REALTIMEWAM: HOW FAST CAN I RUN MY WORLD ACTION MODEL?

Anonymous authors
Paper under double-blind review

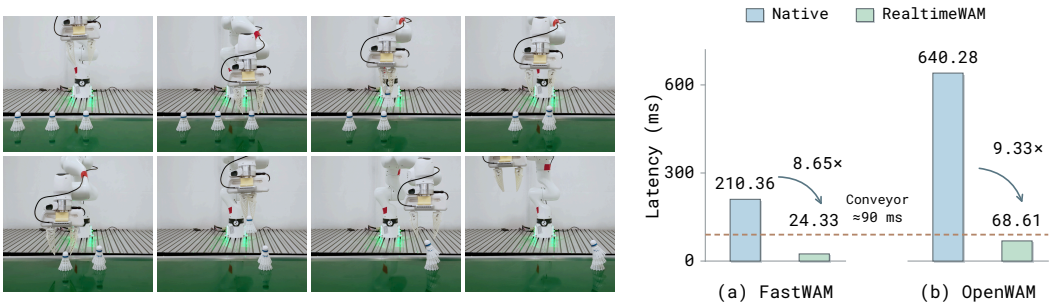


Figure 1: **Responsive robot control with RealtimeWAM.** Left: stacking three shuttlecocks on the conveyor requires rapid responses to scene changes. Right: mean measured inference latency on this task from Table 4, on RTX 4090. The dashed line marks the empirically observed conveyor-task threshold of approximately 90 ms, above which task success was substantially lower (Appendix A).

ABSTRACT

World Action Models (WAMs) combine visual dynamics modeling with action generation, but their high inference latency limits responsive robot control. Recent efforts accelerate inference by removing explicit future-video generation at test time, as in FastWAM, an approach that requires a specially tailored architectural design. More general caching strategies exploit feature redundancy, but redundancy alone does not capture the changing computational demands of closed-loop control. To address these challenges, we present RealtimeWAM, a general, training-free framework that coordinates parallel execution with adaptive computation for low-latency inference across diverse WAM architectures. We exploit layerwise dependencies to overlap observation processing with prediction. However, concurrent branches still compete for GPU resources, limiting the benefit of parallel execution. We therefore adapt computation throughout the pipeline through selective reuse, caching observation features in visually stable regions and reusing Transformer residuals while reserving additional refinement for small predicted adjustments. We evaluate RealtimeWAM on FastWAM and OpenWAM across RoboTwin, LIBERO, and LIBERO-Plus. On an RTX 4090, measured mean inference latencies are 24.09 and 63.09 ms, corresponding to average speedups of 8.90× and 10.67×. Average success rates are 82.75% and 87.41%, respectively, within 0.02 and 0.53 percentage points of native inference. Across five real-world tasks, RealtimeWAM improves average success rates over native inference by 17.2 and 37.2 percentage points on FastWAM and OpenWAM, respectively. Project page: https://anonymous.4open.science/w/realtimewam/.

1 INTRODUCTION

World Action Models (WAMs) integrate visual dynamics modeling with action generation, providing predictive representations for robotic control (Cen et al., 2025; Bi et al., 2025; Kim et al., 2026; Yuan et al., 2026; Wang et al., 2026). However, their substantial inference cost limits how quickly a robot can respond to changes in its environment. This limitation is particularly problematic in dynamic manipulation, where the scene continues to evolve while the model generates an action (Black et al.,

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.

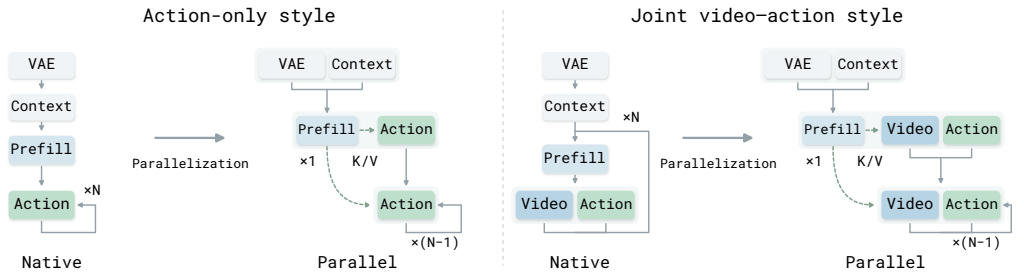


Figure 2: **Parallel execution across WAM architectures.** Left: action prediction alone. Right: joint video and action prediction. Each pair contrasts native and parallel execution. Layerwise release of observation K/V allows prediction to proceed alongside Prefill; shaded regions group work available for concurrent execution.

2025b). For example, a moving object on a conveyor may shift away from its observed position before the predicted action is executed. Reducing latency from observation to action is therefore essential for using WAMs to control robots in real time.

Recent work has reduced inference cost through architectural simplification and computational reuse. For example, FastWAM removes future-video generation at inference while retaining joint training with video (Yuan et al., 2026), and diffusion caching methods such as TeaCache avoid repeated Transformer computation by reusing residual features (Liu et al., 2025a). These advances leave two further questions for WAM execution in real time. First, reducing computation alone does not resolve how observation processing and prediction should be scheduled under their data dependencies and shared hardware constraints. Second, caching decisions based on denoising similarity are not explicitly tied to the responsiveness and precision required by the current action. Fixed evaluation schedules likewise apply the same computational budget throughout a rollout, despite changing execution requirements. Further acceleration therefore calls for coordinating parallel execution with computation that adapts to the current observation and predicted motion.

To address these challenges, we present RealtimeWAM, a framework that coordinates parallel execution with adaptive computation for existing WAM architectures without additional training. We first exploit the layerwise dependencies between processing the current observation (Prefill) and prediction. As each layer produces the required observation features, the corresponding prediction computation can begin while Prefill continues (Figure 2). This allows Prefill to overlap with action prediction and, when present, future-video generation. However, concurrent branches compete for shared GPU compute resources, limiting the speedup achievable through overlap alone.

To ease this contention, we exploit temporal redundancy across observations to reduce the Prefill workload. Changes in the current observation guide which tokens receive fresh computation, while cached features supply the remaining positions. However, reducing the number of refreshed tokens does not always reduce latency, as different token counts can fall within the same latency plateau (Table 3, right). We use this available capacity to refresh additional tokens with larger feature changes in the first layer, limiting reuse error at little extra latency. This reuse policy accounts for hardware latency and reduces the workload competing with prediction while keeping observation features current.

Token reuse reduces the cost of the first stage, but subsequent denoising still requires repeated network evaluations. To further accelerate inference, we adapt the refresh schedule of cached Transformer residuals according to predicted motion. A coarse action prediction provides a displacement signal for deciding whether further refinement is needed. Motion amplitude and the effect of additional computation vary within an episode (Figure 3), motivating a budget adapted to motion. We allocate fewer evaluations to large predicted movements to favor rapid replanning, while retaining additional evaluations for small adjustments. We evaluate this allocation through closed-loop ablations in Section 4.4 and paired prediction comparisons in Appendix E. Refinement continues from the same initial denoising trajectory, preserving the computation already performed.

We evaluate FastWAM and OpenWAM on RoboTwin, LIBERO, and LIBERO-Plus. Combining execution optimizations with adaptive computation gives measured average speedups of 8.90× and 10.67×, respectively, relative to native inference. Average success rates decrease by 0.02 and 0.53

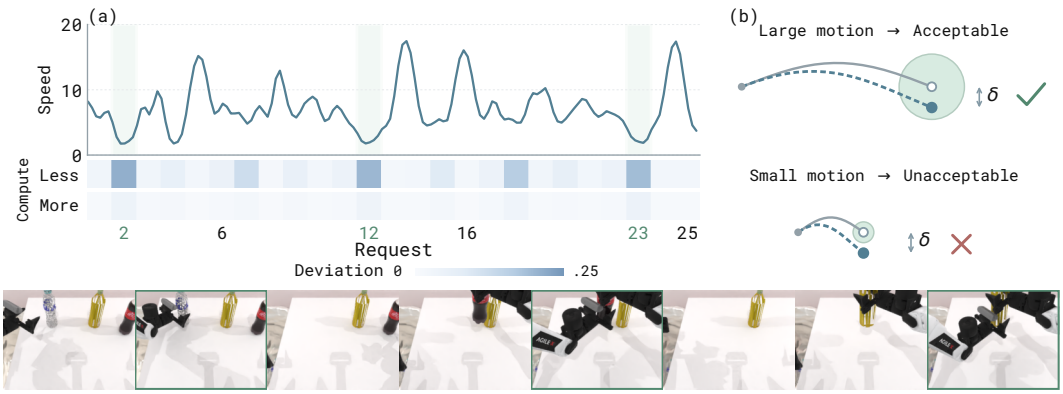


Figure 3: **Changing motion and precision requirements. (a)** Smoothed predicted speed (mm/control step) and relative deviation from a reference using more computation. Darker cells indicate larger deviations; green marks identify selected observations. **(b)** Illustrative cases: the same offset δ fits the large movement's wide tolerance but exceeds the small adjustment's narrow tolerance. Measurement details are in Appendix E.

percentage points (Table 1). Controlled experiments examine which scheduling and computation allocation choices account for these gains.

Our main contributions are:

- We design a parallel schedule guided by layer dependencies that overlaps observation processing with action prediction and, where applicable, future-video generation.
- We develop a token reuse policy that reduces the Prefill workload and exploits hardware latency plateaus to refresh additional informative tokens.
- We adapt Transformer residual caching to predicted motion, using the displacement of a coarse action prediction to determine whether further refinement is needed.
- We evaluate two WAM architectures on three simulation benchmarks and five real-world tasks, with controlled experiments examining execution, token allocation, and predictive refinement.

2 Related Work

World action models and real-time inference. Diffusion Policy and π_0 generate actions with diffusion and flow matching, respectively (Chi et al., 2023; Black et al., 2025a). WAMs couple visual prediction with action generation (Shen et al., 2025; Bi et al., 2025; Kim et al., 2026; Li et al., 2026a). FastWAM omits future-video generation at inference; OpenWAM jointly predicts video and actions (Yuan et al., 2026; Wang et al., 2026). Flash-WAM distills joint prediction to fewer steps (Akbari et al., 2026). RTC, SmolVLA, and LingBot-VA overlap policy inference with action execution (Black et al., 2025b; Shukor et al., 2025; Li et al., 2026a). Realtime-VLA and VLA-Perf examine inference optimization (Ma et al., 2025; Jiang et al., 2026). We overlap observation processing and prediction within one request.

