THE WORLD DOES NOT PAUSE: REAL-TIME BENCHMARKS AND TRAINING-FREE ACCELERATION FOR DYNAMIC MANIPULATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Dynamic manipulation requires timely actions as the world continues to evolve during policy inference. In vision-language-action (VLA) and world action models (WAM), inference latency can leave actions conditioned on outdated observations. We introduce **Dynamic-LIBERO** and **Dynamic-RoboTwin**, which combine controlled target motion with inference-time world evolution while largely preserving established task goals for frozen-policy evaluation. We propose **PACE** (Policy Acceleration through Cached Execution), a training-free runtime that reuses observation representations and action computation across replanning calls and integration steps without policy fine-tuning. Across two simulated benchmarks and two policies, PACE accelerates inference by up to $3.8\times$ and improves dynamic task success by up to 42.8 percentage points over standard deployments, achieving the highest success in all eight dynamic settings. On a physical robot, PACE reduces per-chunk inference stalls from 159 to 69 ms, maintains or improves static-task success, shortens completion times, and increases dynamic success counts. Together, these results highlight that reducing inference latency is critical for successful dynamic manipulation.

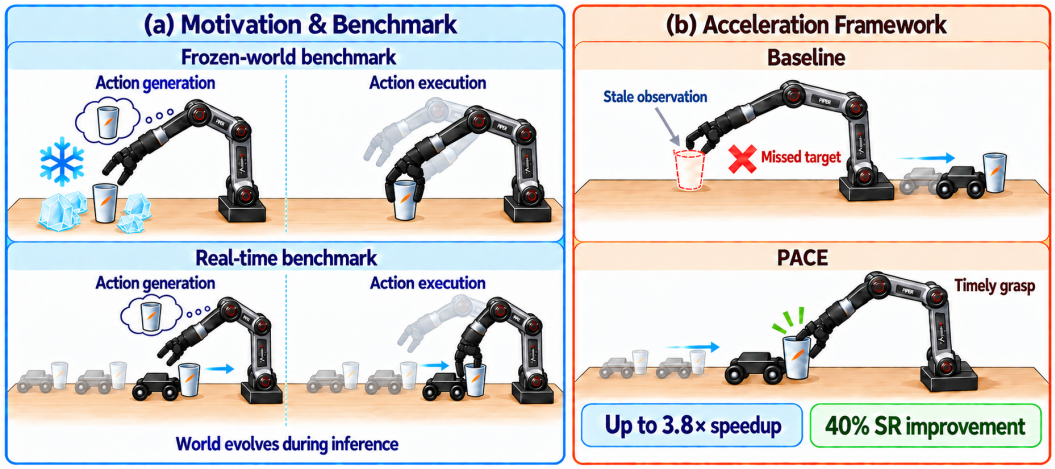

Figure 1: Overview of our motivation and approach. (a) Our benchmarks let the world evolve during policy inference. (b) PACE reduces inference latency to support timely actions on moving targets; the reported task success rates are for $\pi_{0.5}$ on Dynamic-LIBERO at v_2.

1 INTRODUCTION

The world does not pause while a robot decides how to act. Grasping an object moving across a table or placing it into a dynamic container requires actions that remain appropriate at execution time. Vision-language-action (VLA) and world action models (WAM) provide expressive multimodal policies, but their inference latency separates observation from execution (Black et al., 2025; Yuan et al., 2026). During this interval, targets can move and previously issued robot actions can

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

continue, leaving new actions conditioned on an outdated scene. Dynamic manipulation requires evaluating action quality and delivery timing (Figure 1).

Recent work develops motion-aware tasks and responsive policies (Xie et al., 2026; Chen et al., 2026), latency-aware execution (Black et al., 2025; Zhu et al., 2026), and inference caching (Xu et al., 2025; Zhao et al., 2026). These advances motivate evaluating controlled target motion together with inference-time world evolution. Pausing the world during inference hides observation–execution mismatch, while varying target motion alone does not characterize the effects of action delivery timing. We ask: *how does reducing inference latency improve dynamic manipulation for frozen VLA/WAM policies?* Answering this question requires comparing runtimes under common tasks, motion conditions, and execution rules.

We introduce **Dynamic-LIBERO** and **Dynamic-RoboTwin**, extending established manipulation tasks (Liu et al., 2023; Mu et al., 2025) with controlled target motion and inference-time world evolution while largely preserving task goals and interfaces. Their 50 and 45 variants cover dynamic grasping, placement, and combined interactions. Paired scene conditions and delivery-delay interventions support evaluation of existing policies without additional fine-tuning. We also propose **PACE** (Policy Acceleration through Cached Execution), a training-free runtime combining representation and action-residual reuse across replanning calls with velocity reuse within action chunks. Change-gated reuse bypasses redundant computation while preserving weights and action interfaces.

Across FastWAM and $\pi_{0.5}$ on both simulated benchmarks, PACE accelerates inference by up to $3.8\times$, improves dynamic task success by up to 42.8 percentage points, and achieves the highest success in all eight dynamic settings. On a physical Piper robot, PACE reduces per-chunk inference stalls from 159 to 69 ms, shortens task completion times, and improves dynamic success counts. Speed sweeps characterize performance under increasing motion, while added-delay experiments demonstrate that waiting can reduce success at fixed runtime settings. These results highlight the importance of reducing inference latency for dynamic manipulation.

2 Related Work

Manipulation Benchmarks. RLBench and LIBERO benchmark multi-task, language-conditioned manipulation (James et al., 2020; Liu et al., 2023); RoboTwin expands dual-arm and environmental coverage (Mu et al., 2025; Chen et al., 2025). DOM tests moving objects (Xie et al., 2026), RTC studies delay in Kinetix (Black et al., 2025), and ReflexBench advances dynamics during inference (Chen et al., 2026). Our benchmarks combine controlled motion and inference-time world evolution while largely preserving established task goals and interfaces for frozen-policy evaluation.

Latency-Aware Policy Execution. Concurrent control studies decisions in evolving systems (Xiao et al., 2020). RTC conditions on committed actions (Black et al., 2025), VLASH predicts execution-time states (Tang et al., 2025), REMAC uses masked chunk training (Wang et al., 2026), and ARLI incorporates intermediate observations and committed actions (Zhu et al., 2026). Comparisons examine delay, chunk length, and delay exposure (Agouzoul, 2026). We study runtime and delivery-delay effects under a shared executor and controlled target motion.

Inference Acceleration. Training-based methods use policy distillation (Prasad et al., 2024; Wang et al., 2024) or learned visual caching (Wei et al., 2026); Fast-WAM omits future-video generation (Yuan et al., 2026). Training-free approaches include DPM-Solver's numerical sampling (Lu et al., 2022), DeepCache's feature reuse (Ma et al., 2024), and TaylorSeer's extrapolation (Liu et al., 2025). VLA-Cache reuses visual tokens (Xu et al., 2025), BAC schedules action-feature reuse (Ji et al., 2025), and C^{3}ache caches cross-chunk WAM residuals (Zhao et al., 2026). PACE combines representation, residual, and velocity reuse, evaluating both latency and dynamic task performance.

3 Problem Formulation

Following delayed action-chunk execution (Black et al., 2025; Zhu et al., 2026), we study frozen policies under a shared asynchronous executor. The lower panel of Figure 2 illustrates the timing.

