ARE VISION-LANGUAGE AGENTS LATENCY-AWARE? A SYSTEMATIC EVALUATION UNDER ACTION LATENCY

Anonymous authors
Paper under double-blind review

ABSTRACT

Foundation-model agents, such as vision-language-action (VLA) policies, suffer from inference latency: the environment keeps changing while the policy computes. Faster inference reduces this delay but does not eliminate it, and training with a delay chosen in advance may not match the target deployment. Current benchmarks hide the problem by pausing the environment during inference; evaluated in real time on an RTX 3090 GPU, VLA policies on 12 environments retain a median of only 6.6% of their zero-latency return. We study how to estimate the residual latency of a deployment, adapt a policy to it, and determine when the adaptation transfers. We introduce a **latency profiling and replay framework** that models which delays occur and whether large delays occur on consecutive requests, then replays new delay sequences in simulation. (1) Across five GPU deployments, a Temporal profile reproduces hardware-timed return with an average gap of 10.8, versus 87.4 for a fixed mean delay. (2) Training under the target profile improves hardware-timed return in all 36 task–architecture pairs, by 16% to 86% of zero-latency return, while transfer across delays is asymmetric and task-dependent. (3) An explicit latency value can be essential when training across delays, whereas the benefit of visual history depends on the task. Residual deployment latency can be modeled and incorporated into policy training, although adaptation does not generalize uniformly across latency regimes.

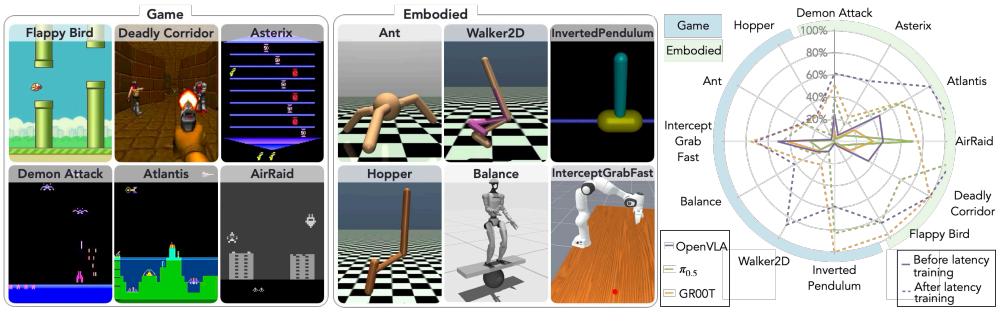

Figure 1: Training with the measured RTX 3090 latency profile improves real-time performance across games and embodied-control tasks. Left: the 12 evaluation environments. Right: OpenVLA, $\pi_{0.5}$, and GR00T before and after profile training. Each value reports the RTX 3090 real-time mean as a percentage of the original policy's zero-latency mean on the same task; 100% means equal performance.

1 INTRODUCTION

Foundation-model agents, such as vision-language-action (VLA) policies (Zitkovich et al., 2023; Ghosh et al., 2024; Kim et al., 2025b; Black et al., 2025b;a), map images and task instructions to actions. They suffer from *inference latency*: while the policy computes, the environment keeps changing. An enemy keeps moving, a humanoid keeps falling, and a rolling ball leaves the reach of a robot arm. Current benchmarks hide this problem because they pause the environment during inference. Without this pause, a delay of only two frames drops the success rate of VLA policies on MIKASA Intercept Grab Fast, where a robot arm must catch a rolling ball, from 50–64% to zero. Evaluated in real time on an RTX 3090 GPU, VLA policies on 12 environments keep a median of only 6.6% of their zero-latency return.

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.

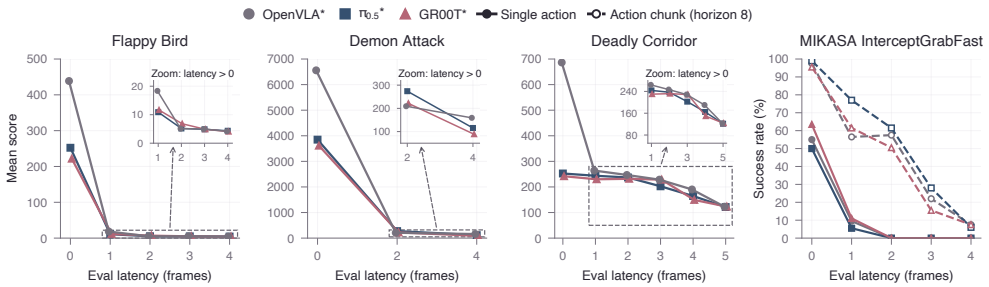

Figure 2: Policies trained without delay lose most of their performance when their actions are delayed by only a few frames. The horizontal axis is the fixed evaluation delay in raw frames; the insets rescale the nonzero-delay region. The three game panels report mean return, and MIKASA reports success rate. H is the number of actions predicted per model call. H1 uses supervised fine-tuning, whereas H8 also uses Dataset Aggregation (DAgger), so the H1–H8 difference does not isolate action-chunk length. Asterisks mark StarVLA implementations (Appendix C).

Inference acceleration, such as quantization (Frantar et al., 2023), speculative decoding (Leviathan et al., 2023), action chunking (Zhao et al., 2023), and asynchronous execution (Black et al., 2025c), reduces latency, but nonzero delay remains. Delayed decision processes and real-time reinforcement learning (RL) formalize acting under delay (Katsikopoulos & Engelbrecht, 2003; Ramstedt & Pal, 2019), and recent VLA methods train policies with delays chosen in advance (Zhu et al., 2026; Huang et al., 2026; Gao et al., 2026). A chosen delay, however, may not reflect which latencies occur in the target deployment, how often they occur, or whether slow requests occur consecutively. Systems engineering addresses the same issue for networks by recording and replaying real traces (Netravali et al., 2015) and designing for the slowest requests (Dean & Barroso, 2013). We therefore pursue a complementary goal, **latency awareness**: explicitly modeling the residual latency of a deployment and using that model during policy training or action selection. We ask whether a profile estimated from measured traces can reproduce the effect of deployment latency, whether a policy can adapt to this profile and transfer when latency changes, and what information supports adaptation when a single image reveals neither the delay nor how the scene changed.

To reproduce deployment latency, we introduce a **latency profiling and replay framework**. It measures *action latency*, the time from capturing an observation until its predicted action is ready, and fits a timing model called a *latency profile*. The profile estimates which delays occur and whether slow requests occur consecutively; it does not predict the runtime of the next request. *Latency replay* samples new delay sequences from the profile during simulated interaction, reproducing timing rather than recorded actions. Among four profile models, a *temporal* profile that reproduces the consecutive slow requests observed in measured traces comes closest to hardware-timed returns across five GPU deployments, with an average gap of 10.8, versus 87.4 for the fixed-mean profile.

To study adaptation, we train a small RL teacher under replayed latency and distill it into three VLA architectures, OpenVLA (Kim et al., 2025b), $\pi_{0.5}$ (Black et al., 2025a), and GR00T (NVIDIA et al., 2025), in their StarVLA implementations (StarVLA Community, 2026), on 12 environments spanning games (Bellemare et al., 2013; Kempka et al., 2016) and embodied control (Todorov et al., 2012; Towers et al., 2024; Cherepanov et al., 2026a; Sferrazza et al., 2024). The central test is whether a profile learned from the target machine can reproduce the effect of that machine's latency during training. Policies trained in simulation under the target profile improve hardware-timed return on the RTX 3090 in all 36 task–architecture pairs, by 16% to 86% of the zero-latency return. Transfer across delays is asymmetric and task-dependent: on Demon Attack, a policy trained at 16 frames retains 85% of the same-delay reference at 8 frames, whereas the reverse retains only 27%, and on Flappy Bird policies work only near their training delay.

Finally, an explicit latency value in the prompt raises the return of a mixed-latency Flappy Bird policy from below 20 to 268–434, because the same frame can call for different actions under different delays. Visual history also helps, but the best representation depends on the task.

We contribute: (1) a 12-task, three-architecture evaluation that exposes the latency gap hidden by world-pausing evaluation; (2) a profiling and replay framework that converts measured traces into reproducible training and evaluation conditions; and (3) evidence for when residual-latency adaptation improves hardware-timed control, transfers across delays, and benefits from timing information.

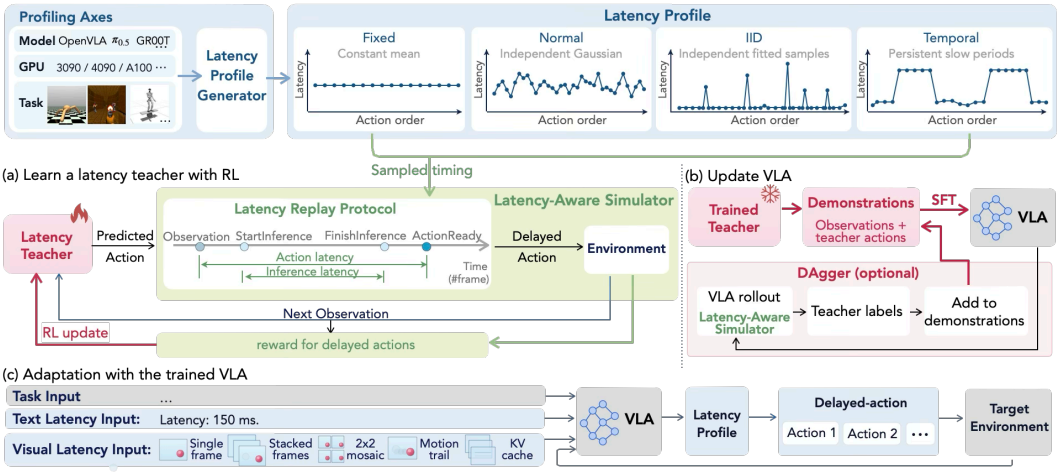

Figure 3: The framework measures action latency on a target setup, replays those delays during training, and applies the same timing during simulated evaluation. Top: measurements for one model, GPU, and task are fitted with a Fixed, Normal, IID, or Temporal profile. (a) For each model prediction, latency replay samples a delay, converts it to environment frames, and holds the predicted action until its scheduled time. RL trains a small teacher from the rewards produced under these delays. (b) The frozen teacher labels demonstrations for VLA supervised fine-tuning. When DAgger is used, the teacher also labels states visited by the VLA in the delayed simulator. (c) At evaluation, the VLA receives the task input and, depending on the experiment, an explicit latency value, recent visual history, or both; the simulator applies each predicted action after its sampled delay.

2 World-Pausing Evaluation Hides Latency Failures

2.1 Latency Conditions

Throughout the paper, t denotes time in milliseconds, k indexes environment frames, and i indexes inference requests. A frame is one benchmark environment step and lasts T_{frame} milliseconds. An inference request asks the policy for an action, or an *action chunk* of H future actions (Zhao et al., 2023), from one observation. Its *action latency* Δt_i runs from observation capture until the predicted action is ready, including preprocessing, communication, model inference, and action decoding; *inference time* covers only the model computation. In frames,

$$\Delta k_i = \left\lceil \frac{\Delta t_i}{T_{\text{frame}}} \right\rceil . \tag{1}$$

For example, at 20 frames per second, $T_{\text{frame}} = 50$ ms, and both 75 ms and 100 ms give $\Delta k_i = 2$. A fixed delay shared by all requests is written Δt or Δk. Appendix B lists all notation.

We evaluate policies in three settings, distinguished by how the environment advances while the policy computes. **World-pausing evaluation** (effective zero latency) freezes the environment until the action is ready, so $\Delta k_i = 0$; it is the reference for all other conditions. **Hardware-timed evaluation** lets the environment advance while the model runs on the target hardware. **Latency simulation** advances the environment according to delays supplied by a timing model, which supports controlled delay sweeps and reproducible replay without waiting for model inference. The model is either a fixed delay Δk for every request or a *latency profile* fitted to measured requests on a target deployment (§3). We name each policy by its training condition: a *zero-latency* policy is trained with world-pausing, a *fixed-delay* policy with one Δk, a *mixed-latency* policy with a different fixed delay in each episode, and a *profile-trained* policy with a latency profile.