Temporal reuse and diffusion caching. Diffusion caching reuses or extrapolates intermediate features across timesteps (Ma et al., 2024; Liu et al., 2025a; Zou et al., 2025; Liu et al., 2025b). FasterCache and PAB reuse diffusion features and attention outputs (Lyu et al., 2025; Zhao et al., 2025); TokenCache learns token importance for selective caching (Lou et al., 2024). PostDiff compares fewer denoising steps with mixed-resolution generation and module caching (Du et al., 2025). VLA-Cache, BAC, and EfficientVLA apply reuse at visual-token, block, and diffusion-head levels (Xu et al., 2025; Ji et al., 2026a; Yang et al., 2025). Our token reuse reduces Prefill contention and uses latency plateaus to refresh additional tokens.

Adaptive computation. DynamicViT learns token sparsification; FastV prunes visual tokens using attention scores (Rao et al., 2021; Chen et al., 2024). PRANCE jointly adapts tokens and channels (Li

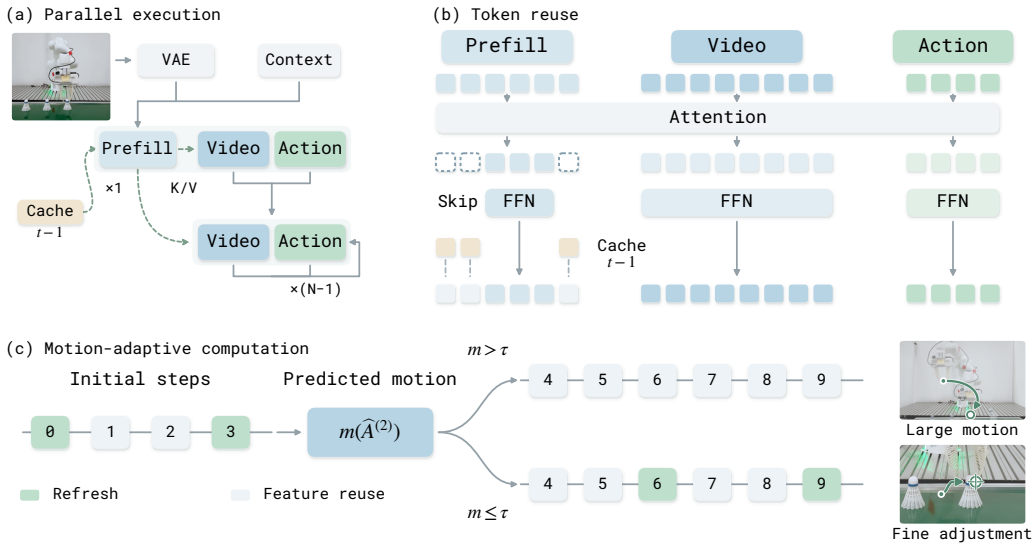


Figure 4: **Coordinating token reuse and motion-adaptive refinement. (a)** Observation processing proceeds alongside Video and Action prediction. **(b)** Selected Prefill tokens receive fresh FFN outputs; the remaining positions reuse cached outputs. Future-video and action prediction execute their full branches at each selected Transformer evaluation. **(c)** Transformer refreshes at Euler updates 0 and 3 provide the motion estimate that selects additional refreshes at 6 and 9. Both schedules perform ten updates, with current input features and output heads evaluated at every step.

et al., 2026d). AdaCache uses video motion to guide caching (Kahatapitiya et al., 2025). SAG learns observation-conditioned pruning; Test-time Sparsity combines parallel inference with learned cached-feature selection (Ji et al., 2026c;b). SP-VLA couples scheduling with token pruning; ElegantVLA allocates computation across perception, language, and action (Li et al., 2026c;b). TS-DP uses reinforcement learning to adapt speculative diffusion decoding (Li et al., 2025). RealtimeWAM allocates Prefill and refinement budgets from observation changes and predicted motion, without training a selector.

3 Method

RealtimeWAM coordinates execution order and fresh computation within each WAM request (Figure 4). We first overlap independent work using the model's dependencies. Hardware-aware token reuse then reduces the observation workload competing with prediction, while motion-adaptive refinement allocates subsequent network evaluations.

3.1 Anatomy of WAM inference

At request t, a WAM maps images, robot state, and an instruction to an H-step action chunk. Current-observation processing, termed *Prefill*, provides features for action prediction and, when present, joint video–action prediction. Under first-frame-causal attention and fixed clean-frame conditioning, observation tokens are independent of the evolving prediction. Their keys and values (K/V) provide the observation context required by the predictive branches at each layer.

Let R_t be the observation positions refreshed after the first layer, and N_t the number of full predictive Transformer evaluations. Request latency can be decomposed as

$$T_t \approx T_{\text{first}}(R_t) + \sum_{j=2}^{N_t} T_{\text{full},j} + T_{\text{light}} + T_{\text{finish}}. \tag{1}$$

The first stage includes input preparation, Prefill, and the initial prediction. Thus R_t controls the workload within the first stage, while N_t controls the amount of subsequent refinement. T_{light} covers the remaining input embeddings, output-head evaluations, and sampler updates; T_{finish} covers final

action postprocessing and return to the CPU. The measured request includes online selection, data movement, and synchronization.

3.2 Dependency-aware parallel execution

We overlap independent input preparation: VAE encoding produces visual latents while context preparation produces instruction features and conditioning K/V, reusing instruction features when unchanged.

Prediction consumes observation K/V layer by layer. We release them immediately after projection, allowing prediction to proceed alongside the remaining Prefill attention and FFN operations (Figure 2). This overlaps observation processing with the first predictive evaluation, including the future-video branch when present.

Within each layer, local projections run concurrently; joint attention waits for the required K/V, after which conditioning attention and FFNs resume. Fused operators, preallocated buffers, and CUDA Graph replay reduce transfers, repeated assembly, and launch overhead.

The overlapping branches share GPU resources, so we reduce their concurrent workload by refreshing selected observation positions and reusing cached outputs at the others.

3.3 Hardware-aware token reuse

The refresh budget must reflect observation changes and hardware cost. GPU matrix operations exhibit tile- and wave-quantization effects (NVIDIA, 2023). Profiling our concurrent first stage reveals ranges of token counts with nearly unchanged latency (Table 3, right). We treat each plateau's upper end as an execution capacity. Figure 5 shows how RGB changes establish required updates and feature changes guide the additional updates that fill this capacity.

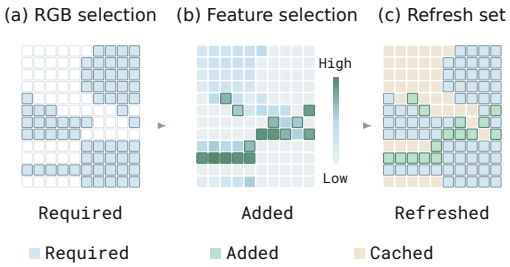


Figure 5: **Prefill token allocation.** **(a)** RGB change identifies required tokens. **(b)** Feature change ranks the remaining tokens to fill spare capacity. **(c)** Required and added tokens are refreshed; the rest stay cached. Each cell is one observation token.

Selecting an execution capacity. For M observation tokens, let D_t contain positions whose mean absolute RGB change exceeds δ. We require these positions to refresh and reserve a fraction α of the remaining positions to capture changes not identified by RGB alone. Offline profiling of the concurrent first stage gives execution capacities $\mathcal{B}$, including full refresh M. We choose the smallest capacity covering the required and reserved updates:

$$r_t = |D_t| + \lceil \alpha(M - |D_t|) \rceil, \qquad b_t = \min\{b \in \mathcal{B} : b \geq r_t\}. \tag{2}$$

Each capacity uses a prepared execution shape for concurrent Prefill and prediction.

Filling the selected capacity. The full first layer provides current features for every position. To spend the remaining capacity on changing features, we rank positions outside D_t by normalized change from the preceding observation:

$$s_{t,i} = \frac{2\|f^0_{t,i} - f^0_{t-1,i}\|_1}{\max(\|f^0_{t,i}\|_1 + \|f^0_{t-1,i}\|_1, \epsilon)}, \qquad R_t = D_t \cup \mathrm{Top}_{b_t - |D_t|}\{s_{t,i} : i \notin D_t\}. \tag{3}$$

Here, $f^0_{t,i}$ is the first-layer feature and Top_k returns the k highest-scoring indices. The set R_t retains every RGB-required position and supplements it with feature-ranked positions. The same set serves subsequent layers; the feature reference advances at every observation.

Executing fresh and cached tokens. For each reused layer $\ell > 0$, selected positions update their FFN cache entries while the others retain their last computed values. Writing Φ_ℓ for the FFN

computation and $z_{t,i}^{\ell}$ for its current input, the update is

$$C_{t,i}^{\ell} = \begin{cases} \Phi_{\ell}(z_{t,i}^{\ell}), & i \in R_t, \\ C_{t-1,i}^{\ell}, & i \notin R_t. \end{cases} \tag{4}$$

FFN reuse applies to Prefill; future-video and action tokens are fully computed at each selected evaluation. Compatible operations batch refreshed observation and prediction rows. Attention and current residual gates remain fully computed (Appendix D). Full refresh initializes the FFN cache; episode or conditioning changes invalidate incompatible history.

3.4 Motion-adaptive refinement

Repeated Transformer evaluations remain on the critical path after the first stage. We cache the input-to-output residual of the Transformer. Each denoising step embeds the current latent, adds the cached residual, and evaluates the output head with the current timestep. Full evaluations refresh this residual. Motivated by the varying precision demands in Figure 3, we use predicted motion amplitude to allocate refinement: large motions favor earlier replanning, while small adjustments receive additional computation.

Both schedules share two initial Transformer refreshes. After the second, we decode a coarse action prediction. For current end-effector position p_t and predicted physical positions $\widehat{p}_{t,k}^{(2)}$, we use the maximum displacement to choose the evaluation budget:

$$m_t = \max_{1 \le k \le H} \|\widehat{p}_{t,k}^{(2)} - p_t\|_2, \qquad N_t = 2 + 2\,\mathbf{1}[m_t \le \tau]. \tag{5}$$

The threshold τ is fixed during deployment. The 2F schedule refreshes at Euler updates 0 and 3; the 4F schedule adds refreshes at 6 and 9. Both perform ten Euler updates using the current output-head prediction at each step. Refinement continues from the shared prefix with the same noise and conditioning (Figure 4c).

Together, R_t and N_t allocate fresh computation within the first stage and across later evaluations. Section 4.4 compares fixed and adaptive budgets; Appendix E gives the sampler settings and paired-output diagnostics.

4 Experiments

We evaluate RealtimeWAM on simulation and real-world manipulation tasks, and ablate execution optimizations, token reuse, and motion-adaptive refinement.