Action chunks and delay. At replanning call i, a policy with frozen weights θ and runtime configuration χ maps multimodal input I_i, cache state C_i, and noise ϵ_i to an H-step action chunk:

$$(A_i, C_{i+1}) = \Pi_{\theta,\chi}(I_i, C_i, \epsilon_i), \qquad A_i = (a_{i,0}, \ldots, a_{i,H-1}). \tag{1}$$

Let τ_i^{obs}, τ_i^{ready}, and τ_i^{exec} denote world times of observation, executor availability, and first action application. Availability, queueing, and observation-to-action delays satisfy

$$\delta_i = \tau_i^{\text{ready}} - \tau_i^{\text{obs}}, \quad q_i = \tau_i^{\text{exec}} - \tau_i^{\text{ready}}, \quad L_i^{\text{act}} = \delta_i + q_i. \tag{2}$$

Availability delay includes more than model computation; queueing includes control-tick alignment.

A continuously evolving world. While a chunk is pending, the executor applies previous actions or its fixed fallback. For robot–object state s_n and external motion ξ_n,

$$s_{n+1} \sim P_{\mathcal{B}}(\cdot \mid s_n, a_n, \xi_n), \qquad n = n_i, \ldots, e_i - 1, \tag{3}$$

where n_i and e_i index observation and first execution. Thus, the state at execution can differ from the policy input even for stationary targets. We evaluate task success and measured latency at fixed executor settings while varying the inference runtime and commanded target speed. Additional delivery-delay interventions hold the runtime fixed (Appendix B.3).

4 Dynamic Manipulation Benchmarks

Dynamic-LIBERO and Dynamic-RoboTwin extend LIBERO and RoboTwin (Liu et al., 2023; Mu et al., 2025) with controlled target motion while largely preserving task goals, target identities, and success criteria, enabling frozen-policy evaluation without additional fine-tuning. Tasks are selected for semantic validity and scene feasibility, independently of policy performance. Dynamic-LIBERO comprises 50 tasks from ten base skills, each with three linear variants (pick, place, both) and two curved variants (pick, place); Dynamic-RoboTwin comprises 45 tasks from 15 skills with three linear variants each. Pick, place, and both variants test dynamic object acquisition, placement relative to a dynamic target, and coordination of both interactions, respectively. Parameterized speeds and trajectories, including curved paths in Dynamic-LIBERO, vary temporal difficulty and motion geometry. Visible cars and carriers provide transport context across single-arm and dual-arm tasks (Figure 2); extensions that change instructions or introduce additional skills are excluded from the main suite.

During inference and subsequent queueing, the shared executor applies available actions or its fallback while external target motion continues (Section 3). Measured-delay evaluation captures deployed runtime effects, while delivery-delay interventions study timing at a fixed runtime configuration. Original, Modified-only, and Static conditions separate scene modification from inference-time world evolution; Moving conditions additionally enable target motion. This protocol connects inference efficiency to closed-loop task performance across VLA/WAM policies. Construction and comparison details appear in Appendix A.

5 Inference Acceleration

PACE (Policy Acceleration through Cached Execution) accelerates frozen VLA/WAM policies through reuse across replanning calls and within an action chunk. Building on representation and feature caching (Xu et al., 2025; Ma et al., 2024; Zhao et al., 2026), it places change-gated shortcuts along the inference path (Figure 3). All weights remain fixed, and no gate requires training.

5.1 Reusing Observation Representations

Let x_i be the multi-camera input and E_θ its encoder. Given a cached representation V_j and reference input x_j, PACE uses

$$\widehat{V}_i = \begin{cases} V_j, & b_i^{\text{vis}} = 1 \text{ and } d_{\text{img}}(x_i, x_j) < \eta_{\text{vis}}, \\ E_\theta(x_i), & \text{otherwise.} \end{cases} \tag{4}$$

Here d_{img} measures relative image change and b_i^{vis} requires a valid cache entry. A miss refreshes the representation and its reference image. FastWAM also reuses the text representation when the instruction is unchanged. We denote the resulting visual and language conditioning together with current robot state by c_i.

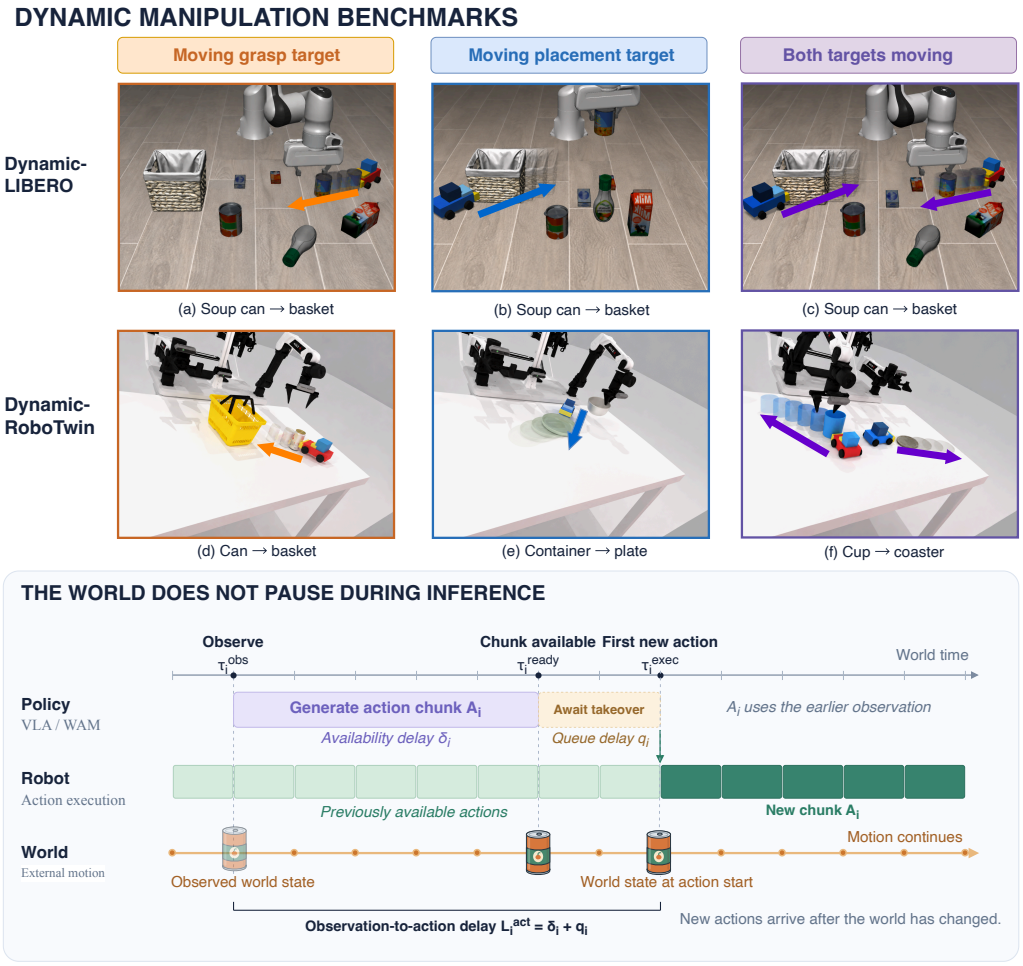

Figure 2: Benchmark card for Dynamic-LIBERO and Dynamic-RoboTwin. Task examples and timing details are shown.

