid of stride 4. We adopt a *sliding-window contract*: the latent associated with grid frame g is defined as the last temporal slice of the encoder applied to the 17-frame causal window ending at g (legal Cosmos clip lengths are $1 + 4k$ frames). This definition is computable both offline over a full demonstration and online from a rolling frame buffer, which is the basis of the exactness guarantee in Section 4.5.

4.3 Write Policy: Physical Anchor Detection

Writing every frame would flood a fixed-capacity bank with near-duplicate entries, while a fixed temporal stride is misaligned with task structure. DynaMem instead writes at *physical anchors*: frames where the interaction state of the gripper changes. The rule, denoted R5* and illustrated in Figure 4.2, fires an anchor at the union of three causal, proprioception-only events: (i) gripper-command flips, (ii) end-effector motion stop/start transitions, and (iii) wrist-rotation onsets, each debounced over 5 frames; the stored observation frame is the event frame plus 5 frames, so that the camera has settled on the

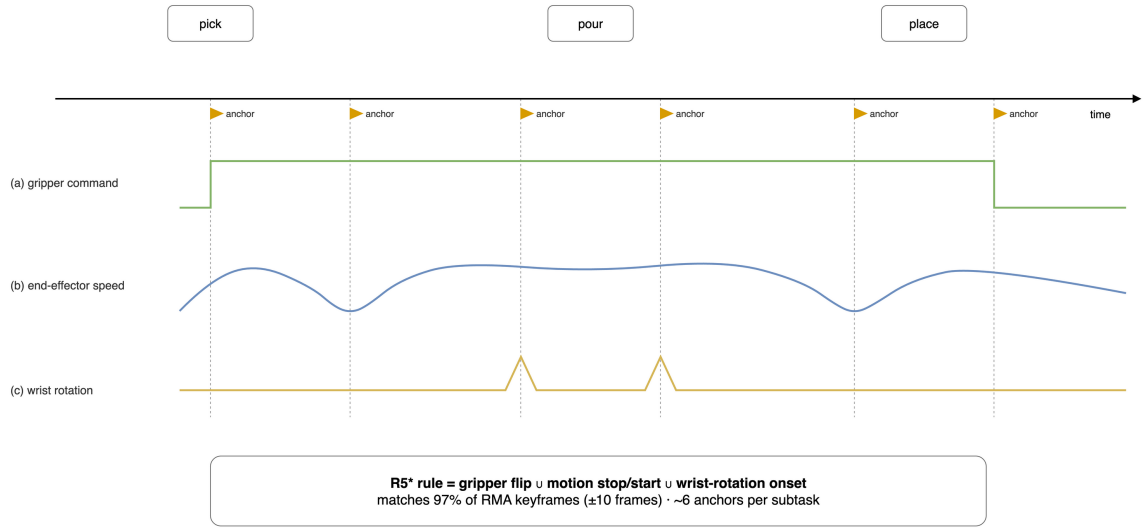

Figure 4.2: The R5* anchor rule. Anchors are fired at gripper-command flips, end-effector motion stop/start events, and wrist-rotation onsets (debounced over 5 frames; the stored observation frame trails the event by 5 frames). On RoboMemArena demonstrations the rule matches 97% of officially annotated keyframes within ±10 frames and fires roughly 6 anchors per subtask.

post-event scene.

The rule is deliberately simple—it reads only the proprioceptive stream, requires no learned components, and is computable online in 11.6 μs per control step on a single CPU core (Section 5.5). It is nevertheless accurate: on RMA demonstrations, R5* matches 97% of the benchmark's officially annotated keyframes within ±10 frames. A pure gripper-flip rule, by contrast, misses 37.8% of official keyframes—for example, drawer-closing keyframes occur at the *start* of a motion segment, and most pouring subtasks involve no gripper flip at all, which is what motivates the motion and wrist-rotation terms.