4.1 Experimental setup

Backbones, benchmarks, and baselines. We evaluate FastWAM and OpenWAM on RoboTwin 2.0 clean/randomized (Chen et al., 2025), LIBERO (Liu et al., 2023), and LIBERO-Plus (Fei et al., 2026). Each backbone's native inference is the reference; VLA-Cache (Xu et al., 2025), TeaCache (Liu et al., 2025a), ToCa (Zou et al., 2025), and BAC (Ji et al., 2026a) provide training-free caching baselines. We enable compilation and CUDA Graph replay for the caching baselines and RealtimeWAM. Baselines retain their original execution schedules; RealtimeWAM adds the proposed parallel execution. Table 1 reports success rates per setting and equally weighted averages over the four settings. LIBERO category results appear in Appendix B.4.

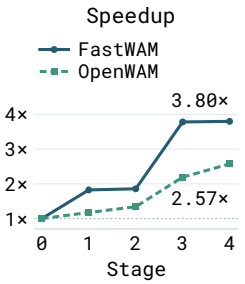
Metrics and timing. We report success rate and full-request latency, with speedup relative to native inference. Runtime measurements use BF16, batch size one, and a modified 48 GiB RTX 4090. Timing covers the prepared CPU image/state interface through denormalized CPU actions, including input processing, model and sampler execution, transfers, and synchronization. Compilation, graph capture, warmup, profiling, and robot execution are excluded. 2F/4F denote two/four full Transformer evaluations within ten Euler updates. The main comparison and execution ablations report mean latencies measured during complete closed-loop rollouts. Model-specific interfaces and checks appear in Appendix C.

Table 1: **Training-free acceleration across manipulation benchmarks.** Columns report success rates (%) and mean latency (ms); Average weights settings equally. Caching baselines use compilation and CUDA Graph replay. Speedup is original Native mean latency divided by method mean latency. Best and second-best results exclude Native; lower latency is better.

Backbone	Method	RoboTwin		LIBERO		Average		
		Clean	Random	LIBERO	Plus	SR (%)	Lat. (ms)	Speedup
FastWAM	Native	91.88	91.78	97.60	49.80	82.77	214.33	1.00×
	VLA-Cache	80.00	83.20	97.00	48.60	77.20	115.55	1.85×
	TeaCache	81.42	90.86	97.05	49.20	79.63	49.28	4.35×
	ToCa	91.35	91.24	94.55	49.15	81.57	102.66	2.09×
	BAC	91.67	89.53	97.50	51.55	82.56	67.14	3.19×
	Ours	**92.04**	**90.50**	**97.25**	**51.20**	**82.75**	**24.09**	**8.90×**
OpenWAM	Native	93.64	94.06	99.30	64.75	87.94	672.83	1.00×
	VLA-Cache	93.36	89.94	98.40	62.70	86.10	400.26	1.68×
	TeaCache	90.42	94.12	98.60	63.80	86.74	245.99	2.74×
	ToCa	93.42	93.98	98.10	63.70	87.30	440.42	1.53×
	BAC	93.52	91.96	98.90	66.70	87.77	305.08	2.21×
	Ours	**92.78**	**94.02**	**99.00**	**63.85**	**87.41**	**63.09**	**10.67×**

Table 2: **Cumulative execution optimizations on RTX 4090 (ms).** Values are mean latencies measured during complete closed-loop rollouts. First: latency from prepared CPU inputs through Prefill and the first predictive evaluation in a 2F run; Full 10F: request latency from a separate 10F run. Stages are cumulative. The plot shows first-forward speedup over Native. Darker/lighter fills mark the lowest/second-lowest latency among optimized stages.

Stage	Optimization	FastWAM		OpenWAM	
		First	Full 10F	First	Full 10F
0	Native	67.24	214.33	81.37	672.83
1	+ CUDA Graph	36.83	104.09	69.34	524.69
2	+ Compile	36.21	99.68	60.62	450.90
3	+ Parallel	17.77	33.52	37.12	223.27
4	+ Token reuse	17.69	29.47	31.62	217.24



4.2 Overall Comparison

Success–latency tradeoff. RealtimeWAM has the lowest reported average latency on both backbones (Table 1). FastWAM latency decreases from 214.33 to 24.09 ms (8.90×), while the success rate changes from 82.77% to 82.75%. OpenWAM latency decreases from 672.83 to 63.09 ms (10.67×), with the success rate changing from 87.94% to 87.41%. Relative to Native with compilation and CUDA Graph replay, speedups are 4.14× on FastWAM and 7.15× on OpenWAM (Table 9).

Comparison with caching baselines. With compilation and CUDA Graph replay, TeaCache is the fastest competing baseline at 49.28/245.99 ms on FastWAM/OpenWAM, with lower average success rates than ours on both. BAC retains a higher OpenWAM success rate (87.77% versus 87.41%) at 305.08 ms. Appendix C.3 compares original and compiled execution for each baseline.

4.3 Sources of Latency Reduction

Table 2 adds execution optimizations at fixed evaluation budgets: first-forward timings come from 2F runs, and complete requests from separate 10F runs. After graph replay and compilation, parallel execution reduces first-forward latency from 36.21 to 17.77 ms on FastWAM and from 60.62 to 37.12 ms on OpenWAM. Token reuse further reduces these to 17.69 and 31.62 ms; its incremental benefit is larger on OpenWAM in this configuration.

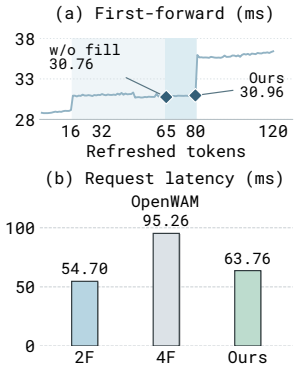
Request latency with 10F falls from 214.33 to 29.47 ms on FastWAM and 672.83 to 217.24 ms on OpenWAM, establishing execution savings before adaptive evaluation counts are introduced. Detailed stage and preparation measurements appear in Appendix C.

4.4 Ablation studies

Table 3 ablates the full OpenWAM method. Token ablations keep the motion gate unchanged; motion ablations keep the token policy unchanged. Request latencies are averaged over complete rollouts. Fixed 2F/4F use fixed evaluation counts with the normal token policy.

Table 3: **Component ablations and latency diagnostics.** Left: OpenWAM/RoboTwin success rate, measured first-forward latency (First), and rollout-mean request latency (Request); latencies are in ms. Fixed 2F/4F retain the normal token policy. ✓/×: enabled/disabled; n/a: inactive; Sel.: RGB/feature selection; Gate: motion criterion; Ada./Rnd.: adaptive/random 2F/4F. Right: (a) historical OpenWAM first-stage profile (line), with measured first-forward latency for w/o filling (65 tokens) and Ours (80 tokens) marked by diamonds; (b) rollout-mean request latencies for Fixed 2F, Fixed 4F, and Ours, on RTX 4090.

Configuration	Token policy			Motion		Results		
	Reuse	Sel.	Fill	NFE	Gate	SR (%)	First	Request
Ours (full)	✓	✓	✓	**Ada.**	✓	**93.4**	**30.96**	**63.76**
w/o token reuse	×	n/a	n/a	Ada.	✓	93.8	37.41	72.10
Random tokens	✓	×	✓	Ada.	✓	91.3	30.75	62.32
w/o capacity filling	✓	✓	×	Ada.	✓	92.1	30.76	61.78
Random gate	✓	✓	✓	Rnd.	×	91.1	31.02	64.77
Fixed 2F	✓	✓	✓	2F	×	89.6	31.11	54.70
Fixed 4F	✓	✓	✓	4F	×	93.7	30.87	95.26



Token reuse and selection. Disabling reuse gives 93.8% success at 72.10 ms, compared with 93.4% at 63.76 ms for the full method. Uniformly selecting b_t random positions instead of the RGB/feature-guided set preserves the refreshed token count but lowers the success rate to 91.3%, supporting informative token selection.

Capacity filling. Filling from the minimum refresh budget r_t to capacity b_t raises the success rate from 92.1% to 93.4%, with request latency increasing from 61.78 to 63.76 ms. First-forward latency rises from 30.76 to 30.96 ms when filling from 65 to 80 tokens (Table 3a).

Motion-adaptive refinement. The motion gate yields 93.4% success at 63.76 ms, versus 91.1% at 64.77 ms for random 2F/4F selection using the original gate's usage proportions. This supports allocating refinement based on motion. Fixed 2F reaches 89.6% success at 54.70 ms, while Fixed 4F reaches 93.7% at 95.26 ms. The adaptive policy attains a success rate close to Fixed 4F with a lower mean request latency. Appendix E gives paired prediction diagnostics.

Visualization of dynamic allocation. Figure 6 aligns observations, selected evaluation budgets, and token allocation over one complete rollout. Both evaluation budgets and all three token capacities occur.

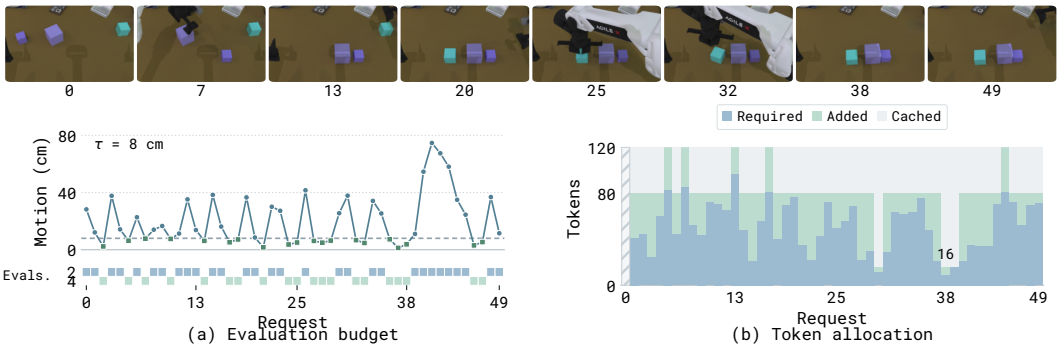


Figure 6: **Dynamic allocation over one rollout.** Shared observations accompany evaluation budgets (a) and token allocation (b) across all 50 requests of an OpenWAM block-sorting rollout. Hatching marks initialization.