5.2 Reusing Action Computation

Within call i, action latent $z_{i,k}$ evolves over integration coordinates $\sigma_{i,k}$, distinct from world time:

$$z_{i,k+1} = z_{i,k} + \Delta\sigma_{i,k}\,\widetilde{u}_{i,k}, \qquad \Delta\sigma_{i,k} = \sigma_{i,k+1} - \sigma_{i,k}. \tag{5}$$

Starting from sampled noise, every integration step is retained, and the terminal latent uses the original action interface. PACE reduces the computation needed to obtain the velocity $\widetilde{u}_{i,k}$.

Residual reuse across calls. Write hidden transformation as $h^{\text{out}}_{i,k} = F_\theta(h^0_{i,k}; c_i, \sigma_{i,k})$ and current output head as $P_{i,k}(h) = P_\theta(h; c_i, \sigma_{i,k})$. A compatible entry $r = (j, \ell)$ from an earlier call stores

$$R_r = F_\theta(h^0_{j,\ell}; c_j, \sigma_{j,\ell}) - h^0_{j,\ell}, \qquad \widehat{h}^{\text{out}}_{i,k} = h^0_{i,k} + R_r. \tag{6}$$

When both visual and latent changes pass their gates ($g^{\text{res}}_{i,k} = 1$), the cached residual replaces the hidden transformation while the current output head remains active. Caching the residual preserves the current input embedding $h^0_{i,k}$, reusing only the hidden transformation from an earlier call. The active output head then predicts velocity using the current conditioning and integration coordinate, rather than replaying a previously generated action chunk.

Velocity reuse within a chunk. Let $\kappa < k$ be the most recent velocity-evaluation step. If the latent change from $z_{i,\kappa}$ is small and the cache is admissible ($g^{\text{step}}_{i,k} = 1$), PACE holds that velocity. The

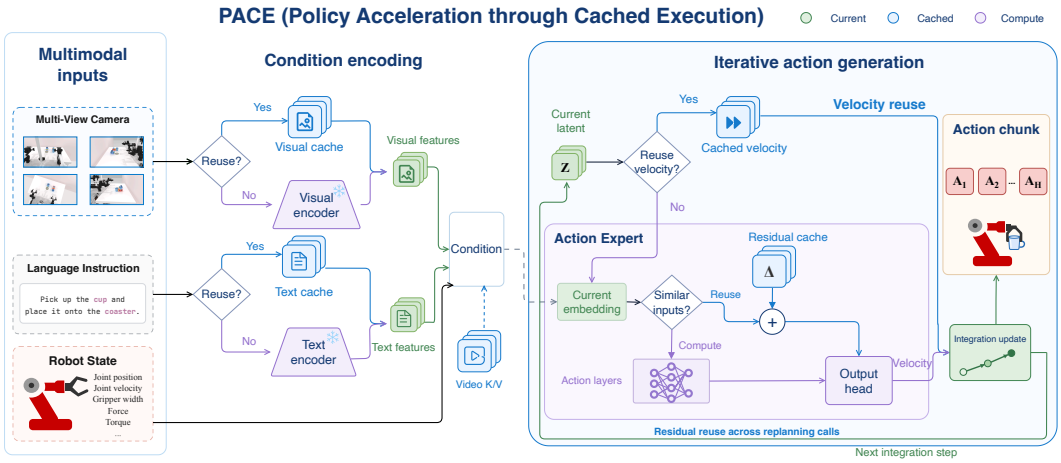

Figure 3: PACE along the VLA/WAM inference path. Visual reuse bypasses encoding; residual reuse bypasses action layers but retains the output head; velocity reuse bypasses both. Every branch advances integration. Green, blue, and purple denote current quantities, cached quantities, and computation.

ordered choice is

$$\widetilde{u}_{i,k} = \begin{cases} \widetilde{u}_{i,\kappa}, & g_{i,k}^{\text{step}} = 1, \\ P_{i,k}(h_{i,k}^{0} + R_r), & g_{i,k}^{\text{step}} = 0,\ g_{i,k}^{\text{res}} = 1, \\ u_\theta(z_{i,k}, \sigma_{i,k}; c_i), & \text{otherwise}, \end{cases} \tag{7}$$

where the full evaluation is $u_\theta(z_{i,k}, \sigma_{i,k}; c_i) = P_{i,k}(h_{i,k}^{\text{out}})$. Velocity reuse skips both hidden layers and the output head; otherwise, a newly evaluated velocity refreshes the within-chunk cache. Holding the full velocity vector preserves an integration update, not a network execution. Gate definitions and cache management appear in Appendix C.

5.3 Runtime Composition and Instantiation

Cache hits bypass the corresponding encoder or action-network computation, while misses evaluate it. Velocity reuse is checked before residual reuse. PACE unifies reuse of observation representations, cross-replanning hidden transformations, and within-chunk velocities through ordered change gates, jointly reducing computation at three levels while retaining the policy's weights, integration grid, and action interface. Reuse is approximate: small input changes do not guarantee identical outputs or task success. We therefore assess both latency and closed-loop performance. Both instantiations combine visual, residual, and velocity reuse; FastWAM additionally caches instruction representations and video tokens. Backbone-specific instantiations are summarized in Appendix C.

6 Experiments

6.1 Experimental Setup

Evaluation settings. We evaluate FastWAM and $\pi_{0.5}$ with frozen weights and no additional fine-tuning. Each task variant uses paired initializations across methods, with matched external-motion random seeds. Task success is averaged within each task and then equally across tasks. Within each model–benchmark comparison, methods share the checkpoint and asynchronous execution protocol. Simulation accounts for measured inference time through delay replay (Appendix B).

Baselines. Baseline uses the full-computation reference implementation with the original number of function evaluations (NFE). **TaylorSeer** applies the existing feature-extrapolation acceleration method to each backbone (Liu et al., 2025). **C^{3}ache** is evaluated with FastWAM (Zhao et al., 2026), and **AdaVLA**'s sampler with $\pi_{0.5}$ (Han et al., 2026). **PACE (Ours)** combines the reuse mechanisms described in Section 5; adaptations and enabled components are specified in Appendix C.5.

Metrics. Benchmark validation compares the original task with a modified scene under paused-world inference, then evaluates that same stationary modified scene while inference time advances the world. Acceleration evaluation advances world time using the measured latency of each replanning call. We report task success and measured replanning latency, with grasp retention as an auxiliary outcome. Stationary and moving-scene results are reported separately; success-rate differences are expressed in percentage points. Construction and execution details are in Appendix A.

Table 1: Original: unmodified scene; Modified-only: modified stationary scene, both with inference-time world pause. Static: the same modified scene with world time advancing during inference. Red values denote percentage-point differences from Original.

Policy	**Dynamic-LIBERO (%)**			**Dynamic-RoboTwin (%)**		
	Original	Modified-only	Static	Original	Modified-only	Static
FastWAM	99.0	91.2 (-7.8)	0.4 (-98.6)	70.0	67.6 (-2.4)	64.0 (-6.0)
$\pi_{0.5}$	99.8	95.8 (-4.0)	95.0 (-4.8)	46.0	40.4 (-5.6)	38.7 (-7.3)

6.2 Benchmark Validation

Paired conditions separate scene and timing effects (Table 1). Original and Modified-only differ in scene construction with the world paused during inference; Static lets the same stationary modified scene and executor advance during computation. Policies, tasks, initializations, and action settings remain fixed.