4.4 Memory Bank and Prefix Injection

Anchor latents are appended to a fixed-capacity bank of $K = 32$ slots. The bank is *append-only with truncation*: entries are never merged or re-weighted, and when more than K anchors occur, the earliest entries are dropped. We deliberately avoid consolidation mechanisms such as token merging because they break train–inference exactness (Section 4.5).

At injection time, each stored latent is flattened and passed through a 19M-parameter linear projector into the backbone's token embedding space, producing one memory token per slot. The projector output is *zero-initialized*, so that at the start of fine-tuning the injected prefix is exactly a no-op and the policy's behavior is unchanged; empty slots are masked out of attention. For the $\pi_{0.5}$ instantiation, the K memory tokens are appended to the PaliGemma prefix after the image and language tokens (Figure 4.1), upstream of the block-causal boundary, so both the VLM's decision-relevant computation and the action expert can attend to them. This corresponds to the backbone-prefix site argued for in Section 2.4.

4.5 Train–Inference Exactness

A central design goal of DynaMem is that the memory bank a training example is built with is *exactly* the bank the policy would have at the same point of the same trajectory at inference time. Three choices combine to guarantee this (Figure 4.3):

1. **Deterministic write rule.** R5* depends only on the proprioceptive history, so the

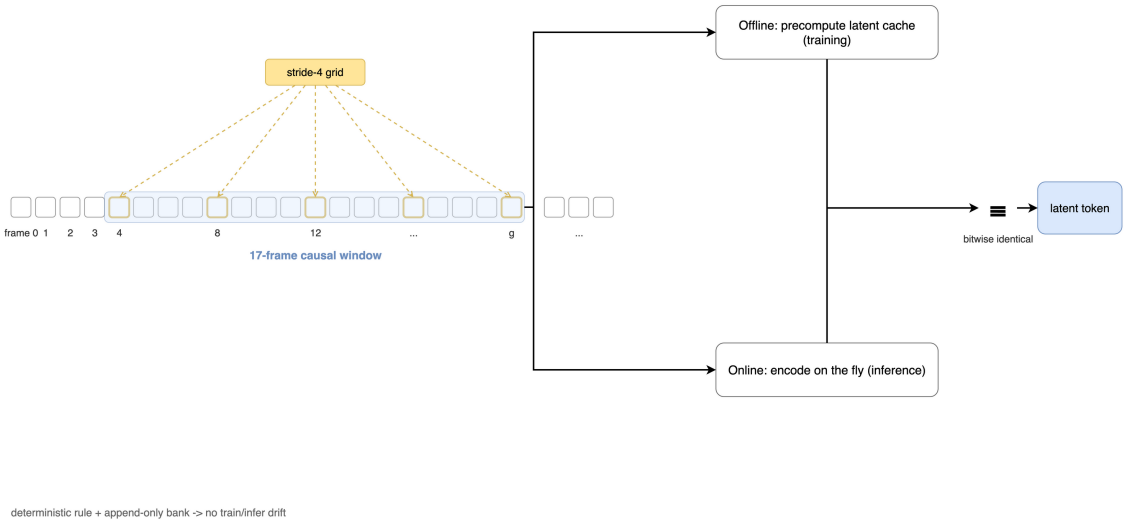

Figure 4.3: The sliding-window contract. The latent for grid frame g (temporal stride 4) is defined as the last slice of the frozen encoder applied to the 17-frame causal window ending at g. The offline cache used to build training examples and the online encoder used during closed-loop evaluation implement the same definition and produce bitwise-identical latent tokens; combined with the deterministic anchor rule and the append-only bank, the memory contents seen in training and at inference cannot drift.

set of anchors for a given trajectory prefix is a pure function of that prefix.

2. **Sliding-window encoding contract.** The offline cache (used to precompute training-time memories for all 15,097 demonstration episodes) and the online rolling-buffer encoder evaluate the same 17-frame causal window; we verify on held-out trajectories that their outputs are bitwise identical.

3. **Append-only bank.** No merging, re-ranking, or learned retrieval intervenes between writes and reads, so equality of writes implies equality of bank contents.