2.2 How Large Is the Gap of World-Pausing Evaluation?

Setup. We take zero-latency OpenVLA, $\pi_{0.5}$, and GR00T policies[1] and change only the latency at evaluation. Figure 2 sweeps fixed delays on Flappy Bird, Demon Attack, Deadly Corridor, and MIKASA Intercept Grab Fast. Table 1 evaluates 12 tasks in real time on an RTX 3090 GPU, where the environment keeps advancing while the policy runs: its Zero/Zero and Zero/Real rows give the training/evaluation conditions, where Real denotes this hardware-timed evaluation. An episode's *return* is its undiscounted total reward until termination or the evaluation limit, and $\bar{R}(\pi; L)$ is the

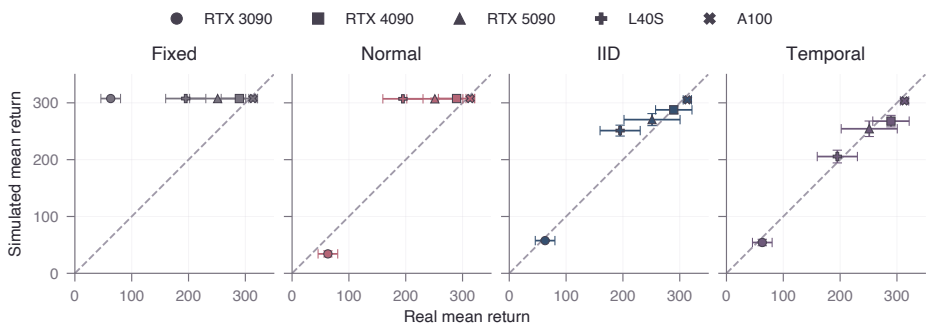

Figure 4: Across five GPU deployments, the Temporal profile gives the closest match between simulated and hardware-timed Flappy Bird return. Each point is one deployment, with hardware-timed mean return on the horizontal axis and simulated mean return on the vertical axis. Panels compare the Fixed, Normal, IID, and Temporal profile generators; the dashed line marks exact agreement. Error bars are 95% confidence intervals over five hardware-timed runs and five simulation seeds.

mean return of policy π under latency condition L, with $L = 0$ for zero latency. We report

$$\mathrm{RelativeReturn}(\pi; L \mid \pi_{\mathrm{ref}}) = 100\,\frac{\bar{R}(\pi; L)}{\bar{R}(\pi_{\mathrm{ref}}; 0)}, \tag{2}$$

where π_{ref} is the zero-latency policy for the same task and architecture. For a frozen policy, $\pi_{\mathrm{ref}} = \pi$. When we compare policies before and after latency training (Figure 1), both share the original policy's zero-latency return as the denominator, so a trained policy can exceed 100%; because the two are separately trained checkpoints, such values can also reflect differences in training data and convergence, not only latency. The MIKASA panel of Figure 2 reports success rate over 200 episodes. Environment details are in Appendix C.

A delay of one or two frames is enough to fail. With a two-frame delay, the six policies on Flappy Bird and Demon Attack keep only 1.2–7.1% of their zero-latency return. On Flappy Bird, even one frame (about 33 ms) leaves 4.2–5.3%. On MIKASA, single-action policies fall from 50.0–63.5% success at zero delay to zero at two frames. Policies that predict eight-action chunks degrade more slowly but still fall to 6.0–7.5% at four frames.

Real GPU latency causes the same failure. Evaluated in real time on the RTX 3090, the median of the 36 zero-latency policies keeps 6.6% of its return, and 26 of them keep less than 20%. The four Gymnasium control tasks are affected most, with a median of 2.1%. Sensitivity depends on the task: on AirRaid, policies keep 35–71% of their return, but on Asterix only 2.5–6.5%. Because the policies were trained with different budgets, Figure 2 measures sensitivity to delay, not a ranking of architectures. A policy should therefore be trained and tested under the latency it will face. To do this without repeatedly using the target deployment, we must first replay its latency in simulation.

3 Profiling and Replaying Deployment Latency

Deployment latency is not fixed, and running every training episode on target hardware would be expensive. We therefore fit a latency profile to measured action latencies and replay it during new interactions (Figure 3). The profile models both the delay distribution and whether large delays occur consecutively.

3.1 Actions Under Delay

The simulator separates predicting an action from applying it. Request i observes frame k_i^{obs} and predicts an action a'_i, or a chunk $(a'_{i,1}, \ldots, a'_{i,H})$; the prime marks a prediction, and a_k is the action applied at frame k. The prediction becomes available at

$$k_i^{\mathrm{ready}} = k_i^{\mathrm{obs}} + \Delta k_i. \tag{3}$$

For example, at 20 frames per second, a request that observes frame 10 and takes 150 ms has $\Delta k_i = 3$ and is ready at frame 13. Meanwhile the environment keeps advancing under a fixed

[1]We use the StarVLA (StarVLA Community, 2026) implementations of all three architectures; the exact checkpoints are listed in Appendix C.

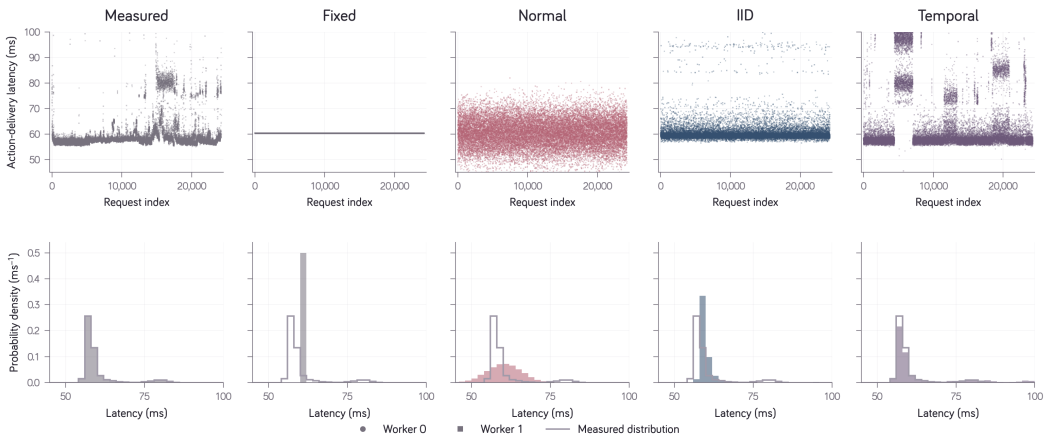

Figure 5: Only the Temporal generator preserves long sequences of consecutive slow requests visible in the measured trace. The top row shows measured and generated action latencies in request order. Marker shape identifies which of the two parallel model processes handled each request. The bottom row shows the corresponding latency distributions from 45 to 100 ms; the measured distribution is outlined in gray.

execution rule: it continues an existing chunk, repeats the previous action, or applies a no-op. In chunked configurations, the next request also receives the *action prefix*, the actions already committed to execution, so the policy knows what will happen while it computes. In simulation, delays are assigned, so the simulator's own computation does not advance the environment clock. Appendix B.2 gives the ordering and capacity rules.

3.2 Latency Profiles and Replay

To build a profile, we run the policy on the target deployment and record the action latency Δt_i of every request, in order, along with its episode and measurement run. Keeping the order preserves sequences of consecutive slow requests. A *latency profile* L is a model fitted to these measurements that generates an ordered sequence of action latencies Δt_i: it models how frequently different delays occur and whether several consecutive requests have large delays. It describes a complete setup, including the model, the software, and the machine, and it also models inference capacity (Appendix D). A fixed delay is a constant profile. *Latency replay* samples a new delay sequence from L during fresh task interactions, rather than replaying recorded actions or trajectories. It converts each delay to frames with Eq. 1 and applies the rules above, while an existing task simulator supplies dynamics and rewards. We use the same replay for training and for simulated evaluation. Applying a profile to another task transfers only the deployment timing, not that task's measured latency.

3.3 Four Profile Models

Let $\Delta t_1, \ldots, \Delta t_N$ be the ordered measurements from the target deployment, with mean $\hat{\mu}$, standard deviation $\hat{\sigma}$, and empirical distribution $\hat{F}$. We compare four generators that keep increasingly more of the measured pattern:

$$\textbf{Fixed:}\quad \Delta t_i = \hat{\mu}, \qquad \textbf{Normal:}\quad \Delta t_i = \max(0, x_i),\ \ x_i \overset{\text{iid}}{\sim} \mathcal{N}(\hat{\mu}, \hat{\sigma}^2),$$
$$\textbf{IID:}\quad \Delta t_i \overset{\text{iid}}{\sim} \hat{F}, \qquad \textbf{Temporal:}\quad \Delta t_i \mid s_i \sim \begin{cases} \hat{F}^{\text{base}}, & s_i = \text{normal}, \\ (1-q)\,\hat{F}^{\text{base}} + q\,\hat{F}^{\text{slow}}, & s_i = \text{slow}. \end{cases} \tag{4}$$

Fixed preserves only the measured mean, and Normal the mean and variance. IID preserves how often each delay occurs, including rare large delays, but samples each request independently. Temporal additionally models consecutive periods of large latency through a hidden state $s_i \in \{\text{normal}, \text{slow}\}$. For each inference process, we use the smallest median busy time across episodes as a baseline. A request is *slow* when its process remains busy for more than 1.25 times this baseline. Slow requests that are close in the trace are merged into *slow periods*. From a normal state, each request starts a slow period with probability p; the period lasts D requests, with D drawn

Table 1: Training with the measured RTX 3090 latency profile improves the real-time score in all 36 task–architecture pairs. Zero/Zero evaluates the original zero-latency policy without delay. Zero/Real evaluates that same policy while the environment advances during RTX 3090 inference. Target/Real first trains with the architecture-specific RTX 3090 profile in simulation, then evaluates on the RTX 3090 in real time. Cells report mean episode return on each task's native scale, except Intercept Grab Fast, which reports success rate in percent. Shading marks Target/Real; ↑ marks an improvement over Zero/Real.

			Games						**Embodied Control**					
Architecture	Train condition	Eval condition	Demon Attack	Asterix	Atlantis	AirRaid	Deadly Corridor	Flappy Bird	Ant	Hopper	Inverted Pendulum	Walker2D	Balance	Intercept Grab Fast
OpenVLA	Zero	Zero	2.36k	5.80k	39.56k	2.00k	2.10k	439.1	5.57k	1.42k	1.00k	983.9	275.6	36.1
	Zero	Real	584.4	374.5	18.59k	878.8	742.4	19.2	194.3	67.9	19.8	103.7	50.7	18.6
	Target	Real	1.45k↑	3.65k↑	39.23k↑	2.35k↑	2.09k↑	373.8↑	1.85k↑	424.6↑	586.9↑	857.4↑	114.6↑	25.8↑
$\pi_{0.5}$	Zero	Zero	2.35k	5.76k	39.62k	1.94k	2.10k	408.7	5.78k	3.48k	1.00k	3.63k	328.6	36.8
	Zero	Real	112.3	143.5	4.37k	1.39k	140.9	5.6	131.6	8.4	20.2	81.0	46.8	8.1
	Target	Real	907.3↑	1.02k↑	26.45k↑	3.23k↑	1.45k↑	344.5↑	1.77k↑	235.2↑	830.4↑	371.1↑	72.3↑	17.6↑
GR00T	Zero	Zero	2.37k	5.71k	39.66k	2.10k	2.10k	428.5	6.22k	872.8	1.00k	3.53k	267.5	33.5
	Zero	Real	231.9	182.0	8.17k	733.8	341.8	6.6	165.4	8.6	15.6	63.2	45.3	14.3
	Target	Real	1.20k↑	1.44k↑	28.49k↑	1.59k↑	2.09k↑	414.5↑	1.96k↑	242.6↑	1.00k↑	852.9↑	90.4↑	25.2↑

from the measured period lengths, and within it a fraction q of requests draws from the slow distribution $\hat{F}^{\text{slow}}$. All of p, D, q, $\hat{F}^{\text{base}}$, and $\hat{F}^{\text{slow}}$ are fitted to the trace; the per-worker version and the modeling of inference capacity are specified in Appendix D.

The difference between IID and Temporal lies in consecutive requests. Let π be the overall fraction of slow requests. Under IID, r consecutive requests are all slow with probability π^r. Under Temporal, inside a slow period this probability is q^r, and $q \geq \pi$ because slow periods contain every slow request. For illustration, with $\pi = 0.1$ and $q = 0.5$, a run of five slow requests is about 3,000 times more likely inside a slow period than under IID. Such a sequence delays many consecutive actions, so a policy that tolerates isolated slow requests can still fail when they occur consecutively.

3.4 Temporal Profiles Best Match Hardware-Timed Performance

In our five-deployment Flappy Bird study, replaying only the mean latency misses hardware-timed performance by a wide margin, and Temporal gives the smallest average and worst-case gap. We run one OpenVLA policy, trained with a fixed two-frame delay, on Flappy Bird on a set $\mathcal{M}$ of five hardware deployments: RTX 3090, RTX 4090, RTX 5090, L40S, and A100. Each deployment uses a frame rate that maps its measured latency near the policy's training delay, and the same rate in hardware-timed runs and simulation (Appendix E). For profile model m fitted to deployment d, we measure the calibration gap

$$\text{Gap}(m) = \frac{1}{|\mathcal{M}|} \sum_{d \in \mathcal{M}} \left| \bar{R}_{\text{hw}}(d) - \bar{R}^{m}_{\text{sim}}(d) \right|, \tag{5}$$

where $\bar{R}_{\text{hw}}(d)$ averages five hardware-timed runs on d and $\bar{R}^{m}_{\text{sim}}(d)$ averages five simulation seeds.