Scene modification lowers success by 2.4–7.8 percentage points across policy–benchmark pairs. Enabling inference-time world evolution reduces FastWAM's Dynamic-LIBERO success from 91.2% to 0.4% (90.8 points), versus 0.8 points for $\pi_{0.5}$. A likely mechanism is that FastWAM's long inference delay allows queued actions to substantially change the robot state before the new chunk executes, creating an observation–execution mismatch that paused-world evaluation masks even with stationary targets. On Dynamic-RoboTwin, the decreases are 3.6 and 1.7 points, respectively. Thus, continuous-world execution can cause substantial, policy-dependent failures even with stationary targets. This contrast measures world evolution and execution during inference; target-motion results follow in Section 6.3.

6.3 Main Results

PACE achieves the lowest mean inference latency in all four policy–benchmark combinations and the highest task-success point estimate in all eight moving-target settings (Table 2). Relative to Baseline, inference is 2.86–3.76$\times$ faster. The largest success gain occurs for FastWAM on Dynamic-LIBERO, where latency falls from 598.6 to 159.1 ms and success at v_1 rises from 0.2% to 43.0%. For $\pi_{0.5}$, which retains stronger unaccelerated performance on the same benchmark, PACE raises v_2 success from 45.2% to 65.9% (a 20.7-point gain) while reducing latency from 320.7 to 97.3 ms (3.30$\times$ faster).

Existing accelerators exhibit substantial variation across settings. For FastWAM on Dynamic-LIBERO, TaylorSeer and C^{3}ache reduce latency to 378.0 and 268.1 ms, respectively, but neither exceeds 4.8% success at either moving speed. PACE reaches 43.0% and 27.8%. For $\pi_{0.5}$, PACE exceeds AdaVLA by 5.4 and 7.3 points at v_1 and v_2 with lower latency. On Dynamic-RoboTwin, AdaVLA accelerates $\pi_{0.5}$ to 124.4 ms but lowers v_1 success from 30.4% to 17.8%; PACE reaches 44.4% at 82.3 ms. Thus, reducing inference time does not uniformly preserve task performance.

With PACE, increasing commanded target speed from 0 to 3.0 cm/s substantially reduces success on both benchmarks, while mean model latency shows no monotonic increase (Figure 4). The relative policy ranking also depends on the benchmark: $\pi_{0.5}$ leads on Dynamic-LIBERO, whereas FastWAM leads on Dynamic-RoboTwin. Faster inference therefore does not eliminate the difficulty of increasing target motion. The sweep measures robustness across speeds; it does not isolate latency from task difficulty. Separate delivery-delay interventions confirm that additional waiting can reduce success at fixed runtime, with sensitivity varying across policies and speeds. Sweep settings and delivery-delay results appear in Appendix B.3.

Table 2: Inference latency and task success under measured-delay evaluation. Bold denotes the best value within each policy and benchmark.

Policy	method	Latency ↓ (ms)	Task success (%) ↑		
			Static	Moving v_1	Moving v_2
Dynamic-LIBERO					
FastWAM	Baseline	598.6	0.4	0.2	0.8
	C^{3}ache	268.1	12.0	4.0	4.0
	TaylorSeer	378.0	6.2	4.8	3.3
	PACE (Ours)	**159.1**	**82.4**	**43.0**	**27.8**
$\pi_{0.5}$	Baseline	320.7	95.0	69.6	45.2
	AdaVLA	153.8	**96.2**	75.8	58.6
	TaylorSeer	239.3	94.0	72.8	53.8
	PACE (Ours)	**97.3**	94.2	**81.2**	**65.9**
Dynamic-RoboTwin					
FastWAM	Baseline	300.7	64.0	47.1	31.5
	C^{3}ache	221.1	62.2	45.6	30.0
	TaylorSeer	205.8	66.0	52.9	41.6
	PACE (Ours)	**105.1**	**68.0**	**54.0**	**43.3**
$\pi_{0.5}$	Baseline	299.3	38.7	30.4	20.7
	AdaVLA	124.4	28.9	17.8	22.2
	TaylorSeer	216.7	40.0	28.7	24.4
	PACE (Ours)	**82.3**	**46.2**	**44.4**	**28.9**

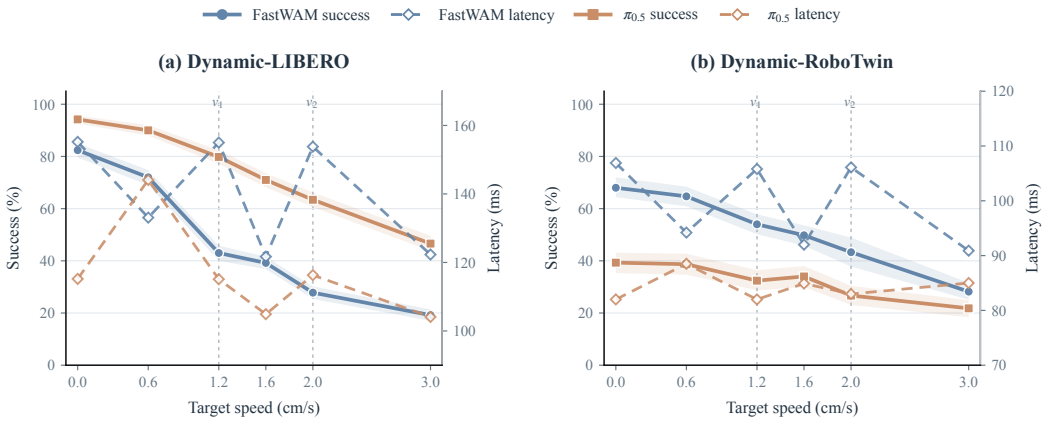

Figure 4: PACE across target speeds: task success (solid, left axis; shaded 95% task-stratified bootstrap interval) and mean model latency (dashed, right axis).

6.4 Ablation Analysis

Removing any reuse component from FastWAM increases latency and reduces success (Table 3). Disabling visual or residual reuse adds 48.6 or 85.8 ms and lowers v_1 success by 7.8 or 15.0 points, respectively. Removing velocity reuse has the largest effect: latency rises from 159.1 to 382.7 ms, while success falls by 31.7 and 20.4 points at v_1 and v_2. All three components therefore contribute to FastWAM's observed runtime–success tradeoff.

For $\pi_{0.5}$, removing velocity reuse raises latency from 97.3 to 229.1 ms and lowers dynamic success by 7.8/11.8 points at v_1/v_2. Removing visual or residual reuse adds 13.9 or 12.2 ms and lowers success by 1.4/1.6 or 2.9/2.5 points, respectively; Static success also decreases in every ablation. Thus, all three reuse components improve the observed latency and success for both backbones,

Table 3: Ablations of PACE on Dynamic-LIBERO. Bold denotes the best value within each policy.

Policy	Configuration	Latency ↓ (ms)	Task success (%) ↑		
			Static	Moving v_1	Moving v_2
FastWAM	**PACE**	**159.1**	**82.4**	**43.0**	**27.8**
	w/o Visual Reuse	207.7	65.3	35.2	17.6
	w/o Residual Reuse	244.9	55.1	28.0	18.8
	w/o Velocity Reuse	382.7	21.2	11.3	7.4
$\pi_{0.5}$	**PACE**	**97.3**	**94.2**	**81.2**	**65.9**
	w/o Visual Reuse	111.2	94.0	79.8	64.3
	w/o Residual Reuse	109.5	94.0	78.3	63.4
	w/o Velocity Reuse	229.1	93.2	73.4	54.1

with velocity reuse contributing the largest gains. Full PACE achieves the lowest latency and highest success across all conditions, supporting the combined reuse design.