The property is enforced by unit tests, including an online-versus-offline equivalence test that replays full episodes through the serving-time memory wrapper and asserts bitwise equality against the training cache. By contrast, MemoryVLA trains on random

sorted frame subsets that approximate its token-merged inference bank, and keyframe methods such as PrediMem are trained with ground-truth keyframes but deployed with predicted ones (Section 2.2); in both cases the policy is never trained on the memory distribution it faces at test time. DynaMem removes this mismatch by construction, which is essential for the interpretation of the experiments in Chapter 5: any failure to exploit memory cannot be attributed to distribution shift between training and inference.

As a built-in diagnostic, the serving wrapper supports a *memory-zeroing* switch that masks all memory tokens at inference time while leaving the rest of the pipeline untouched. Comparing a checkpoint evaluated with and without its memory directly measures how much the policy actually relies on memory content, and is used throughout Chapter 5.

4.6 Integration with the Unified Interface

DynaMem is implemented as a wrapper around any `BasePolicy` instance. The wrapper intercepts the `act(obs)` call, runs the anchor detector on the proprioceptive stream, updates the rolling frame buffer and (at anchors) the memory bank, augments `obs` with the current bank, and delegates to the wrapped policy. Episode boundaries are signaled through the standard `reset()` hook, which clears the bank and the frame buffer; on the evaluation-client side this requires only a four-line patch to forward episode-boundary events. The wrapped policy requires no modification, so DynaMem composes with all backbone adapters described in Chapter 3.

Chapter 5

Experiments

5.1 Experimental Setup

5.1.1 Benchmark, Metrics, and Protocol

We evaluate on RoboMemArena (RMA) [5], described in Section 2.3. Following the benchmark's automatic evaluator, each episode is scored by its BDDL stage predicates. We report two metrics: *checkpoint success rate* (CSR), the mean fraction of completed stages per episode, and *task success rate* (TSR), the fraction of episodes achieving the full goal. Our CSR is the stage-fraction score produced by our evaluation harness; it follows the same partial-credit intent as the RMA paper's CSR but is not numerically identical to it, so we do not compare our CSR values directly against published ones.

All evaluations are closed-loop rollouts with 10 episodes per task (fixed seeds 100–109), a 3,000-step horizon, action-chunk replanning every 5 steps, and 256×256 observations. Rollouts are sharded across 4 NVIDIA RTX PRO 6000 Blackwell GPUs. Unless

stated otherwise we use an 11-task subset spanning all four families—tasks 5, 12, 17, 20, 23 (Occlusion), 6, 8, 15, 16 (Counting), and 22, 26 (Sequence and Transferring controls)—selected to over-represent memory-dependent stages while retaining two control tasks that require no memory.

5.1.2 Baseline Policy

The base policy is $\pi_{0.5}$ [10] fine-tuned with behavior cloning on the RMA demonstration set (15,097 episodes) using the openpi training stack [9], with batch size 256 for 59k steps. Validation across checkpoints showed performance plateauing from roughly 50k steps; the best checkpoint (56k) achieves a macro average of 54.5 CSR / 17.2 TSR over the 25 automatically evaluable tasks (the benchmark's task 1 requires a separate VLM-based stage evaluator and is excluded throughout). On the 11-task subset the same checkpoint scores 57.0 CSR / 11.8 TSR (Table 5.1). Stage-level inspection confirms the memory bottleneck reported by the benchmark authors: stages whose correct execution depends on history—reopening a previously visited drawer, pouring a second time—are essentially never completed (TSR = 0 on all Occlusion tasks), even though their visually identical counterparts succeed.

5.1.3 Training Configurations

All memory fine-tuning starts from a baseline checkpoint with DynaMem attached (K=32, zero-initialized projector). Memory parameters use a peak learning rate of 10^{-4} and the backbone 10^{-5}, sharing one warmup–cosine schedule (200 warmup steps, decaying to 10^{-6}). Full hyperparameters are listed in Appendix A.1.