The mean gap is 87.4 for Fixed, 44.1 for Normal, 18.1 for IID, and 10.8 for Temporal (Figure 4); the largest per-deployment gaps are 244.5, 112.0, 56.1, and 21.7. Fixed maps every request to the same frame count and predicts 307.6 on every deployment, whereas hardware-timed returns range from 63.0 (RTX 3090) to 313.4 (A100). IID is closer than Temporal on three deployments, but Temporal is the only model within 22 of the hardware-timed return on all five. The measured trace contains consecutive high-latency requests (Figure 5). IID disperses them, whereas Temporal preserves their order and the resulting sequence of delayed actions.

4 Latency Adaptation and Generalization

We next test whether profile training improves hardware-timed performance (§4.2) and whether a trained policy transfers across delays (§4.3).

4.1 Training Recipe

We keep the training recipe standard, so that the latency condition is the variable of interest. Training a VLA directly with RL is expensive, so we distill a small teacher. RL first trains the teacher to

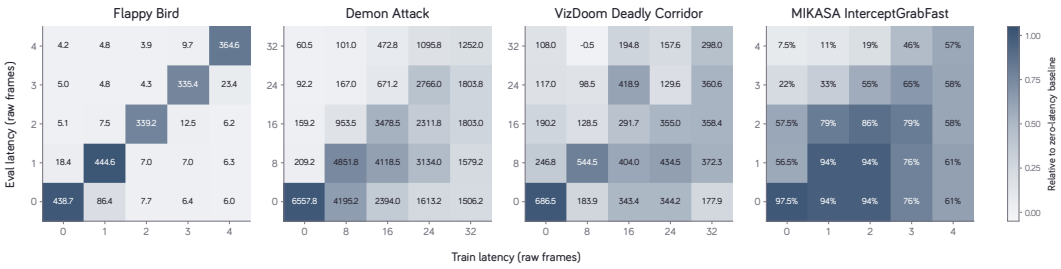

Figure 6: Transfer across fixed delays depends on both the task and the direction of transfer. Demon Attack, Deadly Corridor, and MIKASA transfer more reliably from longer training delays to shorter evaluation delays; Flappy Bird performs mainly near matching delays. Columns are training delays and rows are evaluation delays, both in raw frames. Cells report mean return for the games and success rate for MIKASA; color shows each value relative to a policy trained and evaluated at zero delay on the same task. Game cells use one training run and 20 evaluation episodes; Appendix M.3 gives the MIKASA settings.

maximize task reward in the delayed simulator of §3.1 under the chosen latency condition: a fixed delay, one fixed delay per episode, or a profile. The latency model determines when each predicted action becomes available; it does not provide the teacher's training examples, which come from new simulator rollouts. The frozen teacher then supplies action labels, and the VLA learns them by supervised fine-tuning (SFT). Chunked configurations additionally use DAgger (Ross et al., 2011): the VLA acts in the same delayed simulator, and the teacher labels the states it visits. All VLAs use the StarVLA (StarVLA Community, 2026) implementations of OpenVLA-OFT, $\pi_{0.5}$, and GR00T. During evaluation, all weights are fixed. Appendices B.1 and C give the objective, environments, and hyperparameters; Appendix J reports teacher training and evaluation. We use the measured profile because it represents the deployment condition. A separate small-teacher sweep finds no training distribution that dominates across settings (Appendix F).

4.2 Profile Training Improves Hardware-Timed Performance

Training in simulation with the RTX 3090 profile improves hardware-timed return on the real GPU in all 36 task–architecture pairs, although the amount of recovery varies substantially across tasks. Table 1 reports three conditions per task and architecture. Zero/Zero is a zero-latency policy evaluated without latency; Zero/Real is the same policy evaluated in real time on the RTX 3090; Target/Real is a policy trained in simulation with the architecture-specific RTX 3090 profile and then evaluated in real time on the RTX 3090. Training never uses the GPU's real-time loop; only the evaluation does. Figure 1 shows Zero/Real and Target/Real as percentages of Zero/Zero.

Environment rate. A latency in milliseconds becomes a frame delay through the environment rate. Six tasks in Table 1 use one benchmark rate across architectures, and profile training improves all 18 of their task–architecture pairs. For Flappy Bird, AirRaid, and the four MuJoCo tasks, whose native rates exceed VLA inference speed, we set each architecture's control rate so that its mean RTX 3090 latency spans about one decision step. Appendix C details the rates, decision strides, and conversions (Table 4). Fixed-delay sweeps use each benchmark's standard rate. The game policies in Table 1 additionally use an eight-step decision history, so their zero-latency references differ from Figure 2.

On Flappy Bird, zero-latency policies keep only 1–4% of their return, while profile-trained policies reach 84–97%. The gain, averaged over the three architectures, ranges from 16% of the Zero/Zero return on Balance to 86% on Flappy Bird. Tasks with simple dynamics recover most: on Inverted Pendulum, trained policies reach 59–100%. Locomotion recovers least: on Hopper, trained policies reach 7–30%. On AirRaid, two of the three trained policies even exceed their Zero/Zero reference.

4.3 Transfer Across Latency Is Asymmetric and Task-Dependent

In Demon Attack and Deadly Corridor, policies trained with longer delays transfer more reliably to shorter delays than the reverse, whereas Flappy Bird policies remain strongest near their training delay. We train one fixed-delay OpenVLA policy per delay on Flappy Bird, Demon Attack, and Deadly Corridor, and evaluate every policy at every delay (Figure 6). Relative to each task's zero-latency return, the cells where the training delay exceeds the evaluation delay (below the diagonal) average 37% on Demon Attack and 49% on Deadly Corridor, while the cells above the diagonal

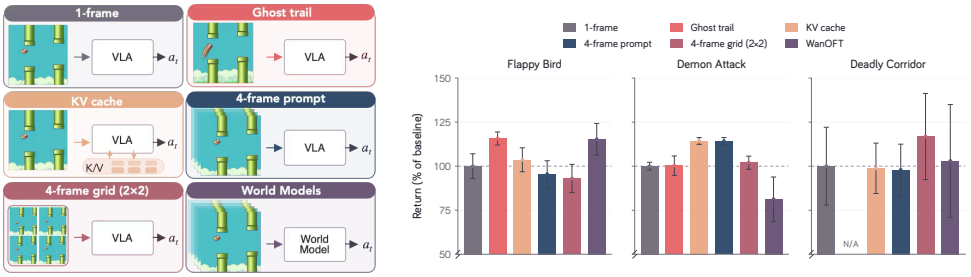

Figure 7: No visual-history input gives the largest improvement on all three tasks. Left: five ways to provide recent frames to OpenVLA, plus a separate world-model policy, WanOFT, based on the Wan2.2 video backbone (Team Wan et al., 2025). Right: each bar is the method's mean return divided by the single-frame mean at the same fixed delay; 100% means no change from the single-frame baseline. The delays are 3 frames for Flappy Bird and 6 frames for Demon Attack and Deadly Corridor. Error bars are 95% confidence intervals over evaluation episodes from one training run per condition. Ghost trail was not evaluated on Deadly Corridor.

Table 2: When both policies train on delays from 0 to 4 frames, stating the evaluation delay in the prompt raises Flappy Bird mean return from below 20 to 268–434. Columns are fixed delays in raw frames; cells report mean return ± the 95% confidence-interval half-width over 20 episodes. Each row uses one training run, with the same data and training budget.

	Evaluation delay (raw frames)				
Latency stated in prompt	0	1	2	3	4
Yes	433.8 ± 15.7	434.0 ± 22.2	341.3 ± 69.2	290.3 ± 81.2	267.6 ± 69.0
No	18.8 ± 7.3	14.3 ± 7.3	9.4 ± 3.1	6.5 ± 1.6	4.5 ± 0.4

average 6% and 24%. MIKASA, where chunked policies receive the committed action prefix, shows the same pattern (77% vs. 40% of zero-latency success), although its columns differ in checkpoint budget and evaluation episodes (Appendix M.3). Calling the policy trained at the test delay the same-delay reference, on Demon Attack a policy trained at 16 frames reaches 85% of this reference at 8 frames, whereas a policy trained at 8 frames reaches only 27% of it at 16 frames.

A possible explanation is that long-delay training repeatedly exposes the policy to observations whose actions take effect only after the scene has changed; strategies learned from this experience may remain useful at shorter delays, whereas short-delay training does not provide the reverse exposure. Training only at the mean latency omits the consecutive slow requests present in a measured profile. We train on the measured profile so that training includes these consecutive slow requests. Where no single delay transfers, as on Flappy Bird, the policy must be trained across delays and told which one it faces (§5.1).

5 What Information Supports Latency Adaptation?

We test two sources of timing information: an explicit latency value tells the policy how many frames its action will be delayed, while recent images show how the scene changed during that delay. A single frame provides neither.

5.1 Textual Latency Signals

In the Flappy Bird mixed-latency experiment, the explicit latency value separates success from failure. Under different delays, the same frame can call for different actions, because earlier actions will take effect at different times. Without the delay value, the same input is therefore paired with conflicting action labels. We call this ambiguity *latency aliasing*. We train two OpenVLA policies on Flappy Bird on the same mixed-latency data with delays of 0 to 4 frames and the same budget; only one prompt states the delay. Across test delays of 0 to 4 frames, the policy with the latency value reaches a mean return of 268–434, and the policy without it stays below 20 (Table 2). The prompt supplies the delay assigned to the evaluation condition; it does not predict the runtime of an individual future hardware request. In §4.2, each task-specific policy keeps the latency value used during its training in the prompt.

5.2 Visual History

Recent frames help, but the best format depends on the task. The latency value states how long the delay is, not what happened during it. We compare six inputs (Figure 7): the current frame; a *ghost trail* that draws recent object positions on one image; four recent frames shown separately or in a 2×2 grid; *KV memory*, which caches keys and values from up to three earlier frames; and *WanOFT*, a StarVLA implementation (StarVLA Community, 2026) that processes five ordered frames with a Wan2.2 backbone (Team Wan et al., 2025) and an OFT action head (Kim et al., 2025a). WanOFT serves as a different-architecture reference. We train and test with fixed 3-frame delays on Flappy Bird and 6-frame delays on Demon Attack and Deadly Corridor (Appendix Table 8).

No format is best everywhere. The ghost trail improves Flappy Bird return by 16% over the current frame but does not help on Demon Attack, where KV memory gives the largest gain (14%). On Deadly Corridor, the four-frame grid is best. More frames are not necessarily better: WanOFT only matches the ghost trail on Flappy Bird and does not improve over a single frame on Demon Attack.

Describing latency is not acting under it. In an exploratory study (Appendix I), GPT-6 Astra (OpenAI, 2026) reviews a frozen OpenVLA's action chunks under a known 100 ms delay on MIKASA Intercept Grab Fast. Its assessments track when each chunk will execute and sometimes predict a miss, yet we observe few successful recoveries. Describing the delay alone did not yield control under it.

6 Related Work

Measuring and replaying deployment conditions. Systems research replays recorded network traces to test web applications (Netravali et al., 2015) and designs services for slow requests, or tail latency (Dean & Barroso, 2013). For agents, PEERNet profiles sensing, networking, and inference in robots (Narayanan et al., 2024), while VLA-Perf compares inference cost across hardware (Jiang et al., 2026b). Real-time benchmarks map median inference time to a fixed delay (Thodoroff et al., 2022). ReflexBench advances manipulation during inference (Chen et al., 2026), while Bench2Drive-Robust evaluates delayed driving (Zhang et al., 2026). We fit a reusable profile, including consecutive slow requests, and replay it for training and evaluation.

Reducing and adapting to latency. Quantization (Lin et al., 2024; Xiao et al., 2023) and speculative decoding (Leviathan et al., 2023) shorten inference; VLA caching (Xu et al., 2025b) and action chunking (Chi et al., 2025) amortize it; asynchronous execution overlaps inference with action delivery or corrects stale chunks (Black et al., 2025c;d; Tang et al., 2025; Sendai et al., 2025). We study a complementary direction: estimating residual latency from measured runs and adapting the VLA under replayed timing.

Learning under delay. Delayed decision processes and real-time RL model stale observations and actions (Walsh et al., 2009; Travnik et al., 2018; Ramstedt & Pal, 2019; Ramstedt et al., 2021). ARLI (Zhu et al., 2026), TIC-VLA (Huang et al., 2026), SmoothRL (Gao et al., 2026), and training-time real-time chunking (Black et al., 2025d) train under fixed or hand-chosen random delays; teacher distillation (Xu et al., 2025a) and DAgger (Ross et al., 2011) are standard for VLA learning. We retain this recipe, derive delays from measured profiles, and study transfer across them.

Representing time. Temporal VLAs and memory modules encode recent frames, action consequences, or longer context (Jang et al., 2025; Lian et al., 2026; Ma et al., 2026; Shi et al., 2026; Li et al., 2026); Appendix A discusses them further.

7 Conclusion and Limitations

Faster inference reduces but does not eliminate action latency. World-pausing evaluation hides its cost: on a real GPU, most policies lose most of their return. Accurate evaluation requires the target deployment's latency profile because mean latency alone cannot reproduce its performance. Training with a measured profile improves hardware-timed performance, but transfer across delays remains asymmetric and task-dependent. Inference acceleration and policy adaptation are complementary: one reduces the delay, while the other prepares the agent to act under the remaining latency.