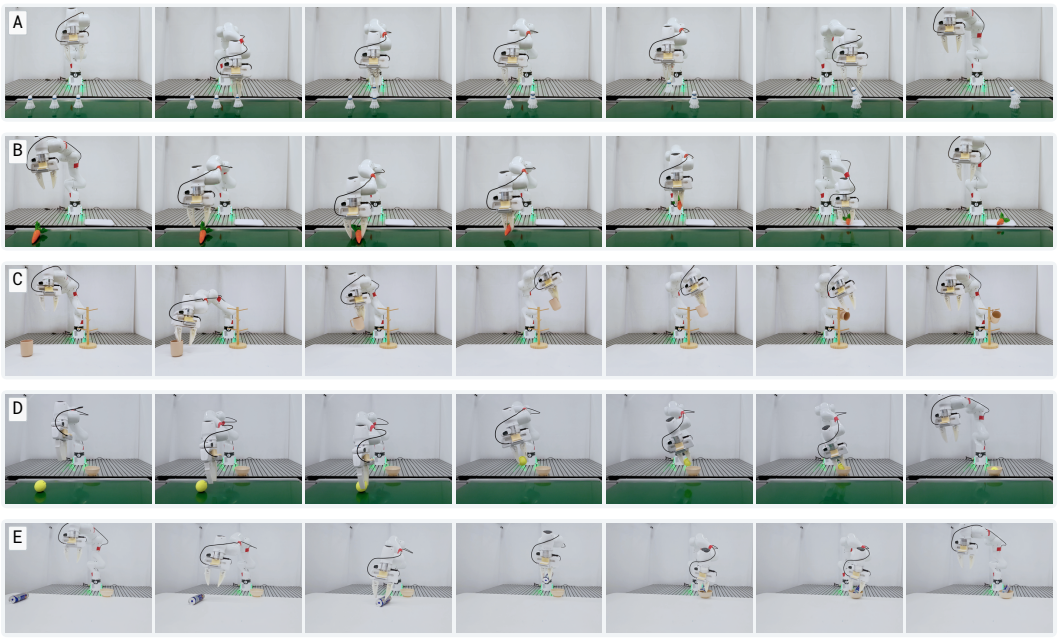


Figure 7: **Real-world task sequences.** Rows A–E show stacking three shuttlecocks, placing a carrot on a plate, hanging a mug, transferring a tennis ball, and intercepting a rolling can, respectively. Seven frames progress from left to right within each recording.

4.5 Real-world evaluation

Table 4 compares Native and RealtimeWAM on the five tasks in Figure 7 under the same synchronous inference and action execution policy. Average success rates rise from 35.6% to 52.8% on FastWAM and 33.6% to 70.8% on OpenWAM (+17.2 and +37.2 percentage points). Mean inference latency decreases from 210.71 to 23.72 ms and from 640.87 to 65.27 ms, respectively.

Table 4: **Real-world manipulation results.** Success rate (SR, %) and mean inference latency (Lat., ms). Average weights tasks equally.

Task	FastWAM				OpenWAM			
	Native		**Ours**		Native		**Ours**	
	SR ↑	Lat. ↓	**SR ↑**	**Lat. ↓**	SR ↑	Lat. ↓	**SR ↑**	**Lat. ↓**
Three shuttlecocks	24.0	210.36	**40.0**	**24.33**	4.0	640.28	**68.0**	**68.61**
Carrot to plate	46.0	209.74	**74.0**	**21.86**	66.0	638.56	**96.0**	**62.12**
Hang mug	72.0	211.08	**70.0**	**25.07**	82.0	642.15	**82.0**	**65.36**
Tennis ball to bowl	26.0	210.52	**34.0**	**23.24**	16.0	639.47	**76.0**	**64.85**
Rolling can to bowl	10.0	211.85	**46.0**	**24.11**	0.0	643.89	**32.0**	**65.43**
Average	35.6	210.71	**52.8**	**23.72**	33.6	640.87	**70.8**	**65.27**

5 Conclusion

RealtimeWAM combines parallel execution, hardware-aware token reuse, and motion-adaptive refinement in a training-free framework. On FastWAM and OpenWAM, measured average speedups reach 8.90× and 10.67×, with success rates within 0.02 and 0.53 percentage points of native inference. Across five real-world tasks, average success rates improve by 17.2 and 37.2 percentage points.

AI USE STATEMENT

We used generative AI tools to assist with manuscript writing and editing, including improvements to wording, grammar, clarity, and organization, as well as LaTeX formatting. The authors reviewed and revised the AI-assisted content to ensure that it accurately reflects the methods and results. The authors take full responsibility for the accuracy and integrity of the final manuscript.

REFERENCES

Arman Akbari, Ci Zhang, Arash Akbari, Lin Zhao, Yixiao Chen, Weiwei Chen, Xuan Zhang, Geng Yuan, and Yanzhi Wang. Flash-WAM: Modality-aware distillation for world action models. *arXiv preprint arXiv:2606.05254*, 2026. URL https://arxiv.org/abs/2606.05254.

Hongzhe Bi, Hengkai Tan, Shenghao Xie, Zeyuan Wang, Shuhe Huang, Haitian Liu, Ruowen Zhao, Yao Feng, Chendong Xiang, Yinze Rong, Hongyan Zhao, Hanyu Liu, Zhizhong Su, Lei Ma, Hang Su, and Jun Zhu. Motus: A unified latent action world model. *arXiv preprint arXiv:2512.13030*, 2025. URL https://arxiv.org/abs/2512.13030.

Kevin Black, Noah Brown, Danny Driess, Adnan Esmail, Michael Robert Equi, Chelsea Finn, Niccolo Fusai, Lachy Groom, Karol Hausman, Brian Ichter, Szymon Jakubczak, Tim Jones, Liyiming Ke, Sergey Levine, Adrian Li-Bell, Mohith Mothukuri, Suraj Nair, Karl Pertsch, Lucy Xiaoyang Shi, Laura Smith, James Tanner, Quan Vuong, Anna Walling, Haohuan Wang, and Ury Zhilinsky. π_0: A vision-language-action flow model for general robot control. In *Robotics: Science and Systems*, 2025a. doi: 10.15607/RSS.2025.XXI.010. URL https://www.roboticsproceedings.org/rss21/p010.html.

Kevin Black, Manuel Y. Galliker, and Sergey Levine. Real-time execution of action chunking flow policies. In *Advances in Neural Information Processing Systems*, 2025b. URL https://arxiv.org/abs/2506.07339.

Jun Cen, Chaohui Yu, Hangjie Yuan, Yuming Jiang, Siteng Huang, Jiayan Guo, Xin Li, Yibing Song, Hao Luo, Fan Wang, Deli Zhao, and Hao Chen. WorldVLA: Towards autoregressive action world model. *arXiv preprint arXiv:2506.21539*, 2025. URL https://arxiv.org/abs/2506.21539.

Liang Chen, Haozhe Zhao, Tianyu Liu, Shuai Bai, Junyang Lin, Chang Zhou, and Baobao Chang. An image is worth 1/2 tokens after layer 2: Plug-and-play inference acceleration for large vision-language models. In *European Conference on Computer Vision*, 2024. URL https://arxiv.org/abs/2403.06764.

Tianxing Chen, Zanxin Chen, Baijun Chen, Zijian Cai, Yibin Liu, Zixuan Li, Qiwei Liang, Xianliang Lin, Yiheng Ge, Zhenyu Gu, Weiliang Deng, Yubin Guo, Tian Nian, Xuanbing Xie, Qiangyu Chen, Kailun Su, Tianling Xu, Guodong Liu, Mengkang Hu, Huan-ang Gao, Kaixuan Wang, Zhixuan Liang, Yusen Qin, Xiaokang Yang, Ping Luo, and Yao Mu. RoboTwin 2.0: A scalable data generator and benchmark with strong domain randomization for robust bimanual robotic manipulation. *arXiv preprint arXiv:2506.18088*, 2025. URL https://arxiv.org/abs/2506.18088.

Cheng Chi, Siyuan Feng, Yilun Du, Zhenjia Xu, Eric Cousineau, Benjamin Burchfiel, and Shuran Song. Diffusion policy: Visuomotor policy learning via action diffusion. In *Robotics: Science and Systems*, 2023. URL https://arxiv.org/abs/2303.04137v4.

Zhenbang Du, Yonggan Fu, Lifu Wang, Jiayi Qian, Xiao Luo, and Yingyan Celine Lin. Fewer denoising steps or cheaper per-step inference: Towards compute-optimal diffusion model deployment. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2025. URL https://arxiv.org/abs/2508.06160.

Senyu Fei, Siyin Wang, Junhao Shi, Zihao Dai, Jikun Cai, Pengfang Qian, Li Ji, Xinzhe He, Shiduo Zhang, Zhaoye Fei, Jinlan Fu, Jingjing Gong, and Xipeng Qiu. LIBERO-Plus: A progressive robustness benchmark for visual-language-action models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2026. URL https://openaccess.thecvf.com/content/CVPR2026/papers/Fei_LIBERO-Plus_A_Progressive_Robustness_Benchmark_for_Visual-Language-Action_Models_CVPR_2026_paper.pdf.

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. In *International Conference on Learning Representations*, 2026a. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/2712b17bb58ea5b2b65c45857b024744-Abstract-Conference.html.

Kangye Ji, Yuan Meng, Jianbo Zhou, Ye Li, Chen Tang, and Zhi Wang. Test-time sparsity for extreme fast action diffusion. *arXiv preprint arXiv:2605.13316*, 2026b. URL https://arxiv.org/abs/2605.13316.

Kangye Ji, Jianbo Zhou, Yuan Meng, Ye Li, Hanyun Cui, and Zhi Wang. Sparse ActionGen: Accelerating diffusion policy with real-time pruning. In *International Conference on Machine Learning*, 2026c. URL https://arxiv.org/abs/2601.12894.

Wenqi Jiang, Jason Clemons, Karu Sankaralingam, and Christos Kozyrakis. How fast can I run my VLA? demystifying VLA inference performance with VLA-Perf. *arXiv preprint arXiv:2602.18397*, 2026. URL https://arxiv.org/abs/2602.18397.

Kumara Kahatapitiya, Haozhe Liu, Sen He, Ding Liu, Menglin Jia, Chenyang Zhang, Michael S. Ryoo, and Tian Xie. Adaptive caching for faster video generation with diffusion transformers. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2025. URL https://arxiv.org/abs/2411.02397.

Moo Jin Kim, Yihuai Gao, Tsung-Yi Lin, Yen-Chen Lin, Yunhao Ge, Grace Lam, Percy Liang, Shuran Song, Ming-Yu Liu, Chelsea Finn, and Jinwei Gu. Cosmos Policy: Fine-tuning video models for visuomotor control and planning. *arXiv preprint arXiv:2601.16163*, 2026. URL https://arxiv.org/abs/2601.16163.

Lin Li, Qihang Zhang, Yiming Luo, Shuai Yang, Ruilin Wang, Fei Han, Mingrui Yu, Zelin Gao, Nan Xue, Xing Zhu, Yujun Shen, and Yinghao Xu. Causal world modeling for robot control. *arXiv preprint arXiv:2601.21998*, 2026a. URL https://arxiv.org/abs/2601.21998.