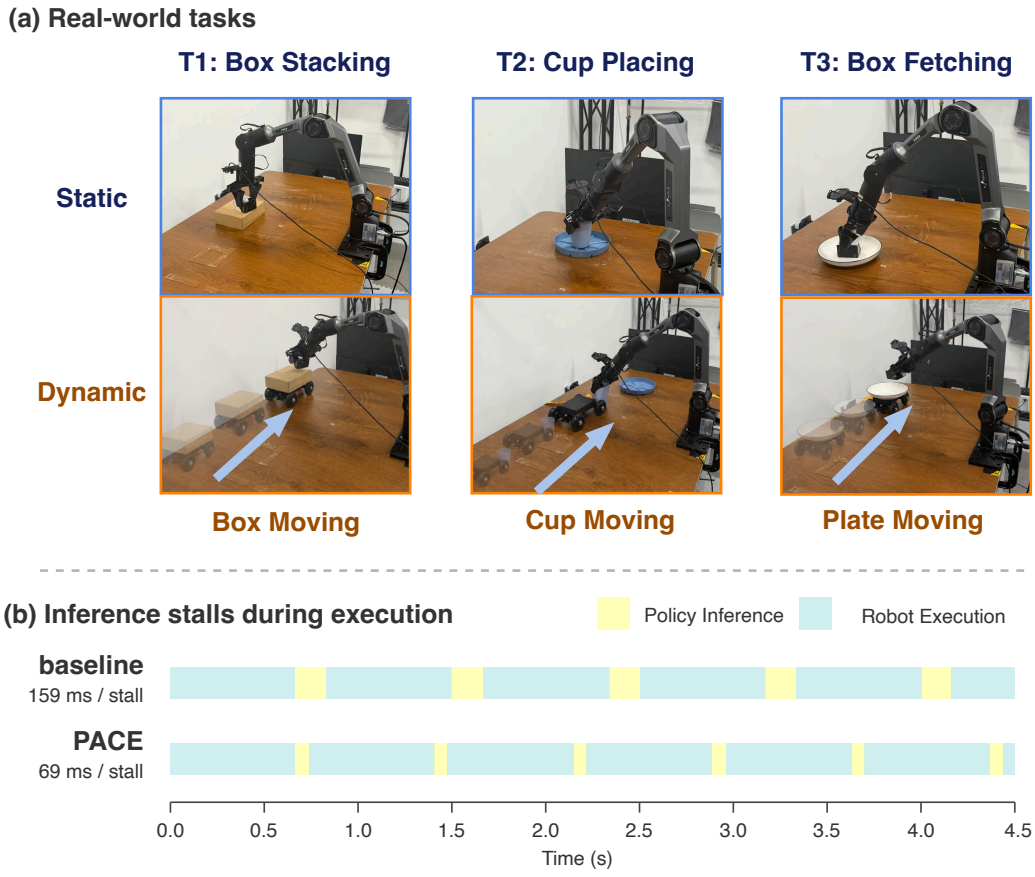

Figure 5: Real-world evaluation. (a) Static and dynamic variants of T1–T3; trails and arrows indicate target motion. (b) Robot execution (cyan) alternates with inference stalls (yellow), shortened by PACE from 159 to 69 ms per chunk (56.6%).

6.5 Real-World Experiments

We evaluate $\pi_{0.5}$ on an AgileX Piper arm using three tasks: T1: Box Stacking, T2: Cup Placing, and T3: Box Fetching, each with static and dynamic variants (Figure 5a). Baseline and PACE share the same fine-tuned weights and synchronous executor, differing only in caching and computation

Table 4: Real-robot results. Success is reported as successful trials / total trials throughout; Δt is PACE's completion-time change over successful static-task trials. Dynamic results use $H = 10$ and are reported at each commanded cart speed.

(a) Static targets: success and completion time

Horizon	Task	Baseline success	PACE success	PACE time change (Δt)
$H = 10$	T1	10/10	10/10	-19.6%
	T2	10/10	10/10	-8.8%
	T3	9/10	10/10	-3.6%
$H = 5$	T1	6/10	6/10	-10.2%
	T2	6/10	6/10	-27.0%
	T3	7/10	7/10	-18.8%

(b) Dynamic tasks ($H = 10$): success by cart speed

Task	Method	1.5 cm/s	2.5 cm/s	3.5 cm/s
T1	Baseline	2/10	1/10	0/10
	PACE	**4/10**	**2/10**	**1/10**
T2	Baseline	3/10	1/10	0/10
	PACE	**5/10**	**2/10**	**1/10**
T3	Baseline	2/10	1/10	0/10
	PACE	**3/10**	1/10	**1/10**

reuse. We run 10 trials per method and condition. Appendix D shows full hardware and protocol details.

Under synchronous execution, the robot waits for each new action chunk while external targets can continue moving. The timelines in Figure 5b illustrate how PACE reduces the observed per-chunk stall from 159 to 69 ms (56.6%), increasing the frequency of action updates. In a separate replay of 123 held-out calls, median inference latency falls from 165.1 to 79.0 ms, with comparable action error (0.0353 for PACE versus 0.0356 rad for Baseline). Thus, the runtime reduction also shortens physical waiting periods while preserving the measured action fidelity. The observed speedup comes from denoising-step reuse and batched image encoding.

At $H = 10$, PACE achieves 10/10 static success on all three tasks, matching Baseline on T1 and T2 and improving T3 from 9/10, while reducing successful-trial completion time by 19.6%, 8.8%, and 3.6%, respectively (Table 4). These gains are smaller than the stall reduction because robot motion occupies much of execution. At $H = 5$, T2 retains 6/10 success with a 27.0% time reduction, consistent with more frequent replanning increasing the waiting share (Appendix D.1). In dynamic tasks at $H = 10$ and 1.5 cm/s, success rises from 20% to 40% on T1, 30% to 50% on T2, and 20% to 30% on T3. At 3.5 cm/s, PACE succeeds in 1/10 trials on each task versus 0/10 for Baseline. Success counts match or exceed Baseline across conditions.

7 Conclusion

Dynamic manipulation requires evaluating policies while the world continues to evolve during inference. Dynamic-LIBERO and Dynamic-RoboTwin expose this timing challenge, while PACE addresses it through training-free reuse across the VLA/WAM inference path. In simulation, PACE reduces latency by up to 73.4% and improves moving-target success by up to 42.8 percentage points. Physical experiments further show shorter inference stalls and improved task efficiency and dynamic success counts. Speed sweeps and delivery-delay interventions highlight that timely action availability matters, but acceleration alone does not resolve high-speed tracking failures. These results motivate evaluating inference efficiency together with dynamic task performance.

AI USE STATEMENT

We used large language models to brainstorm research ideas, obtain feedback on the method and experimental design, and assist in interpreting results. They also helped translate, draft, and polish manuscript text and retrieve and summarize related work. AI-assisted code editors, including Cursor, were used to implement and edit research code. Generative AI tools also assisted in designing and editing figures in the paper. The authors reviewed the AI-assisted text, code, analyses, citations, and figures, verified the reported claims and results, and take responsibility for the final submission.