Our study is limited to controlled-clock digital environments, which permit reproducible timing but do not establish transfer to physical systems. Future work should reach native control rates, validate profiles across more tasks and deployment sessions, and combine latency-aware training with asynchronous execution methods.

AI USE STATEMENT

We used generative AI assistants (Claude and ChatGPT family models) to support several aspects of this work: (1) aiding and polishing the writing of specific manuscript passages; (2) retrieval and discovery of relevant literature and related work; and (3) research ideation and experiment execution, including assistance with boilerplate code, data processing, and evaluation workflows. All AI-assisted outputs were reviewed, verified, and revised by the human authors. The authors made all final decisions regarding the research design, experimental methodology, analysis, interpretation, and conclusions, and take full responsibility for the final content of this submission.

ETHICS STATEMENT

The experiments discussed here use simulated interactive environments. Performance under simulated latency does not establish safe operation of a physical robot. Deployment on physical systems would require task-specific safety measures and evaluation beyond the scope of this study.

REPRODUCIBILITY STATEMENT

The paper assets include local plotting inputs, figure-generation scripts, and experiment-specific descriptions of checkpoints, evaluation horizons, statistics, and source artifacts. Appendix D specifies the profiling methods and distinguishes current definitions from the frozen artifacts used in the calibration experiment. Appendix C lists environments, checkpoints, and evaluation settings.

REFERENCES

Ayoub Agouzoul. Understanding Asynchronous Inference Methods for Vision-Language-Action Models. *arXiv preprint arXiv:2605.08168*, 2026. URL https://arxiv.org/abs/2605.08168.

Ivan Anokhin, Rishav Rishav, Matt Riemer, Stephen Chung, Irina Rish, and Samira Ebrahimi Kahou. Handling Delay in Real-Time Reinforcement Learning. In *International Conference on Learning Representations*, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/770cabd044c4eacb6dc5924d9a686dce-Abstract-Conference.html.

Marc G. Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The Arcade Learning Environment: An Evaluation Platform for General Agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013. URL https://jair.org/index.php/jair/article/view/10819.

Kevin Black, Noah Brown, James Darpinian, Karan Dhabalia, Danny Driess, Adnan Esmail, Michael Robert Equi, Chelsea Finn, Niccolo Fusai, Manuel Y. Galliker, et al. $\pi_{0.5}$: a Vision-Language-Action Model with Open-World Generalization. In *Proceedings of The 9th Conference on Robot Learning*, volume 305 of *Proceedings of Machine Learning Research*, pp. 17–40. PMLR, 2025a. URL https://proceedings.mlr.press/v305/black25a.html.

Kevin Black, Noah Brown, Danny Driess, Adnan Esmail, Michael Robert Equi, Chelsea Finn, Niccolo Fusai, Lachy Groom, Karol Hausman, Brian Ichter, et al. π_0: A Vision-Language-Action Flow Model for General Robot Control. In *Proceedings of Robotics: Science and Systems*, 2025b. doi: 10.15607/RSS.2025.XXI.010. URL https://www.roboticsproceedings.org/rss21/p010.html.

Kevin Black, Manuel Galliker, and Sergey Levine. Real-Time Execution of Action Chunking Flow Policies. In *Advances in Neural Information Processing Systems*, volume 38, 2025c. doi: 10.52202/085713-1122. URL https://proceedings.neurips.cc/paper_files/paper/2025/hash/300ccb2187dedd4edcc07f7e76d8e553-Abstract-Conference.html.

Kevin Black, Allen Z. Ren, Michael Equi, and Sergey Levine. Training-Time Action Conditioning for Efficient Real-Time Chunking. *arXiv preprint arXiv:2512.05964*, 2025d. URL https://arxiv.org/abs/2512.05964.

Anthony Brohan, Noah Brown, Justice Carbajal, Yevgen Chebotar, Joseph Dabis, Chelsea Finn, Keerthana Gopalakrishnan, Karol Hausman, Alexander Herzog, Jasmine Hsu, et al. RT-1: Robotics Transformer for Real-World Control at Scale. In *Proceedings of Robotics: Science and Systems*, 2023. doi: 10.15607/RSS.2023.XIX.025. URL https://www.roboticsproceedings.org/rss19/p025.html.

Baiming Chen, Mengdi Xu, Liang Li, and Ding Zhao. Delay-aware model-based reinforcement learning for continuous control. *Neurocomputing*, 450:119–128, 2021. doi: 10.1016/j.neucom.2021.04.015. URL https://www.sciencedirect.com/science/article/pii/S0925231221005427.

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. URL https://arxiv.org/abs/2608.14379.

Egor Cherepanov, Nikita Kachaev, Alexey Kovalev, and Aleksandr Panov. Memory, Benchmark & Robots: A Benchmark for Solving Complex Tasks with Reinforcement Learning. In *The Fourteenth International Conference on Learning Representations*, 2026a. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/1088fffae5f2aa8c8324e4f45f62248d-Abstract-Conference.html.

Egor Cherepanov, Nikita Kachaev, Daniil Zelezetsky, Aydar Bulatov, Artem Pshenitsyn, Yuri Kuratov, Alexey Skrynnik, Aleksandr I. Panov, and Alexey K. Kovalev. μVLA: On Recurrent Memory for Partially Observable Manipulation in VLA Models. *arXiv preprint arXiv:2606.12497*, 2026b. URL https://arxiv.org/abs/2606.12497.

Cheng Chi, Zhenjia Xu, Siyuan Feng, Eric Cousineau, Yilun Du, Benjamin Burchfiel, Russ Tedrake, and Shuran Song. Diffusion policy: Visuomotor policy learning via action diffusion. *The International Journal of Robotics Research*, 44(10-11):1684–1704, 2025. doi: 10.1177/02783649241273668. URL https://doi.org/10.1177/02783649241273668.

Karl Cobbe, Chris Hesse, Jacob Hilton, and John Schulman. Leveraging Procedural Generation to Benchmark Reinforcement Learning. In *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pp. 2048–2056. PMLR, 2020. URL https://proceedings.mlr.press/v119/cobbe20a.html.

Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

Perry Dong, Kuo-Han Hung, Dorsa Sadigh, and Chelsea Finn. Reinforcement Learning for Real-Time Vision-Language-Action Policies. *arXiv preprint arXiv:2609.18207*, 2026. URL https://arxiv.org/abs/2609.18207.

Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers. In *International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=tcbBPnfwxS.

Guang Gao, Yuxuan Nong, Baifu Huang, and Jianan Wang. SmoothRL: Online Reinforcement Learning During Asynchronous Execution. *arXiv preprint arXiv:2608.29768*, 2026. URL https://arxiv.org/abs/2608.29768.

Dibya Ghosh, Homer Rich Walke, Karl Pertsch, Kevin Black, Oier Mees, Sudeep Dasari, Joey Hejna, Tobias Kreiman, Charles Xu, Jianlan Luo, et al. Octo: An Open-Source Generalist Robot Policy. In *Proceedings of Robotics: Science and Systems*, 2024. doi: 10.15607/RSS.2024.XX.090. URL https://www.roboticsproceedings.org/rss20/p090.html.

Alex Graves. Adaptive Computation Time for Recurrent Neural Networks, 2016. URL https://arxiv.org/abs/1603.08983.

Jiayuan Gu, Fanbo Xiang, Xuanlin Li, Zhan Ling, Xiqiang Liu, Tongzhou Mu, Yihe Tang, Stone Tao, Xinyue Wei, Yunchao Yao, et al. ManiSkill2: A Unified Benchmark for Generalizable Manipulation Skills. In *International Conference on Learning Representations*, 2023. URL https://maniskill2.github.io/.

Xiangming Gu, Tianyu Pang, Chao Du, Qian Liu, Fengzhuo Zhang, Cunxiao Du, Ye Wang, and Min Lin. When Attention Sink Emerges in Language Models: An Empirical View. In *International Conference on Learning Representations*, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/f1b04face60081b689ba740d39ea8f37-Abstract-Conference.html.

Chi Han, Qifan Wang, Hao Peng, Wenhan Xiong, Yu Chen, Heng Ji, and Sinong Wang. LM-Infinite: Zero-Shot Extreme Length Generalization for Large Language Models. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 3991–4008. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.naacl-long.222. URL https://aclanthology.org/2024.naacl-long.222/.

Zhiyu Huang, Yun Zhang, Johnson Liu, Rui Song, Chen Tang, and Jiaqi Ma. TIC-VLA: A Think-in-Control Vision-Language-Action Model for Robot Navigation in Dynamic Environments. *arXiv preprint arXiv:2602.02459*, 2026. URL https://arxiv.org/abs/2602.02459.

Huiwon Jang, Sihyun Yu, Heeseung Kwon, Hojin Jeon, Younggyo Seo, and Jinwoo Shin. ContextVLA: Vision-Language-Action Model with Amortized Multi-Frame Context. *arXiv preprint arXiv:2510.04246*, 2025. URL https://arxiv.org/abs/2510.04246.

Hai Jiang, Yixian Zou, Binbin Liang, Boqian Liu, Fanman Meng, and Shuaicheng Liu. FutureRTC: Real-Time Robot Execution with Anticipatory-Conditioned Action Chunking. *arXiv preprint arXiv:2607.24008*, 2026a. URL https://arxiv.org/abs/2607.24008.

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*, 2026b. URL https://arxiv.org/abs/2602.18397.

Konstantinos V. Katsikopoulos and Sascha E. Engelbrecht. Markov decision processes with delays and asynchronous cost collection. *IEEE Transactions on Automatic Control*, 48(4):568–574, 2003. doi: 10.1109/TAC.2003.809799. URL https://ieeexplore.ieee.org/document/1193736/.

Michał Kempka, Marek Wydmuch, Grzegorz Runc, Jakub Toczek, and Wojciech Jaśkowski. ViZDoom: A Doom-based AI research platform for visual reinforcement learning. In *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, pp. 1–8, 2016. doi: 10.1109/cig.2016.7860433. URL https://doi.org/10.1109/cig.2016.7860433.

Moo Jin Kim, Chelsea Finn, and Percy Liang. Fine-Tuning Vision-Language-Action Models: Optimizing Speed and Success. In *Proceedings of Robotics: Science and Systems*, 2025a. doi: 10.15607/RSS.2025.XXI.017. URL https://www.roboticsproceedings.org/rss21/p017.html.

Moo Jin Kim, Karl Pertsch, Siddharth Karamcheti, Ted Xiao, Ashwin Balakrishna, Suraj Nair, Rafael Rafailov, Ethan P Foster, Pannag R Sanketi, Quan Vuong, et al. OpenVLA: An Open-Source Vision-Language-Action Model. In *Proceedings of The 8th Conference on Robot Learning*, volume 270 of *Proceedings of Machine Learning Research*, pp. 2679–2713. PMLR, 2025b. URL https://proceedings.mlr.press/v270/kim25c.html.

Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast Inference from Transformers via Speculative Decoding. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pp. 19274–19286. PMLR, 2023. URL https://proceedings.mlr.press/v202/leviathan23a.html.

Hang Li, Fengyi Shen, Dong Chen, Liudi Yang, Xudong Wang, Jinkui Shi, Zhenshan Bing, Ziyuan Liu, and Alois Knoll. ReMem-VLA: Empowering Vision-Language-Action Model with Memory via Dual-Level Recurrent Queries. *arXiv preprint arXiv:2603.12942*, 2026. URL https://arxiv.org/abs/2603.12942.

Qing Lian, Kent Yu, and Lei Zhang. Reflective VLA: In-Context Action Consequences Make VLAs Generalize. *arXiv preprint arXiv:2606.25215*, 2026. URL https://arxiv.org/abs/2606.25215.

Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. AWQ: Activation-aware Weight Quantization for On-Device LLM Compression and Acceleration. In *Proceedings of Machine Learning and Systems*, volume 6, 2024. URL https://proceedings.mlsys.org/paper_files/paper/2024/hash/42a452cbafa9dd64e9ba4aa95cc1ef21-Abstract-Conference.html.

Siyuan Ma, Yutian Zhang, Boshi Zhang, Qinglian Wu, Jiaqi Zhai, Dong Wei, and Xiaojin Huang. ForeTime-VLA: Causal Future-Token Distillation from a World Action Model for Conveyor-Belt Manipulation. *arXiv preprint arXiv:2608.20735*, 2026. URL https://arxiv.org/abs/2608.20735.

Aditya Narayanan, Pranav Kasibhatla, Minkyu Choi, Po-Han Li, Ruihan Zhao, and Sandeep Chinchali. PEERNet: An End-to-End Profiling Tool for Real-Time Networked Robotic Systems. In *2024 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 10326–10333, 2024. doi: 10.1109/iros58592.2024.10801747. URL https://doi.org/10.1109/iros58592.2024.10801747.

Ravi Netravali, Anirudh Sivaraman, Somak Das, Ameesh Goyal, Keith Winstein, James Mickens, and Hari Balakrishnan. Mahimahi: Accurate record-and-replay for HTTP. In *USENIX Annual Technical Conference (USENIX ATC)*, pp. 417–429, 2015.