Ye Li, Jiahe Feng, Yuan Meng, Kangye Ji, Chen Tang, Xinwan Wen, Shutao Xia, Zhi Wang, and Wenwu Zhu. TS-DP: Reinforcement speculative decoding for temporal adaptive diffusion policy acceleration. *arXiv preprint arXiv:2512.15773*, 2025. URL https://arxiv.org/abs/2512.15773.

Ye Li, Huanan Liu, Kangye Ji, Yuan Meng, Jiajun Fan, Yuansong Wang, Shiyu Qin, Chenglei Wu, Shu-Tao Xia, and Zhi Wang. ElegantVLA: Learning when to think for efficient vision-language-action models. *arXiv preprint arXiv:2605.29438*, 2026b. URL https://arxiv.org/abs/2605.29438.

Ye Li, Yuan Meng, Zewen Sun, Kangye Ji, Chen Tang, Jiajun Fan, Xinzhu Ma, Shu-Tao Xia, Zhi Wang, and Wenwu Zhu. SP-VLA: A joint model scheduling and token pruning approach for VLA model acceleration. In *International Conference on Learning Representations*, 2026c. URL https://arxiv.org/abs/2506.12723.

Ye Li, Chen Tang, Yuan Meng, Jiajun Fan, Zenghao Chai, Xinzhu Ma, Zhi Wang, and Wenwu Zhu. PRANCE: Joint token-optimization and structural channel-pruning for adaptive ViT inference. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 48(1):283–298, 2026d. doi: 10.1109/TPAMI.2025.3605239. URL https://ieeexplore.ieee.org/document/11146899/.

Yaron Lipman, Ricky T. Q. Chen, Heli Ben-Hamu, Maximilian Nickel, and Matt Le. Flow matching for generative modeling. In *International Conference on Learning Representations*, 2023. URL https://arxiv.org/abs/2210.02747.

Bo Liu, Yifeng Zhu, Chongkai Gao, Yihao Feng, Qiang Liu, Yuke Zhu, and Peter Stone. LIBERO: Benchmarking knowledge transfer for lifelong robot learning. In *Advances in Neural Information Processing Systems*, 2023. URL https://arxiv.org/abs/2306.03310.

Feng Liu, Shiwei Zhang, Xiaofeng Wang, Yujie Wei, Haonan Qiu, Yuzhong Zhao, Yingya Zhang, Qixiang Ye, and Fang Wan. Timestep embedding tells: It's time to cache for video diffusion model. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2025a. URL https://arxiv.org/abs/2411.19108.

Jiacheng Liu, Chang Zou, Yuanhuiyi Lyu, Junjie Chen, and Linfeng Zhang. From reusing to forecasting: Accelerating diffusion models with TaylorSeers. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 15853–15863, 2025b. URL https://arxiv.org/abs/2503.06923.

Jinming Lou, Wenyang Luo, Yufan Liu, Bing Li, Xinmiao Ding, Weiming Hu, Yuming Li, and Chenguang Ma. Token caching for diffusion transformer acceleration. *arXiv preprint arXiv:2409.18523*, 2024. URL https://arxiv.org/abs/2409.18523.

Zhengyao Lyu, Chenyang Si, Junhao Song, Zhenyu Yang, Yu Qiao, Ziwei Liu, and Kwan-Yee K. Wong. FasterCache: Training-free video diffusion model acceleration with high quality. In *International Conference on Learning Representations*, 2025. URL https://arxiv.org/abs/2410.19355.

Xinyin Ma, Gongfan Fang, and Xinchao Wang. DeepCache: Accelerating diffusion models for free. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2024. URL https://arxiv.org/abs/2312.00858.

Yunchao Ma, Yizhuang Zhou, Yunhuan Yang, Tiancai Wang, and Haoqiang Fan. Running VLAs at real-time speed. *arXiv preprint arXiv:2510.26742*, 2025. URL https://arxiv.org/abs/2510.26742.

NVIDIA. *Matrix Multiplication Background User's Guide*, 2023. URL https://docs.nvidia.com/deeplearning/performance/dl-performance-matrix-multiplication/index.html. Accessed September 26, 2026.

Yongming Rao, Wenliang Zhao, Benlin Liu, Jiwen Lu, Jie Zhou, and Cho-Jui Hsieh. DynamicViT: Efficient vision transformers with dynamic token sparsification. In *Advances in Neural Information Processing Systems*, 2021. URL https://arxiv.org/abs/2106.02034.

Yichao Shen, Fangyun Wei, Zhiying Du, Yaobo Liang, Yan Lu, Jiaolong Yang, Nanning Zheng, and Baining Guo. VideoVLA: Video generators can be generalizable robot manipulators. In *Advances in Neural Information Processing Systems*, 2025. URL https://arxiv.org/abs/2512.06963.

Mustafa Shukor, Dana Aubakirova, Francesco Capuano, Pepijn Kooijmans, Steven Palma, Adil Zouitine, Michel Aractingi, Caroline Pascal, Martino Russi, Andres Marafioti, Simon Alibert, Matthieu Cord, Thomas Wolf, and Remi Cadene. SmolVLA: A vision-language-action model for affordable and efficient robotics. *arXiv preprint arXiv:2506.01844*, 2025. URL https://arxiv.org/abs/2506.01844.

Yuran Wang, Siqiao Huang, Mingleyang Li, Chenhao Zhang, Jiaqi Liang, Weiyang Jin, Yue Chen, Xuemin Chi, Donghao Zhou, Qize Yu, Yu-Kai Wang, Yuhan Rui, Shenzhe Yao, Zhen Yuan, Zhenhao Shen, Kefei Zhu, Zijie Zhu, Ning Gao, Xiaowei Chi, Guanqi He, Shanghang Zhang, Hao Dong, Lin Shao, and Hang Zhao. OpenWAM: An open, modular exploration towards systematic world-action model pretraining. *arXiv preprint arXiv:2609.07398*, 2026. URL https://arxiv.org/abs/2609.07398.

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*, 2025. URL https://arxiv.org/abs/2502.02175.

Yantai Yang, Yuhao Wang, Zichen Wen, Luo Zhongwei, Chang Zou, Zhipeng Zhang, Chuan Wen, and Linfeng Zhang. EfficientVLA: Training-free acceleration and compression for vision-language-action models. In *Advances in Neural Information Processing Systems*, volume 38, pp. 40891–40914, 2025. doi: 10.52202/085713-1365. URL https://arxiv.org/abs/2506.10100.

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. URL https://arxiv.org/abs/2603.16666.

Xuanlei Zhao, Xiaolong Jin, Kai Wang, and Yang You. Real-time video generation with pyramid attention broadcast. In *International Conference on Learning Representations*, 2025. URL https://arxiv.org/abs/2408.12588.

Chang Zou, Xuyang Liu, Ting Liu, Siteng Huang, and Linfeng Zhang. Accelerating diffusion transformers with token-wise feature caching. In *International Conference on Learning Representations*, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/bbe024e0517fe12ac3a8d388b19ff9fe-Abstract-Conference.html.

A Real-World Setup and Evaluation

A.1 Platforms and tasks

We use a Franka Research 3 robot and a RealSense D435i camera. Figure 8 shows the three setups used for our five tasks. The conveyor runs at 1 m/min for shuttlecock stacking, carrot placement, and tennis-ball placement. The rolling-can task uses an inclined tabletop with one edge raised by 5 cm; mug hanging uses a level tabletop.

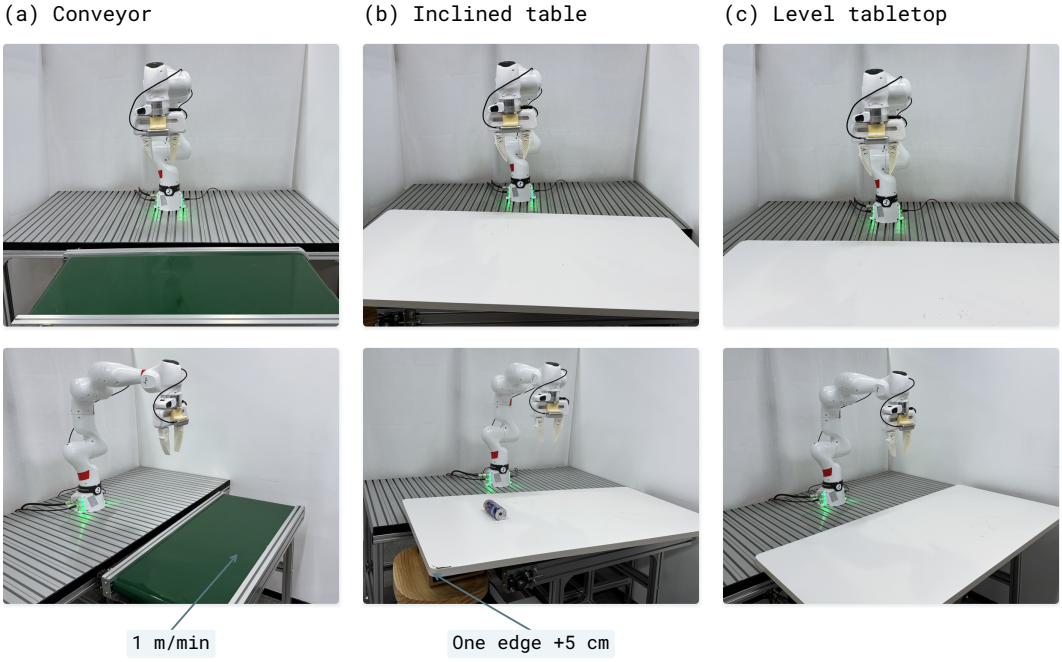


Figure 8: **Real-world experimental setups.** Front views (top) and oblique views (bottom) of the three setups. (a) Conveyor. (b) Inclined tabletop. (c) Level tabletop.

A.2 Evaluation protocol

We compare native inference with RealtimeWAM on both FastWAM and OpenWAM. Each task uses 50 trials per backbone and inference configuration, with the same synchronous inference and action execution policy across the comparison. Each inference request predicts a chunk of 32 future actions. The robot executes actions at a control frequency of 20 Hz. Inference runs locally, with negligible network transmission latency. Table 5 lists the task goals; its task IDs match the rollout rows in Figure 7. Table 4 reports success rate and measured mean inference latency; its averages weight the five tasks equally.

In our conveyor experiments, task success was substantially lower when measured inference latency exceeded approximately 90 ms. The dashed line in Figure 1 marks this empirically observed latency threshold.

B Evaluation Protocols and Detailed Results

B.1 Benchmarks and comparison settings

Benchmarks. We evaluate FastWAM and OpenWAM on RoboTwin 2.0 (Chen et al., 2025), LIBERO (Liu et al., 2023), and LIBERO-Plus (Fei et al., 2026). RoboTwin Clean and Random are reported separately, giving four settings in Table 1. LIBERO and LIBERO-Plus each report Spatial, Object, Goal, and Long categories; their individual success rates appear in Table 6.