5.2 Experiment 1: Memory Fine-Tuning from a Converged Checkpoint

The first experiment asks the headline question directly: *does plug-and-play memory injection improve a converged VLA policy?* Starting from the plateaued 56k checkpoint we train three arms for 4,000 steps each on the 11-task subset, identical in every respect except memory:

- **cont4k** — continued training without memory (K=0); controls for the effect of additional fine-tuning.
- **mem4k** — DynaMem fine-tuning with prefix injection.
- **zero4k** — the *same* mem4k checkpoint, evaluated with the memory-zeroing diagnostic of Section 4.5; measures whether mem4k actually uses its memory.

Table 5.1 reports the results, which support three findings.

Finding 1: the net memory effect is zero. mem4k differs from the cont4k control by +1.0 CSR and 0.0 TSR at the macro level—well within the noise of 10-episode evaluation, with per-task fluctuations of ±10–17 CSR canceling in both directions.

Finding 2: the memory is not used. Masking the memory of the mem4k checkpoint (zero4k) does not hurt it; it scores 3.2 CSR *higher*. Tasks on which mem4k gains over the control (12, 23) gain equally or more with memory masked, so the gains do not come from memory content. The fine-tuned policy has learned to ignore its 32 memory tokens.

Table 5.1: Three-arm comparison on the 11-task RMA subset (CSR/TSR, %; 10 episodes per task). *cont4k* is the no-memory control; *zero4k* evaluates the mem4k checkpoint with memory masked. Δ_{mc} and Δ_{mz} are the CSR differences mem4k−cont4k and mem4k−zero4k.

Task	Family	56k	cont4k	mem4k	zero4k	Δ_{mc}	Δ_{mz}
5	Occlusion	31/0	18/0	10/0	18/0	−7.8	−7.8
6	Counting	70/60	80/70	83/50	87/60	+3.3	−3.3
8	Counting	78/0	75/0	60/0	78/0	−15.0	−17.5
12	Occlusion	25/0	20/0	30/0	22/0	+10.0	+7.5
15	Counting	20/0	0/0	0/0	0/0	0.0	0.0
16	Counting	93/0	70/0	70/0	77/0	0.0	−6.7
17	Occlusion	25/0	12/0	18/0	20/0	+5.0	−2.5
20	Occlusion	52/0	48/0	50/0	50/0	+2.5	0.0
22	Sequence	80/0	77/0	67/0	67/0	−10.0	0.0
23	Occlusion	72/0	65/0	82/0	82/0	+17.5	0.0
26	Transferring	80/70	85/70	90/90	95/90	+5.0	−5.0
Macro average		57.0/11.8	49.9/12.7	50.9/12.7	54.1/13.6	+1.0	−3.2

Finding 3: most movement is fine-tuning drift. The control itself drops 7.0 CSR relative to 56k, with consistent per-task signatures across all three arms (task 15 collapses to zero, task 16 loses 16–23 points, task 5 loses 13–21), and task 26's +20 TSR appears in the zero arm as well. Subset fine-tuning from a plateaued checkpoint perturbs performance far more than memory injection does, which is precisely why the three-arm design with a continued-training control is necessary: single-arm before/after comparisons would have attributed the drift to memory.

5.3 Experiment 2: Injection-Site Probe on Visually Aliased Tasks

A possible objection to Experiment 1 is that the 56k starting point is too converged and the 11-task mixture dilutes memory-dependent supervision. The second experiment

Table 5.2: Prefix-injection probe trained from the 40k checkpoint on the two visually aliased tasks only (CSR/TSR, %; 10 episodes per task). "zero" masks the memory of the same checkpoint at inference time.