Khanh D. Nguyen, Hung T. Ho, Chinh T. Nguyen, Thanh Q. Duong, Linh D. Le, Duy M. H. Nguyen, Vien A. Ngo, and An T. Le. vla.cpp: A Unified Inference Runtime for Vision-Language-Action Models. *arXiv preprint arXiv:2606.08094*, 2026. URL https://arxiv.org/abs/2606.08094.

NVIDIA, Johan Bjorck, Fernando Castañeda, Nikita Cherniadev, Xingye Da, Runyu Ding, Linxi "Jim" Fan, Yu Fang, Dieter Fox, Fengyuan Hu, et al. GR00T N1: An Open Foundation Model for Generalist Humanoid Robots. *arXiv preprint arXiv:2503.14734*, 2025. URL https://arxiv.org/abs/2503.14734.

OpenAI. GPT-6 Astra. OpenAI API Documentation, 2026. URL https://developers.openai.com/api/docs/models/gpt-6-astra.

Sungjae Park and Shubham Tulsiani. $\pi\mathbf{R}^2$: Reactive Real-time Flow Policies. *arXiv preprint arXiv:2607.26055*, 2026. URL https://arxiv.org/abs/2607.26055.

Xue Bin Peng, Marcin Andrychowicz, Wojciech Zaremba, and Pieter Abbeel. Sim-to-Real Transfer of Robotic Control with Dynamics Randomization. In *IEEE International Conference on Robotics and Automation*, pp. 3803–3810, 2018. doi: 10.1109/ICRA.2018.8460528. URL https://doi.org/10.1109/ICRA.2018.8460528.

Simon Ramstedt and Chris Pal. Real-Time Reinforcement Learning. In *Advances in Neural Information Processing Systems*, volume 32, 2019. URL https://proceedings.neurips.cc/paper_files/paper/2019/hash/54e36c5ff5f6a1802925ca009f3ebb68-Abstract.html.

Simon Ramstedt, Yann Bouteiller, Giovanni Beltrame, Christopher Pal, and Jonathan Binas. Reinforcement Learning with Random Delays. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=QFYnKlBJYR.

Stephane Ross, Geoffrey Gordon, and Drew Bagnell. A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, volume 15 of *Proceedings of Machine Learning Research*, pp. 627–635. PMLR, 2011. URL https://proceedings.mlr.press/v15/ross11a.html.

Daniel Ruan, Salma Mozaffari, Sigrid Adriaenssens, and Arash Adel. A Latency-Aware Framework for Visuomotor Policy Learning on Industrial Robots. *arXiv preprint arXiv:2602.14255*, 2026. URL https://arxiv.org/abs/2602.14255.

Sandeep Singh Sandha, Luis Garcia, Bharathan Balaji, Fatima Anwar, and Mani Srivastava. Sim2Real Transfer for Deep Reinforcement Learning with Stochastic State Transition Delays. In *Proceedings of the 2020 Conference on Robot Learning*, volume 155 of *Proceedings of Machine Learning Research*, pp. 1066–1083. PMLR, 2021. URL https://proceedings.mlr.press/v155/sandha21a.html.

Manolis Savva, Abhishek Kadian, Oleksandr Maksymets, Yili Zhao, Erik Wijmans, Bhavana Jain, Julian Straub, Jia Liu, Vladlen Koltun, Jitendra Malik, et al. Habitat: A Platform for Embodied AI Research. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 9339–9347, 2019. URL https://openaccess.thecvf.com/content_ICCV_2019/html/Savva_Habitat_A_Platform_for_Embodied_AI_Research_ICCV_2019_paper.html.

Kohei Sendai, Maxime Alvarez, Tatsuya Matsushima, Yutaka Matsuo, and Yusuke Iwasawa. Leave No Observation Behind: Real-time Correction for VLA Action Chunks. *arXiv preprint arXiv:2509.23224*, 2025. URL https://arxiv.org/abs/2509.23224.

Carmelo Sferrazza, Dun-Ming Huang, Xingyu Lin, Youngwoon Lee, and Pieter Abbeel. Humanoid-Bench: Simulated Humanoid Benchmark for Whole-Body Locomotion and Manipulation. In *Robotics: Science and Systems (RSS)*, 2024. URL https://www.roboticsproceedings.org/rss20/p061.html.

Hao Shi, Bin Xie, Yingfei Liu, Lin Sun, Fengrong Liu, Tiancai Wang, Erjin Zhou, Haoqiang Fan, Xiangyu Zhang, and Gao Huang. MemoryVLA: Perceptual-Cognitive Memory in Vision-Language-Action Models for Robotic Manipulation. In *International Conference on Learning Representations*, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/1fa3d6ccbcd15f7285fee666b2bd57be-Abstract-Conference.html.

Mustafa Shukor, Dana Aubakirova, Francesco Capuano, Pepijn Kooijmans, Steven Palma, Adil Zouitine, Michel Aractingi, Caroline Pascal, Martino Russi, Andres Marafioti, et al. SmolVLA: A Vision-Language-Action Model for Affordable and Efficient Robotics. *arXiv preprint arXiv:2506.01844*, 2025. URL https://arxiv.org/abs/2506.01844.

StarVLA Community. StarVLA: A Lego-like Codebase for Vision-Language-Action Model Developing. *arXiv preprint arXiv:2604.05014*, 2026. URL https://arxiv.org/abs/2604.05014.

Jiayi Su, Yixin Zheng, Mi Yan, Li Yi, Zhizheng Zhang, and He Wang. GPT 6 Astra as an Embodied Policy. Technical report and code, 2026. URL https://github.com/anonymous-report-421/GPT-as-Policy.

Amir Taherin, Juyi Lin, Arash Akbari, Arman Akbari, Pu Zhao, Weiwei Chen, David Kaeli, and Yanzhi Wang. Cross-Platform Scaling of Vision-Language-Action Models from Edge to Cloud GPUs. In *Proceedings of the Great Lakes Symposium on VLSI 2026*, pp. 234–239, 2026. doi: 10.1145/3787109.3816400. URL https://doi.org/10.1145/3787109.3816400.

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. URL https://arxiv.org/abs/2512.01031.

Yuval Tassa, Yotam Doron, Alistair Muldal, Tom Erez, Yazhe Li, Diego de Las Casas, David Budden, Abbas Abdolmaleki, Josh Merel, Andrew Lefrancq, et al. DeepMind Control Suite. *arXiv preprint arXiv:1801.00690*, 2018. URL https://arxiv.org/abs/1801.00690.

Team Wan, Ang Wang, Baole Ai, Bin Wen, Chaojie Mao, Chen-Wei Xie, Di Chen, Feiwu Yu, Haiming Zhao, Jianxiao Yang, et al. Wan: Open and Advanced Large-Scale Video Generative Models. *arXiv preprint arXiv:2503.20314*, 2025. URL https://arxiv.org/abs/2503.20314.

Surat Teerapittayanon, Bradley McDanel, and H. T. Kung. BranchyNet: Fast inference via early exiting from deep neural networks. In *2016 23rd International Conference on Pattern Recognition (ICPR)*, pp. 2464–2469. IEEE, 2016. doi: 10.1109/icpr.2016.7900006. URL https://doi.org/10.1109/icpr.2016.7900006.

Pierre Thodoroff, Wenyu Li, and Neil D. Lawrence. Benchmarking Real-Time Reinforcement Learning. In *NeurIPS 2021 Workshop on Pre-registration in Machine Learning*, volume 181 of *Proceedings of Machine Learning Research*, pp. 26–41. PMLR, 2022. URL https://proceedings.mlr.press/v181/thodoroff22a.html.

Josh Tobin, Rachel Fong, Alex Ray, Jonas Schneider, Wojciech Zaremba, and Pieter Abbeel. Domain randomization for transferring deep neural networks from simulation to the real world. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 23–30, 2017. doi: 10.1109/iros.2017.8202133. URL https://doi.org/10.1109/iros.2017.8202133.

Emanuel Todorov, Tom Erez, and Yuval Tassa. MuJoCo: A physics engine for model-based control. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 5026–5033, 2012. doi: 10.1109/IROS.2012.6386109. URL https://doi.org/10.1109/IROS.2012.6386109.

Mark Towers, Ariel Kwiatkowski, Jordan Terry, John U. Balis, Gianluca De Cola, Tristan Deleu, Manuel Goulão, Andreas Kallinteris, Markus Krimmel, Arjun KG, et al. Gymnasium: A Standard Interface for Reinforcement Learning Environments. *arXiv preprint arXiv:2407.17032*, 2024. URL https://arxiv.org/abs/2407.17032.

Jaden B. Travnik, Kory W. Mathewson, Richard S. Sutton, and Patrick M. Pilarski. Reactive Reinforcement Learning in Asynchronous Environments. *Frontiers in Robotics and AI*, 5:79, 2018. doi: 10.3389/frobt.2018.00079. URL https://www.frontiersin.org/journals/robotics-and-ai/articles/10.3389/frobt.2018.00079/full.

Thomas J. Walsh, Ali Nouri, Lihong Li, and Michael L. Littman. Learning and planning in environments with delayed feedback. *Autonomous Agents and Multi-Agent Systems*, 18(1):83–105, 2009. doi: 10.1007/s10458-008-9056-7. URL https://thomasjwalsh.net/pub/jaamas09Delayed.pdf.

Xin Wang, Fisher Yu, Zi-Yi Dou, Trevor Darrell, and Joseph E. Gonzalez. SkipNet: Learning Dynamic Routing in Convolutional Networks. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pp. 409–424, 2018. URL https://openaccess.thecvf.com/content_ECCV_2018/html/Xin_Wang_SkipNet_Learning_Dynamic_ECCV_2018_paper.html.

Ziye Wang, Modi Shi, Chaojun Ni, Jiazhi Yang, Mengdi Li, Zhizhong Su, Tianwei Lin, and Hongyang Li. NativeMEM: Native Memory Compression for Long-Horizon Robotic Manipulation. *arXiv preprint arXiv:2607.06678*, 2026. URL https://arxiv.org/abs/2607.06678.

Bo Xia, Haoyuan Sun, Bo Yuan, Zhiheng Li, Bin Liang, and Xueqian Wang. A delay-robust method for enhanced real-time reinforcement learning. *Neural Networks*, 181:106769, 2025. doi: 10.1016/j.neunet.2024.106769. URL https://www.global-optimization.com/boyuan/Mypaper/NN-2025.pdf.

Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. SmoothQuant: Accurate and Efficient Post-Training Quantization for Large Language Models. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pp. 38087–38099. PMLR, 2023. URL https://proceedings.mlr.press/v202/xiao23c.html.

Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. Efficient Streaming Language Models with Attention Sinks. In *International Conference on Learning Representations*, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/5e5fd18f863cbe6d8ae392a93fd271c9-Abstract-Conference.html.

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. URL https://arxiv.org/abs/2601.22153.

Charles Xu, Qiyang Li, Jianlan Luo, and Sergey Levine. RLDG: Robotic Generalist Policy Distillation via Reinforcement Learning. In *Proceedings of Robotics: Science and Systems*, 2025a. doi: 10.15607/RSS.2025.XXI.028. URL https://www.roboticsproceedings.org/rss21/p028.html.

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, 2025b. doi: 10.52202/085713-5484. URL https://proceedings.neurips.cc/paper_files/paper/2025/hash/f062da1973ac9ac61fc6d44dd7fa309f-Abstract-Conference.html.

Xinyu Xu, Yizheng Zhang, Yong-Lu Li, Lei Han, and Cewu Lu. HumanVLA: Towards Vision-Language Directed Object Rearrangement by Physical Humanoid. In *Advances in Neural Information Processing Systems*, volume 37, 2024. doi: 10.52202/079017-0591. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/215aeb07b5996c969c0123c3c6ee8f54-Abstract-Conference.html.

Zhiyuan Zhang, Zhenghao Jin, Yanlun Peng, Xianda Guo, Haoran Liu, Shaofeng Zhang, Xingjun Ma, Zuxuan Wu, Junchi Yan, Xiaosong Jia, et al. Bench2Drive-Robust: Benchmarking Closed-Loop Autonomous Driving under Deployment Perturbations. *arXiv preprint arXiv:2605.18059*, 2026. URL https://arxiv.org/abs/2605.18059.

Tony Z. Zhao, Vikash Kumar, Sergey Levine, and Chelsea Finn. Learning Fine-Grained Bimanual Manipulation with Low-Cost Hardware. In *Proceedings of Robotics: Science and Systems*, 2023. doi: 10.15607/RSS.2023.XIX.016. URL https://www.roboticsproceedings.org/rss19/p016.html.

Zhide Zhong, Haodong Yan, Junfeng Li, Junjie He, Tianran Zhang, and Haoang Li. VLA-OPD: Bridging Offline SFT and Online RL for Vision-Language-Action Models via On-Policy Distillation. *arXiv preprint arXiv:2603.26666*, 2026. URL https://arxiv.org/abs/2603.26666.

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. URL https://arxiv.org/abs/2608.23831.