Table 5: **Real-world task goals and trial counts.** Trials are counted per task, backbone, and inference configuration.

Task	Setup	Task goal	Trials
A: Three shuttlecocks	Conveyor	Stack the three shuttlecocks together.	50
B: Carrot to plate	Conveyor	Move the carrot from the belt onto the plate.	50
C: Hang mug	Level tabletop	Hang the mug by its handle on the rack.	50
D: Tennis ball to bowl	Conveyor	Move the tennis ball from the belt into the bowl.	50
E: Rolling can to bowl	Inclined table	Intercept the rolling can and place it in the bowl.	50

Evaluation protocol. We use the benchmark environments' task-success checks. A rollout succeeds if the task goal is satisfied within the evaluation horizon; otherwise it fails. RoboTwin uses 100 rollouts per task in each of its Clean and Random settings. LIBERO uses 50 rollouts per task with the released initial-state sets and goal predicates; Long denotes `libero_10`. LIBERO-Plus uses one rollout per perturbed task variant, covering changes to layout, cameras, robot initial state, language, lighting, background, and sensor noise (Fei et al., 2026). These protocols apply to both backbones and all methods in Table 1.

Comparison settings. We use the officially released fine-tuned checkpoints of FastWAM and OpenWAM. For each backbone and evaluation setting, Native, the caching baselines, and Realtime-WAM share the same model weights. All methods use the same evaluation seeds and initial scenes within each backbone and evaluation setting, enabling paired comparisons. Each backbone's original Native execution provides the reference. VLA-Cache, TeaCache, ToCa, BAC, and RealtimeWAM use compilation and CUDA Graph replay for the main-table latency comparison. The caching baselines retain their original execution schedules, while RealtimeWAM incorporates the proposed parallel execution. Appendix C.3 compares original and compiled execution, including Native. Runtime measurements use BF16, batch size one, and a modified 48 GiB RTX 4090.

B.2 Metrics and aggregation

For an evaluated task or task variant q, let n_q be the number of rollouts and $y_{q,i} \in \{0, 1\}$ the benchmark's success indicator. Its success rate is $\mathrm{SR}_q = 100\, n_q^{-1} \sum_{i=1}^{n_q} y_{q,i}$. RoboTwin averages task success rates within each setting. LIBERO and LIBERO-Plus average task or task-variant success rates within each category; their main-table scores are the arithmetic means of the four categories. For each setting, latency is the mean of measured request times collected during complete closed-loop rollouts. Each request uses its selected token set and Transformer evaluation budget. The execution comparisons in Tables 2, 7, and 9 use the same rollout-based mean latency definition for each evaluated configuration.

Timing runs from prepared CPU images and robot state to denormalized CPU actions. It includes input processing, model and sampler execution, transfers, and synchronization. Compilation, graph capture, warmup, profiling, and robot execution are excluded.

Tables 1 and 9 average latency equally across the four settings; Table 1 also averages success equally across these settings. Speedup is the original Native average latency divided by the method's average latency, computed before rounding. Table 3 reports measured first-forward latency (First) and mean request latency over complete rollouts (Request). All latencies are reported in milliseconds.

B.3 Component ablation settings

Table 3 evaluates variants of the full OpenWAM method on RoboTwin. Token ablations keep the motion gate unchanged; motion ablations keep the token policy unchanged.

Token reuse and selection. *w/o token reuse* refreshes all observation positions. *Random tokens* retains the selected execution capacity b_t and uniformly samples that many positions without replace-

ment from all M positions. *w/o capacity filling* refreshes r_t positions instead of b_t, retaining required positions and feature ranking.

Motion gate. *Fixed 2F* and *Fixed 4F* use two and four Transformer evaluations, respectively, within the ten Euler updates. Their token refresh counts follow the observation. *Random gate* samples between the two budgets using the original motion gate's usage proportions, independently of predicted motion.

B.4 CATEGORY-LEVEL BENCHMARK RESULTS

Table 6 gives the category-level success rates underlying the benchmark summaries in Table 1. The RoboTwin average combines Clean and Random; each LIBERO or Plus average combines Spatial, Object, Goal, and Long with equal weight.

Table 6: **Category-level success rates (%).** Each backbone uses the same methods and success-rate evaluations as Table 1. Plus denotes LIBERO-Plus.

(a) FastWAM

Benchmark	Subset	Native	VLA-Cache	TeaCache	ToCa	BAC	**Ours**
RoboTwin	Clean	91.88	80.00	81.42	91.35	91.67	**92.04**
	Random	91.78	83.20	90.86	91.24	89.53	**90.50**
	Avg. SR	91.83	81.60	86.14	91.30	90.60	**91.27**
LIBERO	Spatial	98.20	98.00	97.80	94.40	98.20	**96.20**
	Object	100.00	100.00	99.80	98.80	100.00	**99.60**
	Goal	97.00	96.00	96.20	93.20	96.80	**97.60**
	Long	95.20	94.00	94.40	91.80	95.00	**95.60**
	Avg. SR	97.60	97.00	97.05	94.55	97.50	**97.25**
Plus	Spatial	52.00	52.80	53.20	53.00	54.60	**53.60**
	Object	69.80	68.80	69.20	69.20	71.00	**71.00**
	Goal	37.00	35.60	36.40	36.40	38.80	**38.00**
	Long	40.40	37.20	38.00	38.00	41.80	**42.20**
	Avg. SR	49.80	48.60	49.20	49.15	51.55	**51.20**

(b) OpenWAM

Benchmark	Subset	Native	VLA-Cache	TeaCache	ToCa	BAC	**Ours**
RoboTwin	Clean	93.64	93.36	90.42	93.42	93.52	**92.78**
	Random	94.06	89.94	94.12	93.98	91.96	**94.02**
	Avg. SR	93.85	91.65	92.27	93.70	92.74	**93.40**
LIBERO	Spatial	99.60	98.80	99.00	98.40	99.20	**99.40**
	Object	99.60	99.40	99.40	99.20	99.60	**99.60**
	Goal	99.80	98.60	98.80	98.20	99.00	**99.20**
	Long	98.20	96.80	97.20	96.60	97.80	**97.80**
	Avg. SR	99.30	98.40	98.60	98.10	98.90	**99.00**
Plus	Spatial	80.80	76.80	80.80	80.80	84.80	**81.40**
	Object	64.00	59.80	64.00	64.00	60.00	**65.80**
	Goal	42.20	42.20	46.40	46.20	54.00	**39.60**
	Long	72.00	72.00	64.00	63.80	68.00	**68.60**
	Avg. SR	64.75	62.70	63.80	63.70	66.70	**63.85**

C Execution Efficiency

C.1 Execution settings and fixed budgets

Table 2 cumulatively adds CUDA Graph replay, compilation, parallel execution, and token reuse. First starts from CPU images and state and ends after input processing, Prefill, and the first complete predictive evaluation. It includes selection, gathering, cache writes, and stream joins. First is measured in 2F runs; Full 10F uses separate complete requests through denormalized CPU action return. Both report mean wall-clock latency measured during complete closed-loop rollouts, excluding compilation, capture, warmup, and profiling.

Table 7 compares dense execution at two, four, and ten Transformer evaluations within ten Euler updates. Measurements use BF16 on the modified 48 GiB RTX 4090, PyTorch 2.7.1/CUDA 12.8, an Intel Core Ultra 9 285K, four PyTorch threads, and CPU affinity 0–7. OpenWAM's Native baseline uses commit `a8992b613fdb`; compilation uses `reduce-overhead` with verified graph replay. Parallel execution also reuses eligible language projections, preallocates attention buffers, and overlaps branches at their dependencies.

OpenWAM timing begins with composed CPU PIL images and state; FastWAM begins with three RGB arrays and state, including composition and resize. Both include input preparation, VAE and context processing, model evaluations, sampler updates, transfers, synchronization, and CPU action return. Prompt embeddings are warm. Sensor acquisition, communication, robot execution, setup, and output-video decoding are excluded.

Table 7: **Dense execution controls at fixed evaluation budgets.** Mean request latency measured during complete closed-loop rollouts in ms, following the main-text timing protocol. All three execution variants compute full FFNs.

Model	N	Native	Compile + Graph	Parallel
OpenWAM	10	672.83	450.90	223.27
OpenWAM	4	240.53	205.49	101.95
OpenWAM	2	127.31	109.09	59.11
FastWAM	10	214.33	99.68	33.52
FastWAM	4	104.61	50.27	27.99
FastWAM	2	71.55	38.69	21.56

At ten evaluations, parallel execution reduces Compile + Graph latency from 450.90 to 223.27 ms on OpenWAM and from 99.68 to 33.52 ms on FastWAM.

C.2 Numerical agreement

Table 8 compares Parallel and Native predictions at matched budgets. FastWAM checks vary state, instruction, images, and noise; direct execution and graph replay agree bitwise on these cases. OpenWAM's official and modified Native controls also agree bitwise.

Table 8: **Parallel versus native output checks at matched budgets.** Mean case MAE averages recorded action MAEs; Max is the largest coordinate difference. Each model uses its own effective action space.

Model	N	Cases	Mean case MAE	Max
OpenWAM	10	3	0.001288	0.007812
OpenWAM	4	3	0.001245	0.011719
OpenWAM	2	3	0.002375	0.031250
FastWAM	10	8	0.001449	0.015625
FastWAM	4	8	0.001384	0.015625
FastWAM	2	8	0.002515	0.031250

C.3 COMPILED CACHING BASELINES

Table 9 compares original execution with compilation and CUDA Graph replay under the main-table evaluation protocol. Native remains the original implementation when computing main-table speedup.

Table 9: **Effect of compilation and CUDA Graph replay on inference latency.** Original denotes each method's original implementation; + C/G enables compilation and CUDA Graph replay with its execution schedule preserved. All methods report measured rollout-mean latencies (ms), with task settings, timing boundaries, and four-setting averaging following Table 1. Native uses the 10F results in Table 7. Reduction is the percentage decrease from Original to + C/G.

Method	FastWAM			OpenWAM		
	Original	+ C/G	Reduction	Original	+ C/G	Reduction
Native	214.33	99.68	53.5%	672.83	450.90	33.0%
VLA-Cache	236.13	115.55	51.1%	743.32	400.26	46.2%
TeaCache	112.85	49.28	56.3%	362.31	245.99	32.1%
ToCa	225.50	102.66	54.5%	719.60	440.42	38.8%
BAC	151.17	67.14	55.6%	460.04	305.08	33.7%

C.4 PREPARATION SCHEDULING

The historical OpenWAM step-600 profiling runtime uses BF16 on the 48 GiB RTX 4090 with PyTorch 2.7.1/CUDA 12.8 (snapshot `5f9dede`, upstream `898e2f9`). It refreshes 80 observation positions, computes all future-video and action positions, and uses two Transformer evaluations within ten Euler updates. Repeated inputs and $\alpha = 0$, $\mathcal{B} = \{80, 120\}$ fix the refresh count; online selection and filling still execute. Two observations are each repeated for 50 rounds with three interleaved schedules, restoring dense history and noise before every call. Timing covers composed CPU images/state through CPU actions, including preparation, selection, cache updates, transfers, and waits; restoration and warmup are excluded.