Task	Critical stage	mem	zero
5 (butter middle drawer)	*Open middle drawer again*	25.6/0	32.2/0
8 (pour sauce twice)	*Pour two*	60.0/0	57.5/0

therefore stacks the deck in memory's favor: we fine-tune from the *earlier, non-plateaued* 40k checkpoint, on only two *visually aliased* tasks in which the same observation demands different actions depending on history—task 5 ("butter middle drawer", whose stage 7 reopens a drawer already operated in stages 3–4) and task 8 ("pour tomato sauce twice", whose second pour is visually identical to the first). Training runs 2,500 steps with batch size 256 on the prefix-injection arm of the planned injection-site matrix; the expert-suffix and dual-injection arms did not complete evaluation by the time of writing and are left to future work.

Table 5.2 shows the same null signature. The memory arm is indistinguishable from its own memory-masked evaluation (-6.7 and $+2.5$ CSR on tasks 5 and 8). Stage-level inspection is more damning: the history-critical stage of task 5 (*Open middle drawer again*) is completed in 0 of 10 episodes in *both* arms, and on task 8 the second pour succeeds at nearly identical rates with memory present (7/10) and masked (6/10)—the second pour is evidently driven by visual cues such as sauce already in the target container, not by recalled history. Even under favorable conditions—earlier checkpoint, task-pure supervision, aliased observations—behavior-cloning fine-tuning did not couple the policy to its memory.

5.4 Analysis: Why Behavior Cloning Ignores Injected Memory

The two experiments, combined with the exactness guarantee of Section 4.5, rule out the usual suspects: there is no train–inference distribution shift, the injected substrate demonstrably contains the anchor history (97% keyframe coverage), and the placement is the decision-reaching prefix site. The remaining explanation is *gradient starvation at the loss level*. By 56k steps the behavior-cloning loss on expert demonstrations is already $\approx 10^{-3}$: on the training distribution, the current observation and prompt suffice to predict the expert action almost everywhere, because the demonstrations rarely visit the aliased states where history changes the correct action. The zero-initialized projector therefore starts as a no-op, receives near-zero gradient pressure, and the cheapest descent direction is to leave the memory pathway unused. The aliasing present at *evaluation* time (Experiment 2's stage-7 failures) is largely absent from the *training* signal, so no amount of architectural correctness compensates.

This analysis yields concrete design guidance for memory-augmented VLA training: (i) memory must be made *necessary in the training loss*, for example by sampling or re-weighting trajectory segments whose correct action is unpredictable without history, or by adding an auxiliary loss that supervises memory readout directly; (ii) starting from a converged behavior-cloning checkpoint is adversarial to memory adoption, and earlier or jointly trained checkpoints should be preferred; and (iii) a memory-zeroing control should be reported by any memory-augmented policy, since Experiment 1 shows that headline gains can otherwise be fine-tuning drift.

5.5 Overhead Profiling

Finally, we verify that the mechanism is cheap enough to be a practical plug-in. Measured in isolation on one RTX PRO 6000 Blackwell GPU (bfloat16, $1 \times 3 \times 17 \times 256 \times 256$ input, 30 timed calls after 3 warmup calls): one sliding-window Cosmos encode—i.e., one memory write—takes 9.76 ± 0.02 ms with a peak VRAM footprint of 151 MiB (67 MiB resident). Writes occur only at anchors, roughly 6 per subtask, so encoding contributes no per-step cost. The R5* anchor detector, which does run every control step, is pure NumPy on a single CPU core and costs 11.6 μs per step (35 ms *total* over a 3,000-step episode). The remaining per-step cost of memory is attention over K=32 additional prefix tokens against a prefix that already contains several hundred image and language tokens. DynaMem is therefore effectively free at deployment time; its open problem is not efficiency but the training-signal question of Section 5.4.

Chapter 6

Conclusions

6.1 Summary

This thesis addressed the problem of memory augmentation for Vision–Language–Action policies in robot manipulation. We made three contributions. First, we designed and implemented a unified `BasePolicy` interface that places three heterogeneous VLA backbones—$\pi_{0.5}$, Cosmos Policy, and DreamZero—behind a common rollout and evaluation protocol without modifying their internal architectures. Second, we introduced DynaMem, an input-level dynamic memory mechanism that stores frozen video-tokenizer latents at physically detected subtask anchors and injects them as backbone-prefix tokens through a zero-initialized projector. DynaMem is, to our knowledge, the first memory-augmented VLA design whose training-time and inference-time memory contents are bitwise identical by construction, eliminating the exposure bias present in prior designs. Third, we conducted a controlled study on the RoboMemArena benchmark built around two safeguards that we argue should be standard practice: a continued-training control arm and a memory-zeroing

diagnostic.