Brianna Zitkovich, Tianhe Yu, Sichun Xu, Peng Xu, Ted Xiao, Fei Xia, Jialin Wu, Paul Wohlhart, Stefan Welker, Ayzaan Wahid, et al. RT-2: Vision-Language-Action Models Transfer Web Knowledge to Robotic Control. In *Proceedings of the 7th Conference on Robot Learning*, volume 229 of *Proceedings of Machine Learning Research*, pp. 2165–2183. PMLR, 2023. URL https://proceedings.mlr.press/v229/zitkovich23a.html.

A Extended Related Work

A.1 Reducing and Compensating for Inference Latency

Model lineages and faster inference. RT-1 (Brohan et al., 2023) and RT-2 (Zitkovich et al., 2023) established large-scale language-conditioned robot policies, while Octo (Ghosh et al., 2024), π_0 (Black et al., 2025b), $\pi_{0.5}$ (Black et al., 2025a), and GR00T N1 (NVIDIA et al., 2025) broadened the range of generalist action models. The cited papers identify model families; the OFT, PI, and GR00T rows in our experiments use StarVLA (StarVLA Community, 2026) with Qwen backbones and the checkpoints specified in Appendix C, rather than the original papers' weights. General inference methods reduce model-side compute through weight or activation quantization (Frantar et al., 2023; Lin et al., 2024; Xiao et al., 2023) and speculative decoding for autoregressive generation (Leviathan et al., 2023). Adaptive computation varies how much work an input receives through recurrent halting (Graves, 2016), early exits (Teerapittayanon et al., 2016), or dynamic routing (Wang et al., 2018). Quantization, speculative decoding, and adaptive computation can change inference time, but do not specify what a policy should do when the environment advances during that time.

OpenVLA-OFT speeds action generation with parallel decoding and action chunking (Kim et al., 2025a); SmolVLA combines a smaller backbone with asynchronous deployment (Shukor et al., 2025); and VLA-Cache reuses visual-token representations (Xu et al., 2025b). ACT (Zhao et al., 2023) and Diffusion Policy (Chi et al., 2025) popularized predicting action sequences instead of a single action per inference call. Chunking amortizes inference over several control steps, but the chunk must still be reconciled with observations and actions that change before the next result arrives.

Execution while a policy computes. Real-Time Chunking (RTC) inpaints actions already committed to execution when generating the next chunk (Black et al., 2025c); training-time action conditioning learns this prefix continuation and avoids the inference-time inpainting step (Black et al., 2025d). FutureRTC predicts the context at action execution (Jiang et al., 2026a), whereas VLASH rolls the robot state forward from committed actions to align new chunks with their future execution interval (Tang et al., 2025). DynamicVLA continuously streams recent predictions (Xie et al., 2026), and a comparison of asynchronous VLA methods evaluates how such choices respond to increasing fixed control-step delays (Agouzoul, 2026). Timestamp-based scheduling can reject actions that are no longer feasible without changing policy weights (Ruan et al., 2026). A2C2 instead adds a fast correction head using a fresh observation (Sendai et al., 2025); Real-Time EXPO-FT trains an edit policy for asynchronously generated chunks (Dong et al., 2026); and $\pi\mathrm{R}^2$ separates fresh proprioception from slower vision-language features (Park & Tulsiani, 2026). Our target-profile training fine-tunes VLA weights on trajectories collected under replayed device delays; the resulting policy needs no separate correction module at inference time.

A.2 Training and Evaluating under Latency

Delayed decisions and control. Delayed Markov decision processes (Katsikopoulos & Engelbrecht, 2003) and delayed-feedback planning (Walsh et al., 2009) make the age of observations and actions part of the decision problem. Reactive RL (Travnik et al., 2018) and real-time RL (Ramstedt & Pal, 2019) explicitly account for computation while the environment evolves. DCAC handles random action and observation delays through off-policy value estimation (Ramstedt et al., 2021); other real-time RL work uses augmented histories and temporal skip connections (Anokhin et al., 2025). Model-based controllers (Chen et al., 2021) and delay-robust controllers (Xia et al., 2025) likewise account for delayed transitions or actions. ARLI fine-tunes a generalist flow policy through a lightweight steering policy that sees committed actions and an intermediate observation (Zhu et al., 2026). TIC-VLA trains a navigation controller with delayed VLM features, explicit reasoning latency, and ego-motion (Huang et al., 2026). SmoothRL models asynchronous action-chunk execution during online RL and propagates value gradients only through actions actually executed. Its reported implementation uses a fixed latency budget, freezes the base policy, and trains an attached network in action space (Gao et al., 2026). Our study extends learning under delay to a fitted deployment profile: we use the profile to collect delayed-control demonstrations and fine-tune VLA weights, and separately evaluate transfer between fixed delays.

World-pausing and real-time benchmarks. Atari (Bellemare et al., 2013), VizDoom (Kempka et al., 2016), Procgen (Cobbe et al., 2020), continuous-control (Tassa et al., 2018), navigation (Savva

et al., 2019), and manipulation (Gu et al., 2023) simulators support the usual observation–action–step evaluation loop. In that loop, the simulator waits for the policy's action before advancing, so the policy's inference time does not enter the environment clock. Real-time protocols expose the difference: Thodoroff et al. (2022) turn measured inference time into action delay; ReflexBench advances dynamic manipulation during inference (Chen et al., 2026); and Bench2Drive-Robust evaluates delayed closed-loop driving (Zhang et al., 2026). We measure request-to-action timing on a target deployment configuration, then replay its variable delays in simulation while training and evaluating VLAs on other hardware.

Profiling device latency. PEERNet profiles sensor, network, and inference paths in networked robots (Narayanan et al., 2024); VLA-Perf models inference cost across architectures and hardware (Jiang et al., 2026b). Cross-platform measurements compare VLA latency, throughput, and task success on edge and data-center GPUs (Taherin et al., 2026). The vla.cpp runtime also reports heterogeneous-hardware latency, real-robot outcomes, and fixed delay injection (Nguyen et al., 2026). PEERNet, VLA-Perf, cross-platform measurements, and vla.cpp show why deployment timing must be measured. Our replay additionally preserves variation across requests, including consecutive slow requests and the time for which each inference process remains busy, so the same target profile can be evaluated on other training hardware.

Teacher transfer and simulation variation. RLDG uses specialist RL rollouts to fine-tune generalist policies (Xu et al., 2025a), and HumanVLA transfers a privileged-state RL teacher to a vision-language humanoid policy (Xu et al., 2024). DAgger labels learner-visited states to reduce distribution shift (Ross et al., 2011); VLA-OPD similarly supervises student rollouts with teacher action distributions (Zhong et al., 2026). For delayed students, the teacher's action must be learnable from information available when the student acts; a fresh execution-time state cannot simply be treated as a stale-observation label. Domain randomization (Tobin et al., 2017) and dynamics randomization (Peng et al., 2018) expose simulated policies to variation before deployment. Stochastic transition-delay sim2real work extends this idea to timing (Sandha et al., 2021). Our profile replay can be viewed as temporal variation grounded in measured deployment requests rather than a hand-chosen delay range.

A.3 Representing Time under Latency

Temporal VLA policies and robot memory. Temporal VLAs encode changing scenes in different ways: ContextVLA compresses multi-frame context (Jang et al., 2025), Reflective VLA incorporates observed action consequences (Lian et al., 2026), and ForeTime-VLA distills predictive features from a future-aware teacher into a causal history model (Ma et al., 2026). MemoryVLA maintains perceptual and cognitive history (Shi et al., 2026), ReMem-VLA uses recurrent queries at two timescales (Li et al., 2026), μVLA carries recurrent state across steps (Cherepanov et al., 2026b), and NativeMEM compresses historical visual information into action-supervised tokens (Wang et al., 2026). The reported gains of temporal VLA methods do not by themselves establish robustness to a measured inference-to-action delay.

Streaming caches and latency cues. StreamingLLM retains initial attention-sink keys and values alongside a recent window (Xiao et al., 2024); LM-Infinite combines prefix retention with bounded attention (Han et al., 2024), while later analysis investigates why attention sinks arise in language models (Gu et al., 2025). Streaming caches provide useful precedents for managing token history, though a language-model KV cache is not automatically a robot state estimator. In our setting, a textual latency signal states the operating condition, whereas recent images provide evidence of scene dynamics. We evaluate these two inputs separately under delayed action delivery and compare visual-history interfaces on the same control tasks.

B Notation and Formal Definitions

We use t for time in milliseconds, k for an environment frame index, and i for an inference request that is accepted for processing. A prefix Δ marks a duration or a number of delayed frames, so Δt_i and Δk_i are not timestamps or frame indices. The same conventions apply to measurement and simulation. Event superscripts name the observation (obs), worker start (start), action availability (ready), or first application (exec). The prime in a'_i marks a predicted action; a_k is the action applied to the environment. For a chunk, $a'_{i,h}$ is the h-th predicted action, with $h = 1, \ldots, H$; this position is not an environment frame index. When trainable parameters matter, we write the policy π as π_θ.

Symbol	Meaning	Unit
$t,\ k,\ i$	Continuous time; environment frame index; inference request index	ms; frames; requests
T_{frame}	Duration of one benchmark environment frame	ms/frame
f	Environment frame rate; $T_{\text{frame}} = 1000/f$	frames/s
$t_i^{\text{obs}}, t_i^{\text{start}}, t_i^{\text{ready}}$	Times when request i is observed, starts inference, and returns its action	ms
Δt_i	Observation-to-result action latency of request i	ms
Δt_i^{busy}	Time the assigned model process spends serving request i	ms
$\Delta k_i, \Delta k_i^{\text{busy}}$	Frame counts corresponding to action latency and process busy time	frames
$k_i^{\text{obs}}, k_i^{\text{ready}}, k_i^{\text{exec}}$	Observation, availability, and first-application frame indices	frame index
$a'_i,\ a_k$	Predicted action for request i; action applied at frame k	task-specific
$a'_{i,h}$	Action at position h in the chunk predicted by request i	task-specific
o_k	Policy input associated with frame k	task-specific
$\pi,\ \pi_\theta$	Policy; same policy with explicit parameters θ	
π_{ref}	Reference policy used for score normalization	
H	Number of actions predicted in one action chunk	actions
C	Number of observation frames retained as visual history	frames
K	Number of already committed actions supplied as an action prefix	actions
K_e	Number of environment steps in episode e	steps
$r_{e,k}, R, R_e, \bar{R}$	Step reward; random episode return; observed episode return; mean return	reward units
N_{ep}	Number of evaluation episodes	episodes
L	Latency condition, such as a fixed delay or fitted profile	
$\mathcal{P}_{\text{train}}$	Distribution of training latency conditions	
d, j, e, s	Deployment, inference process, episode, and measurement run identifiers	
N_d	Number of valid profiling records for deployment d	requests
$\widehat{\mu}_d, \widehat{\sigma}_d$	Measured mean and standard deviation of action latency	ms

Table 3: Symbols used for action timing, action chunks, returns, and latency profiles. Time variables use milliseconds, while frame delays count benchmark environment frames. Symbols for process busy time are needed only for the concurrent-inference profiling protocol in Appendix D. When deployments are compared, f_d and $T_{\text{frame},d}$ denote the frame rate and frame duration of deployment d; the subscript is omitted within one setup.

B.1 Evaluation Metrics and Training Objective

All returns below refer to one task. Let L denote the latency condition: a fixed delay or a fitted profile, with $L = 0$ denoting world-pausing. As defined in §3.2, a profile L supplies action latencies across requests; a fixed delay is a deterministic special case. The complete concurrent-inference profile also supplies process busy times as specified in Appendix D. It names the complete timing condition; individual request delays remain Δt_i and Δk_i. Let $R(\pi; L)$ be the total reward of one random episode under policy π and condition L. For observed episode e, let K_e be its number of environment steps and $r_{e,k}$ its reward at step k. The observed episode return and its mean over N_{ep} evaluation episodes are

$$R_e(\pi; L) = \sum_{k=0}^{K_e-1} r_{e,k}, \qquad \bar{R}(\pi; L) = \frac{1}{N_{\text{ep}}} \sum_{e=1}^{N_{\text{ep}}} R_e(\pi; L). \tag{6}$$

The sum is undiscounted and ends at termination or the specified evaluation limit. For example, two episode returns of 10 and 20 give $\bar{R} = 15$.

Let π_{ref} be the reference policy for the same task and architecture. The normalized metric in Eq. 2 is

$$\text{RelativeReturn}(\pi; L \mid \pi_{\text{ref}}) = 100\, \frac{\bar{R}(\pi; L)}{\bar{R}(\pi_{\text{ref}}; 0)}, \qquad \bar{R}(\pi_{\text{ref}}; 0) > 0. \tag{7}$$

For a fixed-policy delay sweep, $\pi_{\text{ref}} = \pi$, so this is the percentage of that policy's zero-latency return retained under delay. For the before/after-training comparison, π_{ref} is the original zero-latency-trained policy in both cases; π is either that policy or the latency-trained policy. The denominator stays fixed. For an illustrative reference score of 100, before/after scores of 20 and 120 give 20% and 120%, respectively. These percentages are not success rates. Success rate instead counts successful episodes divided by the number of evaluation episodes, multiplied by 100.