REFERENCES

Ayoub Agouzoul. Understanding asynchronous inference methods for vision-language-action models. *arXiv preprint arXiv:2605.08168*, 2026.

Kevin Black, Manuel Galliker, and Sergey Levine. Real-time execution of action chunking flow policies. In *Advances in Neural Information Processing Systems*, volume 38, pp. 33383–33407, 2025.

Tianxing Chen, Zanxin Chen, Baijun Chen, Zijian Cai, Yibin Liu, Zixuan Li, Qiwei Liang, Xianliang Lin, Yiheng Ge, Zhenyu Gu, et al. Robotwin 2.0: A scalable data generator and benchmark with strong domain randomization for robust bimanual robotic manipulation. *arXiv preprint arXiv:2506.18088*, 2025.

Yuxuan Chen, Wanruo Zhang, and Xiao Li. Reflex: Enabling fast and predictive vision-language-action models for reaction-critical manipulation. *arXiv preprint arXiv:2608.14379*, 2026.

Sunghwan Han, Youngtae Han, and Youngmin Yi. Adavla: Adaptive step flow matching for training-free acceleration of vision-language-action models. *arXiv preprint arXiv:2608.29208*, 2026.

Stephen James, Zicong Ma, David Rovick Arrojo, and Andrew J Davison. Rlbench: The robot learning benchmark & learning environment. *IEEE Robotics and Automation Letters*, 5(2):3019–3026, 2020.

Kangye Ji, Yuan Meng, Hanyun Cui, Ye Li, Jianbo Zhou, Shengjia Hua, Lei Chen, and Zhi Wang. Block-wise adaptive caching for accelerating diffusion policy. *arXiv preprint arXiv:2506.13456*, 2025.

Bo Liu, Yifeng Zhu, Chongkai Gao, Yihao Feng, Qiang Liu, Yuke Zhu, and Peter Stone. Libero: Benchmarking knowledge transfer for lifelong robot learning. *Advances in Neural Information Processing Systems*, 36:44776–44791, 2023.

Jiacheng Liu, Chang Zou, Yuanhuiyi Lyu, Junjie Chen, and Linfeng Zhang. From reusing to forecasting: Accelerating diffusion models with taylorseers. In *2025 IEEE/CVF International Conference on Computer Vision (ICCV)*, pp. 15853–15863. IEEE, 2025.

Cheng Lu, Yuhao Zhou, Fan Bao, Jianfei Chen, Chongxuan Li, and Jun Zhu. Dpm-solver: A fast ode solver for diffusion probabilistic model sampling in around 10 steps. *Advances in neural information processing systems*, 35:5775–5787, 2022.

Xinyin Ma, Gongfan Fang, and Xinchao Wang. Deepcache: Accelerating diffusion models for free. In *2024 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 15762–15772. IEEE, 2024.

Yao Mu, Tianxing Chen, Zanxin Chen, Shijia Peng, Zhiqian Lan, Zeyu Gao, Zhixuan Liang, Qiaojun Yu, Yude Zou, Mingkun Xu, et al. Robotwin: Dual-arm robot benchmark with generative digital twins. In *2025 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 27649–27660. IEEE, 2025.

Aaditya Prasad, Kevin Lin, Jimmy Wu, Linqi Zhou, and Jeannette Bohg. Consistency policy: Accelerated visuomotor policies via consistency distillation. *arXiv preprint arXiv:2405.07503*, 2024.

Jiaming Tang, Yufei Sun, Yilong Zhao, Shang Yang, Yujun Lin, Zhuoyang Zhang, James Hou, Yao Lu, Zhijian Liu, and Song Han. Vlash: Real-time vlas via future-state-aware asynchronous inference. *arXiv preprint arXiv:2512.01031*, 2025.

Haoxuan Wang, Gengyu Zhang, Yan Yan, Yuzhang Shang, Ramana Kompella, and Gaowen Liu. Real-time robot execution with masked action chunking. In *International Conference on Learning Representations*, volume 2026, pp. 120161–120179, 2026.

Zhendong Wang, Zhaoshuo Li, Ajay Mandlekar, Zhenjia Xu, Jiaojiao Fan, Yashraj Narang, Linxi Fan, Yuke Zhu, Yogesh Balaji, Mingyuan Zhou, et al. One-step diffusion policy: Fast visuomotor policies via diffusion distillation. *arXiv preprint arXiv:2410.21257*, 2024.

Yujie Wei, Jiahan Fan, Jiyu Guo, Ruichen Zhen, Rui Shao, Xiu Su, Zeke Xie, Hongxun Yao, and Shuo Yang. Learning to accelerate vision-language-action models through adaptive visual token caching. *arXiv preprint arXiv:2602.00686*, 2026.

Ted Xiao, Eric Jang, Dmitry Kalashnikov, Sergey Levine, Julian Ibarz, Karol Hausman, and Alexander Herzog. Thinking while moving: Deep reinforcement learning with concurrent control. *arXiv preprint arXiv:2004.06089*, 2020.

Haozhe Xie, Beichen Wen, Jiarui Zheng, Zhaoxi Chen, Fangzhou Hong, Haiwen Diao, and Ziwei Liu. Dynamicvla: A vision-language-action model for dynamic object manipulation. *arXiv preprint arXiv:2601.22153*, 2026.

Siyu Xu, Yunke Wang, Chenghao Xia, Dihao Zhu, Tao Huang, and Chang Xu. VLA-Cache: Efficient vision-language-action manipulation via adaptive token caching. In *Advances in Neural Information Processing Systems*, volume 38, pp. 164448–164473, 2025.

Tianyuan Yuan, Zibin Dong, Yicheng Liu, and Hang Zhao. Fast-wam: Do world action models need test-time future imagination? *arXiv preprint arXiv:2603.16666*, 2026.

Weisen Zhao, Lam Nguyen, Zhicong Lu, and Yuzhang Shang. C^{3}ache: Accelerating world action models with cross inference chunk cache. *arXiv preprint arXiv:2606.08962*, 2026. URL `https://arxiv.org/abs/2606.08962`.

Brian Zhu, Momen Khalil, E Harrison, Emanuele Poggi, Philipp Schmitt, Bernd Kast, Philine Meister, Pranav Atreya, Qiyang Li, Finn Ferchau, et al. Learning to act while waiting: Rl finetuning of generalist robot policies under inference latency. *arXiv preprint arXiv:2608.23831*, 2026.

A Benchmark Construction and Task Catalogs

A.1 Task Catalogs

The main suite retains original instructions, object meshes, and success predicates. Dynamic-LIBERO uses ten `libero_object` skills, each with linear pick/place/both and curved pick/place variants (50 tasks). Dynamic-RoboTwin uses 15 skills with linear pick/place/both variants (45 tasks). Only the grasp target, placement target, or both move in the corresponding variant. Extensions that change instructions or motion families are excluded from these counts.

All LIBERO instructions follow "pick up the object and place it in the basket," with placement body `basket_1`. Table 5 specifies objects and curved templates; Table 6 identifies RoboTwin skills and target attributes. The t03 curved pick uses `clothoid_right` with length 0.26 m and curvature parameter 9.