The headline empirical finding is a carefully established negative result: plug-and-play memory injection into a behavior-cloned $\pi_{0.5}$ policy is not adopted by the policy. Fine-tuned with memory from a converged checkpoint, the policy performs identically to its no-memory control, and masking its memory at inference time costs nothing—even on visually aliased tasks engineered so that history should matter. Because the mechanism's exactness guarantees rule out train–inference mismatch, the failure localizes to the training signal itself: a behavior-cloning loss that is already near zero provides no gradient pressure for a zero-initialized memory pathway. The controls proved essential: continued fine-tuning alone moved per-task scores by up to 20 points, movement that a single-arm comparison would have misattributed to memory. The mechanism itself is practically free at deployment time (under 10 ms per memory write, 12 μs per control step, and a 151 MiB peak memory-write footprint).

6.2 Limitations

Several limitations bound these conclusions. First, only the $\pi_{0.5}$ instantiation was evaluated end-to-end; the Cosmos Policy and DreamZero adapters are implemented behind the unified interface but were not trained with memory, and the expert-suffix and dual-injection arms of the injection-site matrix did not complete evaluation. Second, evaluations use 10 episodes per task, which bounds the resolvable effect size at roughly ± 10 CSR per task; macro-level conclusions rest on consistent direction across 11 tasks rather than per-task significance. Third, the negative result concerns one training recipe—behavior cloning

from RMA demonstrations—and does not establish that prefix-injected memory cannot work under other losses. Fourth, all results are in simulation.

6.3 Future Work

The mechanistic analysis points to a concrete agenda. The immediate next step is to make memory necessary in the loss: re-weighting aliased segments of the demonstrations, or adding an auxiliary readout loss that supervises the memory pathway directly, before any architectural iteration. Completing the injection-site matrix (expert-suffix and dual injection) from a common early checkpoint would close the placement question that Experiment 2 opened. Because the memory substrate is the native latent space of video and world models, extending DynaMem to the Cosmos Policy and DreamZero backbones through the same interface is a natural test of the backbone-agnostic design. Finally, the train–inference-exact machinery and the memory-zeroing diagnostic are independent of our specific design and could be adopted as evaluation hygiene by future memory-augmented VLA work.

Bibliography

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

[2] Aydar Bulatov, Yury Kuratov, and Mikhail S. Burtsev. Recurrent memory transformer. In *Advances in Neural Information Processing Systems*, 2022.

[3] Moo Jin Kim et al. OpenVLA: An open-source vision-language-action model. *arXiv preprint arXiv:2406.09246*, 2024.

[4] Moo Jin Kim et al. Cosmos policy: Fine-tuning video models for visuomotor control and planning. *arXiv preprint arXiv:2601.16163*, 2026.

[5] Huashuo Lei et al. RoboMemArena: A comprehensive and challenging robotic memory benchmark. *arXiv preprint arXiv:2605.10921*, 2026.

[6] Bo Liu et al. LIBERO: Benchmarking knowledge transfer for lifelong robot learning. In *Advances in Neural Information Processing Systems*, 2023.

[7] Oier Mees et al. CALVIN: A benchmark for language-conditioned policy learning for long-horizon robot manipulation tasks. *IEEE Robotics and Automation Letters*, 2022.

[8] NVIDIA. Cosmos world foundation model platform for physical AI. *arXiv preprint arXiv:2501.03575*, 2025.

[9] Physical Intelligence. openpi: Open-source models and tooling for the π vision-language-action model family. `https://github.com/Physical-Intelligence/openpi`, 2025.