The task-level training objective is high expected episode return under the chosen latency conditions:

$$\max_{\theta} \mathbb{E}[R(\pi_\theta; L)], \qquad L \sim \mathcal{P}_{\text{train}}. \tag{8}$$

Here θ denotes the policy parameters and $\mathcal{P}_{\text{train}}$ the distribution of training conditions. The expectation also includes rollout randomness. A profile is one such condition and can itself generate varying per-request delays. The recipe in §4.1 trains the small teacher with reinforcement learning and the VLA through distillation and DAgger.

B.2 Timing and Action Scheduling

Inference workers and capacity. An *inference worker* is a process that handles one policy-inference request at a time: it receives an observation, runs the model, and returns a predicted action or chunk. Multiple workers can process different requests in parallel. The calibration experiments use two GPUs and two workers per deployment. If all workers are occupied when an observation is scheduled, that request is dropped. Only requests accepted by a worker receive an index i in the equations. We keep the observation schedule fixed within each comparison; capacity drops can still reduce the number of observations processed by the policy. The process busy time determines when a worker can accept another request. These are execution details of the profiling and replay protocol, rather than properties of action chunking itself. The exact timing measurements are given in Appendix D.

Fixed and varying delays. For a fixed delay, we omit the request index and write Δt and Δk. For a varying profile, we retain i. Equation 1 converts milliseconds to frames by dividing by the frame duration and rounding up. We write the frame duration as T_{frame} within one setup and $T_{\text{frame},d}$ when distinguishing deployments d. At environment rate f frames per second, $T_{\text{frame}} = 1000/f$ ms; the deployment-specific version is $T_{\text{frame},d} = 1000/f_d$. The superscript busy consistently marks the process busy time in either unit, giving Δt_i^{busy} and Δk_i^{busy}. It does not denote an additional action delay.

Prediction and availability. Request i observes frame k_i^{obs} and predicts $a'_i = \pi(o_{k_i^{\text{obs}}})$ in the single-action setting, where the input includes any configured history or latency signal. Its result becomes available at k_i^{ready} according to Eq. 3. For the single-action setting, let k_i^{exec} denote the frame when this prediction is first applied. It does not describe when the resulting physical motion finishes.

Ordered execution. Requests are numbered from $i = 1$ and environment frames from $k = 0$. Only requests accepted by an inference worker receive an index. Predictions are applied in request order, with at most one new prediction applied per frame:

$$k_i^{\text{exec}} = \max\Big(k_i^{\text{ready}}, k_{i-1}^{\text{exec}} + 1\Big), \qquad k_0^{\text{exec}} = -1. \tag{9}$$

Thus readiness and first application can differ even when inference is parallel. For illustration, worker A receives request 1 at frame 10 but does not return its result until frame 14. Worker B receives request 2 at frame 11 and returns its result at frame 13. Request 2 is ready first, but the ordering rule waits for request 1: their actions are first applied at frames 14 and 15, respectively. Request 2's action latency is two frames; its additional two-frame wait after readiness is a scheduling delay. The inference resources can be released when results arrive even though those results are still waiting to be applied.

Action between predictions. For tasks that repeat the most recently applied action,

$$a_k = a'_i \quad \text{for} \quad k_i^{\text{exec}} \le k < k_{i+1}^{\text{exec}}. \tag{10}$$

The final prediction is held until episode termination. Tasks that use a no-op between predictions follow that rule instead. Before the first prediction, the environment uses its designated idle action. The rule is fixed within each comparison. These equations describe one predicted action per request; the chunked configurations and their execution settings are documented in Appendix C.

Frames, decisions, and chunks. A frame is one benchmark environment step, not a physics substep. A decision step spans the interval between scheduled observations and may contain several frames. For example, the Atari configurations with stride four have four raw frames per decision step. We report decision-step delays explicitly rather than using Δk for a different unit. Action-chunk length H counts predicted actions; visual-history length C counts retained observation frames. For example, $H = 8$ means predicting eight actions per inference, while $C = 8$ means retaining eight observation frames. Neither is a delay or an episode length.

C Implementation Details

This appendix reports the experimental settings used for the figures in the paper.

Latency units and environment rates. Latency values denote action-delivery delay unless explicitly marked as process busy time. For fixed-latency experiments, a delay of Δk raw frames corresponds to $\Delta t = \Delta k\, T_{\text{frame}} = 1000\Delta k/f$ milliseconds at environment rate f frames per second. Flappy Bird runs at 10 Hz in the eight-observation decision-history results, so one raw frame is 100 ms; the earlier fixed-delay sweeps use 30 Hz, or 33.33 ms per raw frame. Demon Attack runs on raw Atari frames at 60 Hz while observations and actions are emitted at 15 Hz with frameskip 4; the policy therefore makes one decision every four raw frames, or every 66.67 ms. Deadly Corridor and Defend the Line run at 35 raw FPS with 8.75 Hz observation and action updates, so the policy again makes one decision every four raw frames, or every 114.29 ms. Asterix and Atlantis use the same 60 Hz raw rate and 15 Hz policy decision rate as Demon Attack. MIKASA InterceptGrabFast (Cherepanov et al., 2026a) is evaluated at 20 Hz, and HumanoidBench Balance Simple at 50 Hz in the plotted profile experiments. Some Atari and VizDoom configurations specify latency in policy steps; we convert these to raw frames using the frameskip above.

Table 4: Environment rates and decision strides determine how a delay in raw frames maps to policy decisions and wall-clock time. Environment rate is the benchmark stepping rate, not the physics substep rate. Policy rate is the frequency at which the policy receives an observation and can issue a new action. Decision stride is the number of raw environment frames between these policy calls. Model-dependent entries are listed in OpenVLA / GR00T / $\pi_{0.5}$ order for the profile-trained evaluation condition.

Environment	Environment rate (FPS)	Policy rate (Hz)	Decision stride
Flappy Bird (fixed-delay sweeps)	30	30	1 frame
Flappy Bird (Table 1)	10	10	1 frame
Demon Attack	60	15	4 raw frames
Deadly Corridor	35	8.75	4 raw frames
Asterix / Atlantis	60	15	4 raw frames
Defend the Line	35	8.75	4 raw frames
MIKASA InterceptGrabFast	20	20	1 raw frame
HumanoidBench Balance Simple	50	50	1 raw frame
InvertedPendulum	10 / 5 / 5	10 / 5 / 5	1 frame
AirRaid	50 / 50 / 10	12.5 / 12.5 / 2.5	4 frames
Ant	10 / 5 / 5	10 / 5 / 5	1 frame
Hopper	10 / 5 / 5	10 / 5 / 5	1 frame
Walker2D	10 / 5 / 5	10 / 5 / 5	1 frame

The Gymnasium rates are configured rates from the results record. The archived WanOFT MIKASA H1 result uses 10/10 Hz; the Qwen MIKASA results use 20/20 Hz as listed above.

VLA model families. OpenVLA denotes the StarVLA QwenOFT implementation initialized from StarVLA/Qwen3VL-OFT-Bridge-RT-1 at checkpoints/steps˙5000˙pytorch˙model.pt. The base vision-language model (VLM) is StarVLA/Qwen3-VL-4B-Instruct-Action. Unless overridden, QwenOFT uses AdamW with learning rates $2 \cdot 10^{-5}$ for the base model, 10^{-5} for the Qwen-VL interface, and 10^{-4} for the action model, FlashAttention 2, gradient checkpointing, image size 224, and an effective batch size of 128. The standard RL-games SFT configuration sets per-device batch size 32 and gradient accumulation 4 on one process. Deadly Corridor uses the multibinary seven-action layout and binary cross-entropy (BCE) loss in the corresponding configuration.

The $\pi_{0.5}$ rows use the QwenPI v3 framework initialized from StarVLA/Qwen3VL-PI˙v3-Bridge-RT˙1 at step 50,000, with a LayerwiseFM action model, horizon 16, two repeated diffusion steps, four inference timesteps, and learning rates $5 \cdot 10^{-6}$ for both base and action modules. The GR00T

rows use StarVLA/Qwen3VL-GR00T-Bridge-RT-1 at step 20,000, a diffusion-transformer (DiT-B) action model with horizon 8, four inference timesteps, and default learning rates $2 \cdot 10^{-6}$; the Demon Attack GR00T run used an explicit lower-learning-rate configuration with 10^{-6} for the base/interface and 10^{-5} for the action model. WanOFT uses the Wan-AI/Wan2.2-TI2V-5B-Diffusers world model initialized from StarVLA/WM4A-Wan2d2-OFT-LIBERO-4in1 at step 60,000, scaled dot-product attention (SDPA), an action-query multilayer perceptron (MLP) head, ordered context windows, and learning rates $2.5 \cdot 10^{-5}$ for the base, 10^{-5} for the interface, and 10^{-4} for the action model and action-query projector.

Zero-latency degradation. Figures 2 and 6 use the StarVLA/LatencyBench convention above. Figure 6 trains OpenVLA fixed-latency specialists. Flappy Bird uses five fixed training latencies 0–4, 200 episodes per latency, seed 42, 5,000 optimizer steps, and post-training evaluation at latencies 0–4 for at least 20 episodes and a 3,600-step cap. Demon Attack trains specialists at source settings 0, 2, 4, 6, and 8 policy steps, plotted as 0, 8, 16, 24, and 32 raw frames. Each Demon Attack specialist uses 200 training episodes and a final step-7,000 checkpoint; evaluation uses at least 20 episodes per cell and a 14,400 raw frame cap, equivalent to 3,600 decisions. Deadly Corridor uses the same 0/2/4/6/8 decision-interval grid, plotted as 0/8/16/24/32 raw frames; each specialist uses 1,000 training episodes and the step-500 checkpoint. These train–evaluation grids use one training seed per cell and at least 20 evaluation episodes per cell. Because each cell has one training seed, the reported variation comes from evaluation episodes and does not measure variation across training runs.

Figure 2 additionally compares OpenVLA, $\pi_{0.5}$, and GR00T zero-latency policies on Flappy Bird, Demon Attack, Deadly Corridor, and MIKASA InterceptGrabFast. The Flappy and Demon VLA runs use 200-episode zero-latency datasets; Deadly Corridor uses 1,000 episodes. The following checkpoints are used for evaluation: Flappy $\pi_{0.5}$ at step 5,000, Flappy GR00T at step 3,000, Demon $\pi_{0.5}$ and GR00T at step 5,000, and Deadly Corridor OpenVLA, $\pi_{0.5}$, and GR00T at step 500. The MIKASA panel evaluates QwenOFT, QwenPI v3, and QwenGR00T H1/H8 checkpoints for at least 20 episodes at fixed delays 0–4 in the 20 Hz environment. QwenPI v3 H1 uses an isolated inference config with interleave˙self˙attention: false to match its trained action architecture, without changing weights. The game evaluation JSON files retained for Figure 2 report at least 20 episodes per latency condition.

Profiling and timing calibration. The calibration compares simulated timing with measured inference on real hardware while both use digital task environments. Figures 4 and 5 use an OpenVLA Flappy Bird policy trained at a fixed two-frame delay. The real deployments run a warmed runtime with two GPUs and two inference workers. The profiled quantity is a composed action-delivery latency: observation capture, inter-process communication (IPC), worker-side preprocessing, CUDA-synchronized policy inference, action decoding, and result delivery until the action is available to the scheduler. Environment stepping after action arrival is not part of the profiled latency. Figure 4 evaluates the same policy on five complete deployment instances: RTX 3090, RTX 4090, RTX 5090, L40S, and A100 SXM4 80GB. Each instance is calibrated to a frame rate that maps its measured latency profile to the two-frame training condition while satisfying the 80% two-worker utilization cap. The calibrated rates are 22, 24, 30, 38, and 23 FPS, respectively; the full hardware, driver, CUDA/PyTorch/Triton, RAM, power, NUMA, and latency-sample inventory is reported in Tables 6–7. Fixed and Normal set process busy time equal to the rounded action latency; IID and Temporal sample both action-delivery latency and process busy time and therefore reproduce when each of the two processes can accept another request.

Latency-aware and mixed-latency VLA training. The OpenVLA mixed-latency and prompt-ablation runs use the same QwenOFT/Bridge initialization and optimizer settings as the fixed-latency runs. The Flappy latency-information ablation in Table 2 trains two policies on latencies $\{0, 1, 2, 3, 4\}$ with 40 episodes per latency (200 total), 5,000 optimizer steps, seed 42, batch 32, gradient accumulation 4, and post-training fixed-latency evaluation at 0–4 for at least 20 episodes per delay with a 3,600-step cap. The two variants differ only in whether the training and evaluation prompts include the explicit latency suffix: the prompt-aware variant uses the normal mixed-latency prompt, while the prompt-neutral variant does not have any latency information in the prompt.

Prompt templates. For the RL-game VLA experiments, each policy query consists of the visual observation input and one natural-language task prompt.