Table 10: **Preparation scheduling at a fixed 80-token budget.** Historical OpenWAM profiling, $N = 2$. Each row summarizes 100 requests in ms. First and later are GPU-event medians; request percentiles use wall time.

Preparation schedule	First	Later	Request P50	P90	P95
Sequential	23.06	20.06	51.32	51.65	51.79
VAE/condition overlap	23.06	20.08	50.55	50.91	50.97
Overlap + early RGB	22.97	20.05	50.26	50.62	50.73

The schedules select identical positions and return bitwise-identical actions. Paired median savings are 0.778 ms for VAE/condition overlap and 0.274 ms for early RGB routing, with round-bootstrap 95% intervals [0.742, 0.822] and [0.225, 0.324] ms.

D TOKEN REUSE AND SELECTION

D.1 CACHE STATE AND FEATURE CORRESPONDENCE

The observation cache stores ungated FFN outputs for 120 positions at each layer. Layer 0 is dense; layers 1–29 share a refresh set. Attention uses current features at every position, and current residual gates apply to fresh and cached FFN outputs. Future-video and action branches are complete at every requested evaluation. Selected cache entries update; others retain their last refreshed outputs. The layer-0 selector reference advances at each observation. Episode boundaries reset history; changes to weights, conditioning, camera layout, or preprocessing require compatible cache state or full refresh.

We compare 14 observation pairs from one FastWAM trajectory and 44 pairs from four OpenWAM `adjust_bottle` trajectories, eight environment steps apart. FFN change at each token is the channelwise mean absolute or L2 difference between dense features. Mean within-frame RGB/FFN-MAE correlations are 0.827/0.817 for FastWAM/OpenWAM at layer 0 and 0.716/0.701 across layers 1–29. Figures 10 and 11 show typical and weaker correspondence: small RGB changes can accompany larger feature changes. Each model and metric uses the same color range across examples.

For the 44 OpenWAM pairs, a dense-history probe selects the 56 positions with the smallest layer-0 change and compares them with later layers. Change is measured by symmetric relative L1 distance, $2\|x - y\|_1 / \max(\|x\|_1 + \|y\|_1, 10^{-12})$. The mean observation-set overlap is 85.4%, versus 46.7% by chance, supporting a shared selection across observation layers.

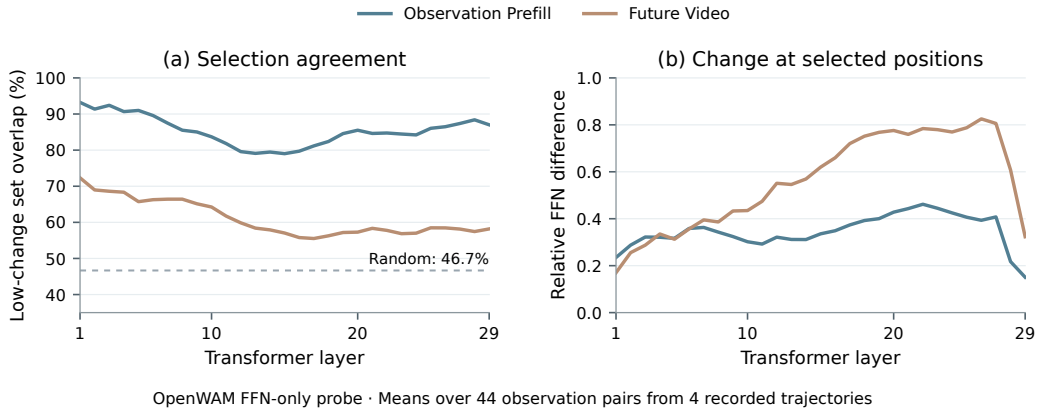


Figure 9: **Stability of a selection across layers.** (a) Overlap with each layer's lowest-change set. (b) Relative FFN change at selected positions. Curves average 44 observation pairs. The future-video branch is a separate diagnostic selection; its mean overlap is 61.0%.

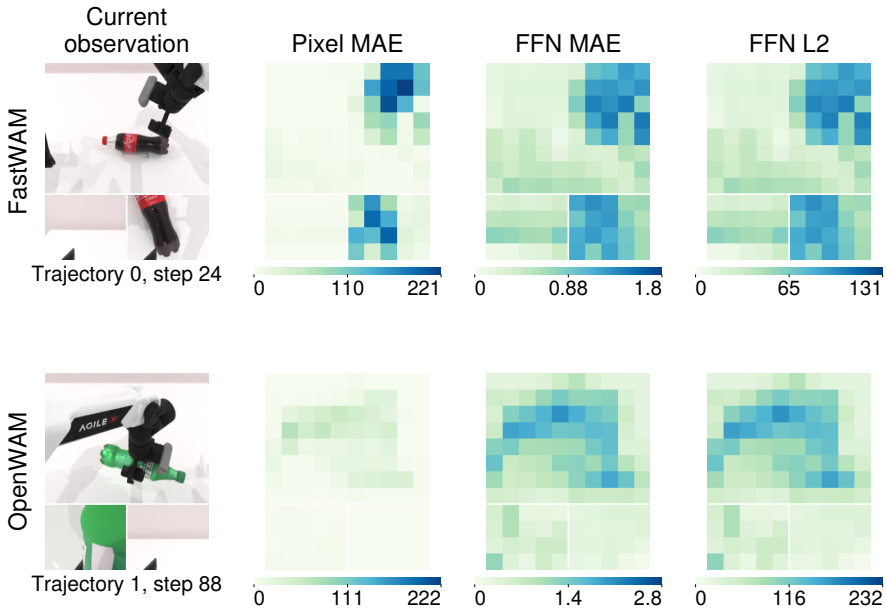


Figure 10: **RGB and layer-0 FFN changes: typical examples.** Observation pairs are eight environment steps apart. Camera regions are spatially aligned; each model and metric uses a fixed color range.

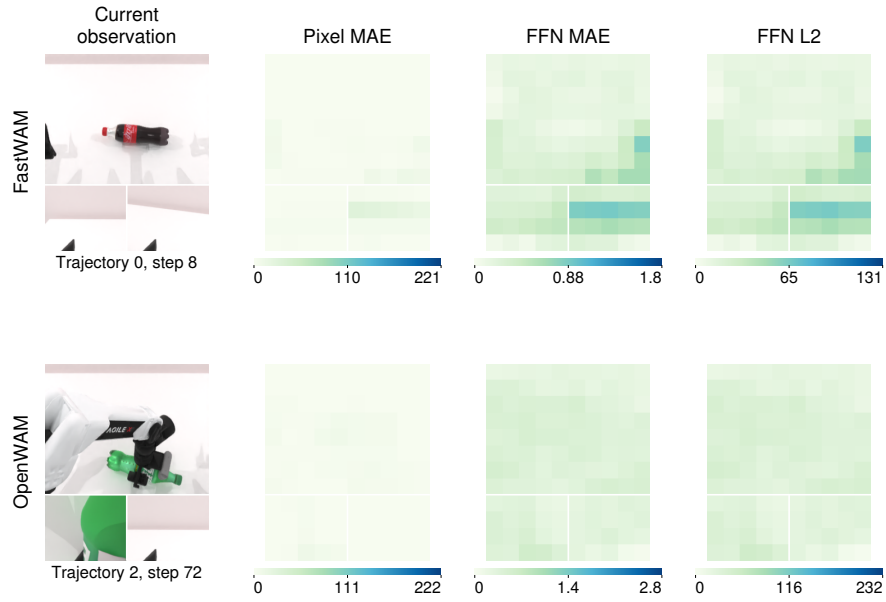


Figure 11: **Examples with weaker spatial correspondence.** Color ranges match Figure 10; low pixel change does not always imply low feature change.

D.2 SELECTION AT MATCHED REFRESH COUNTS

Figure 12 compares RGB-guided and random selection on one simulation pair and one real-video pair. Each comparison fixes refresh count, cache history, inputs, noise, and the two Transformer evaluations at updates 0 and 3. Layer 0 and attention remain complete. Action MAE is measured against the same-input dense prediction over the 32-action chunk, using the model's active normalized coordinates.

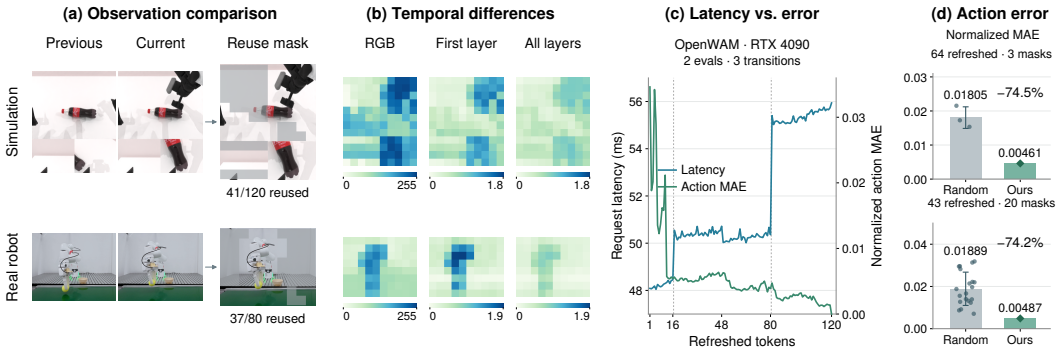


Figure 12: **Token selection and reuse diagnostics. (a,b)** Simulation and real-video observations, reuse masks, and RGB/feature changes. **(c)** Historical OpenWAM profiling: request latency and normalized action MAE versus refreshed-token count at $N = 2$. **(d)** Random versus RGB-guided selection at matched counts; bars show mean action MAE, dots show individual masks, and whiskers show one standard deviation.

Real video. The tennis-ball pair is sampled at 14.6 and 15.0 s from a 120-fps recording, selected by RGB MAE among candidate pairs 0.4 s apart. An 1800×1440 crop is resized to 320×256 in the front-camera slot, with other camera rows fixed black. The step-3,500 OpenWAM checkpoint is fine-tuned on the five robot tasks. Both frames use fixed proprioception from frame 294 of a recorded tennis-ball trajectory, training normalization, and the instruction "Pick up the tennis ball from the conveyor and place it into the bowl." BF16 feature capture and replay use A100. Guided and 20 random masks each reuse 37 of 80 visible positions plus 40 padded positions. Mean action MAE over ten active coordinates is 0.004866 for guided selection and 0.018892 for random selection (74.2% lower).