[10] Physical Intelligence et al. $\pi_{0.5}$: A vision-language-action model with open-world generalization. *arXiv preprint arXiv:2504.16054*, 2025.

[11] RLinf Contributors. RLinf: Scalable reinforcement learning infrastructure for embodied AI. `https://github.com/RLinf/RLinf`, 2025.

[12] Hao Shi et al. MemoryVLA: Perceptual-cognitive memory in vision-language-action models for robotic manipulation. *arXiv preprint arXiv:2508.19236*, 2025.

[13] Greg Wayne et al. Unsupervised predictive memory in a goal-directed agent. *arXiv preprint arXiv:1803.10760*, 2018.

[14] Seonghyeon Ye et al. World action models are zero-shot policies. *arXiv preprint arXiv:2602.15922*, 2026.

Appendix A

Supplementary Material

A.1 Full Hyperparameters

Table A.1: DynaMem memory-module hyperparameters used in all experiments.

Parameter	Value	Description
Memory capacity K	32	Append-only slots; earliest dropped beyond K
Substrate encoder	Cosmos CV4×8×8	Frozen causal video tokenizer
Entry shape	$16 \times 32 \times 32$	Latent per anchor (256×256 input)
Encoding window	17 frames	Causal window ending at grid frame (stride 4)
Projector	19M params, linear	Zero-initialized output
Injection site	PaliGemma prefix	After image and language tokens
Anchor rule	R5*	Gripper flip $\cup$ motion stop/start $\cup$ wrist onset
Anchor debounce	5 frames	Per event channel
Anchor obs. offset	+5 frames	Stored frame trails event

Table A.2: Training configurations. All runs use behavior cloning with the openpi stack on 4×NVIDIA RTX PRO 6000 Blackwell GPUs.

Parameter	Exp. 1 (three-arm)	Exp. 2 (probe)
Initial checkpoint	baseline 56k (plateaued)	baseline 40k
Task set	11-task subset	tasks {5, 8}
Steps	4,000	2,500
Batch size	256	256
Backbone peak LR	10^{-5}	10^{-5}
Memory-params peak LR	10^{-4}	10^{-4}
Schedule	200-step warmup, cosine decay to 10^{-6}	
Step time	≈7.8 s/iteration	

Table A.3: Evaluation protocol for RoboMemArena.

Parameter	Value	Description
Episodes per task	10	Fixed seeds 100–109
Horizon	3,000 steps	Before timeout
Replanning interval	5 steps	Action-chunk re-query
Observation	256×256 RGB	Agent-view camera
Evaluable tasks	25 of 26	Task 1 requires a separate VLM evaluator
Sharding	4 GPUs	RTX PRO 6000 Blackwell

A.2 Additional Results

Stage-level results for the injection-site probe (Section 5.3): on task 5, the history-critical stage *07 Open Middle Drawer Again* is completed in 0/10 episodes in both the memory and memory-masked arms; the preceding memoryless stages 01–04 succeed in the majority of episodes, confirming that the failure is specific to the aliased re-visit. On task 8, stage *03 Pour Two* completes in 7/10 (memory) versus 6/10 (masked) episodes, and the terminal stage *04 Place Bowl Drainer* in 0/10 for both, which accounts for the zero TSR.

A.3 Reproducibility

All training and evaluation code lives on a dedicated branch of our openpi fork (`litian/memory-v1`); training curves are logged to Weights & Biases (run `pi05_rma_mem_4k`). The training-time memory cache is validated against the online encoder by an automated test suite, including a bitwise online-versus-offline equivalence test over full replayed episodes, and the demonstration-to-simulation index mapping is verified for all 15,097 episodes. Two practical notes for exact reproduction: the Cosmos tokenizer is bitwise reproducible only at batch size 1 with at least 3 warmup calls per input shape, and legal clip lengths are $1 + 4k$ frames. GPU kernels are not bit-level reproducible across driver versions, so benchmark numbers are reported as means over 10 fixed-seed episodes per task.