A.2 Motion and Outcome Rules

Main runs use `main_track` with appearance retained (`keep`), `ghost_hold` for Static, and `ghost_drive` for Moving; auxiliary tracks and occlusion perturbations are disabled. Layout speed is fixed at 0.0006 m/tick, giving stationary and dynamic variants the same cleared corridor (0.40 m forward and 0.06 m backward). At 20 Hz, $v_1 = 1.2$ and $v_2 = 2.0$ cm/s correspond to 0.0006 and 0.001 m/tick. Motion follows the world clock rather than replanning calls. LIBERO linear headings

Table 5: Dynamic-LIBERO skills and curved-path templates.

ID	Grasp body	Heading	Curved pick	Curved place
t00	alphabet_soup_1	0°	s_wave	arc_right
t01	cream_cheese_1	36°	double_s	arc_left
t02	salad_dressing_1	72°	shallow_s	wide_ellipse
t03	bbq_sauce_1	108°	clothoid_right	clothoid_left
t04	ketchup_1	144°	liss_1_2	ellipse_left
t05	tomato_sauce_1	180°	soft_spiral	comma
t06	butter_1	216°	diag_s	bowl
t07	milk_1	252°	long_arc	side_s
t08	chocolate_pudding_1	288°	chirp_s	bean
t09	orange_juice_1	324°	tight_s	soft_spiral_ccw

Table 6: Dynamic-RoboTwin skills, body attributes, and nominal headings.

Skill	Grasp attribute	Placement attribute	Heading
place_empty_cup	cup	coaster	0°
place_phone_stand	phone	stand	24°
place_container_plate	container	plate	48°
place_object_basket	object	basket	72°
place_can_basket	can	basket	96°
place_object_stand	object	displaystand	120°
place_object_scale	object	scale	144°
place_bread_skillet	bread	skillet	168°
move_pillbottle_pad	pillbottle	pad	192°
move_stapler_pad	stapler	pad	216°
place_a2b_left	object	target_object	240°
place_a2b_right	object	target_object	264°
move_can_pot	can	pot	288°
place_mouse_pad	mouse	target	312°
place_fan	fan	pad	336°

are $(36j + 12b) \bmod 360$ degrees for skill j and variant $b = 0, 1, 2$; RoboTwin uses $24j$ degrees. Paths are oriented toward the table center. LIBERO curved paths span approximately 0.26–0.34 m with lateral amplitudes 0.03–0.13 m.

Targets follow kinematic constraints; visible vehicles provide context, not driving contact forces. Vehicle collisions are disabled (`contype=0`, `conaffinity=0` in LIBERO; `rails_collide=false` in RoboTwin). Constraints are released when the end effector enters a 0.06 m (LIBERO) or 0.12 m (RoboTwin) neighborhood, identically for stationary and dynamic variants. Release is not grasp or success evidence; subsequent contact can cause paired trajectories to diverge. Success uses the original BDDL predicate or `check_success()`. LIBERO targets escaping before release ($x, y \notin [-0.45, 0.45]$ m or $z \notin [-0.02, 0.40]$ m) fail early. Auxiliary grasp retention requires a 2 cm lift and proximity to the end effector for ten ticks; missing tracker outcomes are excluded only from retention rates.

B Evaluation and Timing Protocol

B.1 Policies and Execution Settings

The evaluation configuration uses eight A800-SXM4-80GB GPUs, one per worker; each policy invocation uses one GPU. FastWAM uses bfloat16, Python 3.10.20, and PyTorch 2.7.1+cu128; the OpenPI environment uses Python 3.11.15 and PyTorch 2.7.1+cu126 with selected parameters in bfloat16. RoboTwin $\pi_{0.5}$ runs as a localhost HTTP service; other policy calls run in the simulator process. Checkpoints and inputs are listed in Table 7.

Table 7: Checkpoints and observation preprocessing. Dimensions follow the evaluation configuration.

Policy / benchmark	Checkpoint and adapter	Camera inputs
FastWAM / LIBERO	`libero_uncond_2cam224.pt`	`agentview` and `robot0_eye_in_hand`; rendered at 256 and horizontally combined into a 224×448 model input.
FastWAM / RoboTwin	`robotwin_uncond_3cam_384.pt`	Head camera 320×256, left/right wrists 160×128; combined input 384×320.
$\pi_{0.5}$ / LIBERO	`pi05_libero_pytorch`; normalization statistics `pi05_libero`	The same two LIBERO cameras, each resized with padding to 224×224.
$\pi_{0.5}$ / RoboTwin	`pi05_robotwin2`; configuration `pi05_aloha`, `adapt_to_pi=True`	`cam_high`, `cam_left_wrist`, `cam_right_wrist`; environment image resolutions retained at the interface.

Table 8 distinguishes predicted length H, executor input length Q, replanning horizon r, and integration budget K. Replanning starts when the queue is empty or r actions have executed. FastWAM returns 32 actions; RoboTwin $\pi_{0.5}$ predicts 32 but returns only eight. PACE retains the nominal integration grid.

Table 8: Action generation and execution settings at 20 policy ticks per second.

Policy / benchmark	H	Q	r	Tail $Q - r$	K
FastWAM / Dynamic-LIBERO	32	32	10	22	10
FastWAM / Dynamic-RoboTwin	32	32	8	24	4
$\pi_{0.5}$ / Dynamic-LIBERO	10	10	10	0	10
$\pi_{0.5}$ / Dynamic-RoboTwin	32	8	8	0	10

Methods share task/seed/trajectory pairs, using official LIBERO initial states and RoboTwin initialization seeds. LIBERO uses 30 settling steps with $[0, 0, 0, 0, 0, 0, -1]$, followed by at most 400 counted steps; RoboTwin uses the native task limit, capped at 500 policy steps in the locked evaluation.

B.2 Delay Replay and Measurement

The asynchronous backend uses measured-delay replay: blocking prediction is timed, then $20(L_i + w)$ policy ticks are replayed before installing the new chunk, which remains conditioned on the pre-replay observation. Inference and simulation do not execute concurrently. Replay consumes old queued actions through `env.step`; with an empty queue, the robot holds its last controlled pose while the trajectory driver advances targets and calls `sim.forward()`, without ordinary robot dynamics steps. Unused tail actions are discarded at replacement. FastWAM retains 22/24 tail actions at the horizon trigger; both $\pi_{0.5}$ settings have $Q = r$.

L_{model} times the complete policy call between GPU synchronizations, including preprocessing, transfer, encoding, generation, and decoding; RoboTwin $\pi_{0.5}$ also includes the HTTP round trip. Replay uses $L_i^{\text{act}} = L_{\text{model},i} + w$, with no separately measured scheduling delay. Eight startup probes precede evaluation; their first three calls are discarded for the probe median. Episode latency includes first-call cache misses but excludes startup probes and added delivery wait. The probe median is distinct from per-replanning-call latency.

Success is averaged within each task and then equally across tasks; mean latency weights replanning calls. Bootstrap intervals resample seeds within tasks, keeping episode calls together for latency and aligned seeds together for paired contrasts. Per-setting latency means are exported separately.

B.3 Speed and Delivery-Delay Analysis

The speed sweep holds PACE and executor settings fixed at commanded speeds 0, 0.6, 1.2, 1.6, 2.0, and 3.0 cm/s. Zero speed uses the stationary modified scene. The joint success–latency curves measure robustness across speeds, not a causal latency effect.