Simulation. The released three-camera FastWAM RoboTwin checkpoint uses transition `fast_ep0_obs3` in A100 eager mode. Guided and three random masks each refresh 64 of 120 positions. MAE over 14 normalized joint-action coordinates is 0.004609 for guided selection and 0.018048 for random masks on average (74.5% lower). Full refresh matches dense computation bitwise.

Spatial masks. Panels a,b use an RGB-only diagnostic rule on the 12×10 grid: refresh positions above $5/255$ mean absolute RGB change, then protect the top quarter of remaining positions by a camera-local smoothed score. Smoothing uses fourfold downsampling and a Gaussian with standard deviation four and reflected boundaries. These displayed masks reuse 41/120 simulation positions and 37/80 visible real-camera positions. Panel d uses the fixed counts above. The full selector is evaluated through Random tokens in Table 3.

D.3 CAPACITY FILLING AND RECURSIVE REUSE

Refresh-count sweep. Figure 12c uses the historical step-600 OpenWAM profiling runtime on RTX 4090 with two Transformer evaluations. Three observation pairs (frames 1598/1600, 2798/2800, and 3998/4000 of a 30-FPS recording) are evaluated at all 120 refresh counts with five timing rounds per pair. Each call restores dense history and noise. RGB-required positions precede other positions, ranked within each group by layer-0 feature change. The three pairs have 1, 0, and 12 required positions; smaller budgets omit required positions in this fixed-budget diagnostic.

The 3,585 timed calls interleave target counts with an 80-token control. Each point uses the control median plus the median paired latency difference. Timing includes online selection, packing, cache writes, transfers, and waits; restoration, warmup, and error analysis are excluded. Action MAE averages the 32×10 normalized action coordinates over three pairs relative to full refresh. That reference has mean MAE 0.00113 against the original dense engine.

Reducing refresh from 120 to 80 changes First from 36.41 to 30.95 ms and Request from 55.94 to 50.36 ms. Within the middle plateau, 32-to-80 filling reduces MAE from 0.00503 to 0.00389 (22.7%), with request latencies 50.71 and 50.36 ms; individual error increments are not monotonic. Table 3a separately marks measured First at 65 and 80 tokens (30.76/30.96 ms). Its Request column contains complete-rollout means (61.78/63.76 ms).

Recursive reuse. Table 11 covers 44 transitions from four OpenWAM `adjust_bottle` trajectories. Each starts with full initialization, followed by 11 updates eight environment steps apart. Recursive variants carry cache history and use their own feature anchors. An age cap of two permits at most two consecutive reuses. Errors are measured against dense FFN computation over the normalized 32×20 active action chunk. At 64 refreshed positions, an age cap increases mean MAE from 0.002571 to 0.004006; increasing refresh to 96 with the cap gives 0.001596.

Table 11: **Controlled OpenWAM token reuse in Prefill.** All 44 transitions, $N = 2$; one L0-feature mask per observation. Mean and worst-observation MAE and maximum component error use normalized active actions. Future-video and action branches remain dense.

History / selector	Refresh	Mean MAE	Worst MAE	Max. error
Dense previous history / L0	64/120	0.002722	0.022480	0.140625
Recursive / L0	64/120	0.002571	0.011384	0.269531
Recursive / L0 + age cap	64/120	0.004006	0.035773	0.473633
Recursive / L0 + RGB + age cap	64/120	0.003292	0.016858	0.273438
Recursive / L0 + age cap	96/120	0.001596	0.012808	0.117188

E MOTION-ADAPTIVE REFINEMENT

E.1 SAMPLER AND MOTION SCORE

Shared-prefix continuation. We integrate the predicted velocity field using Euler updates, following the flow-matching formulation (Lipman et al., 2023). For updates $i = 0, \dots, 9$, let $h_i = \mathrm{Embed}(x_i, s_i, c)$ be the current latent embedding, with timestep s_i and conditioning c. Writing B for the full Transformer, we cache its input-to-output residual

$$C_i = \begin{cases} B(h_i, s_i, c) - h_i, & i \in \mathcal{E}, \\ C_{i-1}, & i \notin \mathcal{E}. \end{cases} \tag{6}$$

The output head and Euler update are evaluated at every step:

$$\widetilde{v}_i = \mathrm{Head}(h_i + C_i, s_i), \qquad x_{i+1} = x_i + \Delta t_i \widetilde{v}_i. \tag{7}$$

The cache is initialized at update 0 of each request. The 2F schedule uses $\mathcal{E}_2 = \{0, 3\}$ and the 4F schedule uses $\mathcal{E}_4 = \{0, 3, 6, 9\}$; the 10F reference refreshes at every update. NFE counts full Transformer evaluations. All schedules share ten Euler updates and fixed observation and instruction conditioning within a request.

After the second Transformer refresh at update 3, we decode a coarse action prediction and measure its amplitude using Equation 5. If refinement is selected, the runtime continues with additional Transformer refreshes at updates 6 and 9, retaining the same initial noise and conditioning. OpenWAM caches the residuals of the future-video and action Transformers; FastWAM caches the action Transformer residual.

Coordinates and threshold. In the Cartesian-action probes, displacement is the maximum Euclidean distance from current end-effector XYZ across 32 predicted positions, using the training action statistics to restore metres. The joint-action FastWAM runtime obtains XYZ by forward kinematics before evaluating displacement. The recorded RoboTwin motion probes use a fixed 0.08 m threshold; rotation and gripper coordinates do not enter this score.

E.2 PAIRED PREDICTION COMPARISONS

Table 12: **Motion refinement ablation.** Means over 93 paired observations; green ✓/red ×: enabled/disabled. Refine adds two evaluations; the motion gate selects requests. XYZ deviations are relative to 10F; absolute values are in mm.

Policy	Components		Mean NFE	XYZ deviation ↓	
	Refine	Motion gate		Abs. (mm)	Relative
Fixed 2F	×	×	2.00	7.65	0.0869
Fixed 4F	✓	×	4.00	3.68	0.0280
Ours	✓	✓	**2.65**	**6.48**	**0.0468**

Deviation is the mean Euclidean XYZ error over both arms and the first 24 predicted steps. Relative deviation divides this error by the largest 10F displacement across those steps and arms, with a 0.02 m floor. Across 93 paired observations from one trajectory each of `stack_bowls_two`, `blocks_ranking_size`, and `put_bottles_dustbin` (18, 49, and 26 requests), we compare 2F and 4F with the same model's 10F output. Table 12 compares the fixed and adaptive budgets, weighting requests equally. The gate selects 4F for 30 of 93 requests at a coarse 2F displacement threshold of 0.08 m. For these 30 small-motion requests, mean relative deviation is 0.164 with 2F and 0.040 with 4F. For the other 63 requests, the corresponding means are 0.050 and 0.022. The absolute mean reductions are 3.62 mm and 4.14 mm, respectively. The larger relative reduction in the small-motion group partly reflects its smaller normalization denominator; its absolute 2F deviation is also lower.

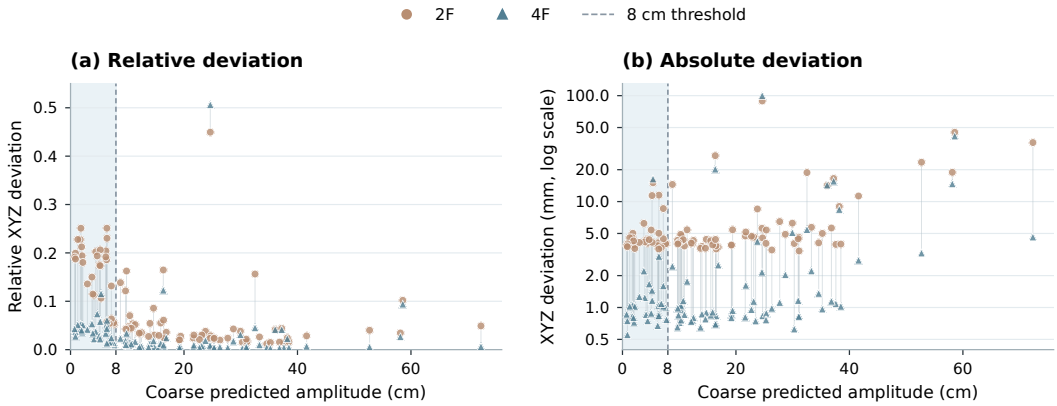


Figure 13: **Prediction deviations over all 93 paired observations.** Each request contributes a 2F and a 4F point at the same coarse predicted displacement, joined by a faint vertical line. Both compare with the same-input 10F prediction over the first 24 XYZ steps. Left: deviation normalized by the 10F amplitude, with a 0.02 m floor. Right: absolute mean XYZ deviation on a logarithmic scale, in mm. The dashed vertical line marks the 8 cm motion-gate threshold.

Figure 13 separates relative and absolute error. Small-motion requests show larger relative deviations under 2F, while the absolute-error view reveals both the role of normalization and variation across requests. Refinement reduces deviation from 10F on 90 of the 93 observations and increases it on the remaining three.

Table 13: **Fixed-budget action deviations on FastWAM.** Executed-prefix normalized MAE against the same-observation 10F prediction over 1,798 observations.

Evaluation positions	Mean	Median	P95	Max
$\{0, 3\}$	0.007304	0.006876	0.010497	0.027701
$\{0, 3, 6, 9\}$	0.001224	0.001186	0.001713	0.005981

A separate FastWAM replay covers 1,798 observations (Table 13). Four evaluations reduce mean executed-prefix action MAE against the 10F reference from 0.007304 to 0.001224, and P95 from 0.010497 to 0.001713. Fixed-budget comparisons share the observation, initial noise, integration schedule, and action normalization.

E.3 Visualization settings

Figure 3a displays requests 1–25 of the `put_bottles_dustbin` episode; all 26 requests enter the paired analysis. Predicted speed is end-effector displacement per control step. The display smooths 608 samples with a Gaussian ($\sigma = 4$ steps, radius 12, reflected endpoints), then retains every fourth sample. The two deviation strips use 2F/4F predictions against 10F, with the relative metric above and a common range of 0–0.25. Images are recorded request inputs. Panel b uses schematic paths and task-tolerance radii.

Figure 6 uses all 50 requests from one OpenWAM `blocks_ranking_size` rollout. At the 8 cm threshold, 30 requests use 2F and 20 use 4F. Token capacities are 16, 80, and 120, with full initialization. The episode ends in task failure.