The delivery intervention adds $w = 100$ ms after each native policy call without changing weights or cache settings. Initializations and prescribed motion are paired, but later observations and release events may diverge. Table 9 reports the closed-loop success difference and its speed interaction. All four LIBERO delay intervals lie below zero; RoboTwin's available interval spans zero. Three corrupted RoboTwin variants are excluded only from this paired comparison (42 tasks), not the main suite. Other RoboTwin delay contrasts are unavailable.

Table 9: Paired effect of adding 100 ms to action delivery with PACE fixed. $\Delta = \mathrm{SR}(+100\,\mathrm{ms}) - \mathrm{SR}(0\,\mathrm{ms})$; the interaction is $\Delta(v_2) - \Delta(v_1)$. RoboTwin excludes three corrupted task variants (42 paired tasks); its other contrasts are unavailable.

Benchmark	Policy	Contrast	Δ (pp)	95% CI (pp)
Dynamic-LIBERO	FastWAM	v_1	-16.2	$[-20.0, -12.6]$
	FastWAM	v_2	-10.0	$[-13.0, -7.2]$
	FastWAM	$v_2 - v_1$	$+6.2$	$[+1.8, +10.2]$
	$\pi_{0.5}$	v_1	-3.8	$[-7.0, -0.6]$
	$\pi_{0.5}$	v_2	-8.6	$[-12.6, -4.8]$
	$\pi_{0.5}$	$v_2 - v_1$	-4.8	$[-9.8, +0.0]$
Dynamic-RoboTwin	FastWAM	v_1	-0.5	$[-3.6, +2.9]$

C PACE and Baseline Implementations

C.1 PACE Instantiations

FastWAM reuses VAE latents, instruction representations, video tokens, action residuals, and within-chunk velocities; $\pi_{0.5}$ reuses SigLIP features, action residuals, and velocities. Both instantiations retain frozen weights, the nominal integration grid, and the original action interface. The gates below determine which computations are bypassed; all integration updates remain active.

C.2 Visual and Instruction Caches

The visual branch in Equation 4 compares the current multi-camera input with the reference associated with the cached representation. A hit reuses the representation; a miss evaluates the encoder and refreshes both the representation and its reference image. FastWAM's VAE cache uses a relative RGB-L_1 threshold of 0.015, one warm-up call, and at most eight consecutive reuses. The SigLIP cache in $\pi_{0.5}$ uses a relative RGB-L_1 threshold of 0.02.

FastWAM additionally caches the T5 representation for unchanged instructions, with up to 64 entries. Its video-token cache reuses the whole video K/V block below a global L_1 change of 0.03; otherwise, it permits sparse refresh when no more than 40% of tokens have changed. Larger changes trigger a full refresh. These thresholds apply to different representations and are not directly comparable across modules.

C.3 Action Residual and Velocity Caches

Residual reuse across replanning calls. A compatible reference entry r stores the hidden residual $R_r = h_r^{\mathrm{out}} - h_r^0$ and the conditioning and latent signals used by the gate. Reuse is enabled when

$$g_{i,k}^{\mathrm{res}} = b_{i,k}^{\mathrm{res}} \mathbb{1}\big[d_c(c_i^{\mathrm{gate}}, c_r^{\mathrm{gate}}) < 0.05\big] \mathbb{1}[d_z(z_{i,k}, z_r) < 0.15]\,. \tag{8}$$

Here c^{gate} denotes the conditioning signal inspected by the module, while d_c and d_z measure conditioning and latent changes. The admissibility flag $b_{i,k}^{\mathrm{res}}$ requires a valid reference, a one-chunk cold start, and at most four consecutive reuses. On a hit, the current embedding receives the cached

residual and the current output head predicts the velocity, as in Equation 6. On a miss, a full hidden-network evaluation refreshes the residual reference. Thus, the cache stores a hidden transformation rather than a previously generated action chunk.

Velocity reuse within an action chunk. Let $\kappa < k$ be the most recent step at which a velocity was evaluated. The within-chunk gate is

$$g_{i,k}^{\text{step}} = b_{i,k}^{\text{step}} \mathbb{1}[d_{\text{step}}(z_{i,k}, z_{i,\kappa}) < 0.20], \tag{9}$$

where d_{step} measures relative-L_1 latent change. The first two integration steps evaluate velocities, and the admissibility flag permits at most two consecutive holds. A hit reuses the complete velocity vector, including its magnitude, in the current Euler update. A miss evaluates a new velocity through the residual path or the full network and refreshes the within-chunk reference. The nominal integration coordinates and step sizes remain unchanged.

C.4 Ordered Runtime Execution

Visual reuse reduces observation encoding, residual reuse reduces hidden transformations across replanning calls, and velocity reuse reduces repeated evaluations within a chunk. The velocity gate is checked before the residual gate because a held velocity bypasses both the hidden layers and the output head. Such a hit receives no additional residual correction. When velocity reuse fails, PACE evaluates the current output head using either a cached residual or a freshly computed hidden transformation. Table 10 summarizes the execution order.

Table 10: PACE inference with backbone-specific enabled cache paths.

Step	Stage	Runtime operation
1	Observation caches	Check visual and instruction caches and, for FastWAM, video-token caches. Reuse admissible representations and refresh the required quantities on misses. Condition on the current robot state.
2	Latent setup	Sample the initial action latent and iterate over the native integration coordinates.
3	Velocity reuse	If the within-chunk velocity gate passes, supply the cached velocity directly to the integration update.
4	Residual or full path	Otherwise, check residual reuse. On a hit, add the cached residual to the current embedding and evaluate the current output head; on a miss, evaluate the full action network and refresh the residual. Refresh the velocity cache.
5	Update and decode	Apply the Euler update at every coordinate and decode the terminal latent through the original action interface.

C.5 Baseline Adaptations

Methods share checkpoints, observations, action interfaces, and executors. Baseline uses full computation with the original sampling budget. TaylorSeer (Liu et al., 2025) extrapolates action velocities while retaining the integration grid; its FastWAM adaptation also caches instruction representations. C^{3}ache (Zhao et al., 2026) uses scheduled cross-chunk residual reuse on FastWAM with periodic full refreshes. AdaVLA (Han et al., 2026) uses curvature-based adaptive sampling on $\pi_{0.5}$; the evaluated adaptation covers its sampler without channel pruning.

D Real-World Experimental Details

D.1 Hardware, Control, and Trial Protocol

We use an AgileX Piper 6-DoF arm with an AgxGripper (0.1 m maximum opening), third-person and wrist cameras (1280×720, 30 fps), and an RTX 4090 D shared by policy server and controller. OpenPI WebSocket communication uses client images padded to 224×224. $\pi_{0.5}$ is LoRA-fine-tuned on 30 real and 60 simulated demonstrations; weights are merged at step 10,000. Both runtimes use identical weights and ten denoising steps. Baseline uses full Euler sampling; PACE adds residual/step caching and batched image encoding.

The policy predicts 25-step chunks at 15 Hz, interpolated to 30 Hz. Execution is synchronous with $H = 10$ or 5 steps between inference calls. Joint/gripper targets are filtered with $\tau = 0.10/0.04$ s; arm speed is 10% of the firmware limit. T1 stacks a small box on a base box; T2 places the cup in the tray; T3 takes the box from the plate to the near table edge. Trials begin at joint zero with an open gripper and marked object positions. Methods are interleaved (ABAB), with ten trials per condition. Completion time spans motion onset to gripper release for successful trials; pilot and operator-interrupted trials are excluded before analysis.