The prompt-aware condition uses the task prompt plus an explicit latency sentence. The prompt-neutral ablation removes the latency information by splitting the prompt at the exact marker Current

Table 5: Prompts used for the six RL-game VLA tasks. The middle column gives the task instruction. In the latency-aware condition, the evaluator appends "Current action latency is Δk raw frames (Δt ms). The environment runs at f_{env} FPS and observations are emitted at f_{obs} FPS. Choose the best next action." The final column gives the environment and observation rates substituted into that sentence.

Task	Task prompt prefix	Rates
Flappy Bird	You are playing Flappy Bird. Pass through the pipe gaps and stay alive. Choose the action: NOOP, FLAP.	$f_{\text{env}} = 30$, $f_{\text{obs}} = 30$
Demon Attack	You are playing Demon Attack from a single game image. Choose exactly one action from: NOOP, FIRE, RIGHT, LEFT, RIGHTFIRE, LEFTFIRE. Survive, avoid enemy fire, and shoot enemies to maximize score.	$f_{\text{env}} = 60$, $f_{\text{obs}} = 15$
Deadly Corridor	You are playing Deadly Corridor in VizDoom. Your goal is to survive the corridor, fight enemies, and reach the green armor vest at the far end. The available actions are: MOVE_FORWARD, MOVE_BACKWARD, MOVE_LEFT, MOVE_RIGHT, TURN_LEFT, TURN_RIGHT, and ATTACK. Choose the best next action from the current visual observation.	$f_{\text{env}} = 35$, $f_{\text{obs}} = 8.75$
Asterix	Collect useful objects and avoid lyres. Choose exactly one action from: noop, up, right, left, down, upright, upleft, downright, downleft.	$f_{\text{env}} = 60$, $f_{\text{obs}} = 15$
Atlantis	Defend Atlantis by firing at descending enemies. Choose exactly one action from: noop, fire, rightfire, leftfire.	$f_{\text{env}} = 60$, $f_{\text{obs}} = 15$
Defend the Line	You are playing Defend the Line in VizDoom. Your goal is to survive by turning toward enemies and shooting them before they reach you. The available actions are: TURN_LEFT, TURN_RIGHT, and ATTACK. Choose the best next action from the current visual observation.	$f_{\text{env}} = 35$, $f_{\text{obs}} = 8.75$

action latency is and keeping only the prefix. For example, the latency-neutral Flappy Bird prompt is "You are playing Flappy Bird. Pass through the pipe gaps and stay alive. Choose the action: NOOP, FLAP." The latency-aware prompt appends the latency sentence shown in Table 5. The prompt templates cover latencies $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 20, 25, 30\}$ in the relevant latency unit, with raw-frame and millisecond values filled in for each entry.

Memory interfaces. Figure 7 compares temporal input interfaces under matched fixed-delay conditions. The OpenVLA rows use checkpoint 4,000 and are evaluated with the standalone LatencyBench evaluator for at least 20 episodes per cell. Flappy Bird uses latency 3 raw frames and a 3,600 raw-frame horizon; Demon Attack uses latency 6 raw frames and a 14,400 raw-frame horizon; Deadly Corridor uses latency 6 raw frames and a 3,600 raw-frame horizon. The four standard OpenVLA methods are single-frame input, four-frame prompt, four-frame 2×2 grid, and packed streaming key-value (KV) memory. In KV training, *replay* means processing a saved observation sequence to reconstruct its attention history during supervised updates. Serial replay processes its frames sequentially; packed replay processes the sequence jointly with the same causal attention window. This differs from latency-profile replay, which samples timing for new task interactions. Their training configurations use DeepSpeed ZeRO-2 with two processes and save safetensors checkpoints. The single-frame and grid variants use effective batch 128; the prompt variant uses 128 four-image prompts; the packed and serial KV variants use 32 eight-step rollouts per optimizer step, i.e., 256 per-frame action labels. KV inference encodes only the newest frame and reuses a four-frame per-environment cache reset at episode boundaries. Packed KV replay uses FlexAttention and full temporal backpropagation; the serial KV implementation has the same inference cache but detaches cached states during training.

The ghost-trail rows are trained from regenerated teacher datasets that overlay motion history into a single image. The Flappy Bird and Demon Attack ghost-trail datasets replay the same fixed-latency teacher episodes as the memory-rollout datasets, with 15-frame histories; Flappy uses latency 3 and Demon Attack uses latency 6. The WanOFT rows in the memory table use separate WanOFT checkpoints with ordered five-frame context. Their raw-frame latencies are 3 for Flappy Bird and 6 for Demon Attack and Deadly Corridor, corresponding to 100.0 ms, 100.0 ms, and 171.43 ms,

respectively. WanOFT memory-table evaluations use at least 20 episodes and are merged with the OpenVLA table only after recomputing the same Student-t interval convention.

The WanOFT Flappy latency-transfer grid in Figure 11 trains five separate context-5 policies at fixed latencies 0–4. Each policy uses 200 episodes, seed 42, 2,000 optimizer steps, per-device batch size 4, gradient accumulation 32, effective batch 128, current-action cross-entropy, and five ordered frames at offsets $[-4, -3, -2, -1, 0]$. Evaluation uses the same 0–4 fixed-latency grid, at least 20 episodes per cell, and a 3,600-step cap.

MIKASA fixed-latency grid snapshot. The MIKASA panel of Figure 6 uses the reported naive H8 train-zero column (seven-dimensional proprioception, the robot's measured internal state; DAgger step 5,000; 200 held-out episodes, seeds 4242424242–4242424441). Its four fixed-delay columns use QwenOFT H8 with a 63D state: seven proprioceptive values and eight slots for seven-dimensional actions aligned to the observation time. Let K be the number of committed actions in the prefix. For a policy trained at delay Δk_{train}, $K = \Delta k_{\text{train}}$ because this task uses one control step per raw frame. The first K slots contain those committed actions; the remaining slots are zero, and all prefix inputs use fixed $[-1, 1]$ bounds. Every conditioned column uses SFT step 5,000 followed by DAgger step 3,500, with 100 development episodes per cell and seeds 4242424442–4242424541. These are fixed-step snapshots, not independently selected best checkpoints or held-out final results. Evaluation uses 20/20Hz, 60 transitions, 10 parallel environments and inference batch 8; prefix length stays fixed while eval latency varies.

WanOFT timing probes. Figures 12 and 13 probe frozen Flappy Bird WanOFT representations at fixed latency 2. A *linear probe* is a classifier trained on frozen model features to predict a label, here the action or its timing; the VLA weights remain fixed. The pre-SFT checkpoint is the official LIBERO WanOFT step-60,000 initialization; the post-SFT checkpoint is the Flappy context-5 WanOFT step-2,000 policy. The probe data consist of five training episodes (9,112 decision frames) and two disjoint validation episodes (7,200 decision frames), selected with episode seed 3047. Each probe is an independent linear classifier trained for 50 epochs with AdamW, batch size 512, learning rate 10^{-3}, weight decay 10^{-4}, class-weighted cross-entropy, and probe seed 12109. The input-control seed is 6201 and the variational-autoencoder (VAE) seed is 9109. The reported controls retrain the probe under ordered context, shuffled frame order, repeated-current-frame context, or latency-neutral prompting; they do not evaluate a single probe under distribution shift.

Small-policy sweeps. Appendix Figures 8, 9, and 10 use our benchmark codebase's Sample Factory asynchronous proximal policy optimization (APPO) backend, not VLA SFT. The base configs train for 25M environment steps. Training or rollout workers below collect task trajectories; inference workers in the profiling protocol instead serve model requests and determine simulated inference capacity. Flappy Bird uses CUDA sampling/rendering, two workers, 128 environments per worker, rollout 128, batch size 4096, eight epochs, eight batches per epoch, learning rate $2.5 \cdot 10^{-4}$, and non-recurrent APPO. Demon Attack uses the GPU Atari backend at 60 raw FPS and 15 observation FPS, one worker with 512 environments, rollout 64, recurrence 32, batch size 1024, two epochs, eight batches per epoch, learning rate $3.03 \cdot 10^{-4}$, and raw reward evaluation. Deadly Corridor uses the VizDoom backend at 35 raw FPS and 8.75 observation FPS, 32 workers with four environments each, rollout and recurrence 128, batch size 1024, two epochs, two batches per epoch, learning rate 10^{-4}, a gated recurrent unit (GRU) of size 512, and raw reward evaluation.

The jitter-robustness sweep uses three seeds per environment. Flappy Bird uses seeds 911, 922, and 933 with fixed latencies 1, 2, 3, Normal$(2, 1)$ clipped to $[0, 4]$, Uniform$(1, 3)$, and two mixed rows. Demon Attack and Deadly Corridor use seeds 0, 42, and 114514 with fixed latencies 2, 4, 6, Normal$(4, 1)$ clipped to $[0, 15]$, Uniform$(2, 6)$, and the corresponding mixed rows. Evaluation uses at least 20 episodes for Flappy Bird, Demon Attack, and Deadly Corridor in the source configs, and the reported summaries retain the finest available statistical unit. The observation-stride and context-window sweeps use seeds 10–14 and at least 20 evaluation episodes per seed. Observation-stride sweeps evaluate Flappy strides 1–5, Demon Attack strides 2–10 in the extended setting, and Deadly Corridor strides 1–6. Context-window sweeps vary stacked frames: Flappy 1–8 in the extended setting, Demon Attack 4–12 in the extended setting, and Deadly Corridor $\{1, 2, 4, 6, 8, 10, 12, 14, 16\}$. Extended horizons are 36,000 steps for Flappy Bird and Demon Attack and 30,000 steps for Deadly Corridor; the capped variants use 3,600 steps for Flappy Bird and Demon Attack.

Cross-task summary. Figures 1 and 17, and Table 1, combine several sources. In all of them, Zero/Real and Target/Real are evaluated in real time on an RTX 3090; Target training uses the architecture-specific RTX 3090 profile in simulation. Both figures normalize each real-time return

by its task/architecture zero-latency mean; Figure 1 shows 12 tasks in a radar chart, while Figure 17 shows the same tasks as bars. Both show the three architectures separately. The table reports raw mean returns. The MIKASA column uses H8 DAgger best checkpoints, 200 held-out episodes, and 20 Hz evaluation. HumanoidBench Balance Simple uses 64 held-out seeds, 50 Hz observation/control, capacity 1, a 1,000-frame cap, inference seed 941731, inference batch 8, and 16 rollout workers. The QwenOFT Balance policy uses H8, while QwenPI and QwenGR00T use H16. The Flappy Bird, Demon Attack, and Deadly Corridor columns use separately trained zero- and profile-latency H1 policies with eight-observation binary decision history. Each of the 18 checkpoints completed 5,000 updates. All three evaluation conditions use the same 100 seeds (1,000,000–1,000,099), 32 parallel environments, worker capacity 1, and game-specific raw-frame caps of 3,600 / 7,200 / 3,600. Environment/observation rates are 10/10, 60/15, and 35/8.75 Hz, respectively. Each architecture's profile-trained policy is trained with its own RTX 3090 temporal profile; Deadly Corridor uses the corrected training-matched 160×120 HUD rendering protocol.

Compute The supervised VLA training runs were carried out on NVIDIA H100 and NVIDIA RTX 6000-class GPUs, depending on allocation availability. We report optimizer steps, episode counts, batch sizes, and evaluation budgets as the main reproducibility units throughout this appendix, rather than normalizing result by a single wall-clock budget. The timing-calibration experiments use complete two-GPU, two-worker deployments and record GPU model, VRAM, power limits, CPU topology, host-visible RAM, software stack, calibrated environment rate, and the number of accepted latency samples in Tables 6–7.

D Profile Sampling and Concurrent Inference

This appendix specifies the timing model needed to reproduce the multi-worker profile experiments. These details determine whether a request can be processed and when its action becomes available; they are not part of the policy-adaptation objective. We retain the operational definitions and fitted state transitions here. Quantile-table construction and numerical edge handling are implementation details and do not change the definitions summarized below.

Action latency and process busy time. Appendix B.2 defines an inference worker as a process that serves one request at a time. For request i accepted by a worker, let t_i^{obs} be the observation time, t_i^{start} the time at which its worker starts serving it, and t_i^{ready} the time at which the inference parent receives the predicted action or chunk. We record two overlapping durations:

$$\Delta t_i = t_i^{\mathrm{ready}} - t_i^{\mathrm{obs}}, \qquad \Delta t_i^{\mathrm{busy}} = t_i^{\mathrm{ready}} - t_i^{\mathrm{start}}. \tag{11}$$

The action latency Δt_i determines action readiness. The process busy time $\Delta t_i^{\mathrm{busy}}$ determines when the worker can accept another request. The two intervals overlap and must not be added. For example, if an observation arrives at 500 ms, its worker starts at 510 ms, and the result arrives at 650 ms, then action latency is 150 ms and process busy time is 140 ms.

For deployment d, the fitted records are

$$\mathcal{D}_d = \{(\Delta t_i, \Delta t_i^{\mathrm{busy}}, j_i, e_i, s_i)\}_{i=1}^{N_d}, \tag{12}$$

where j_i, e_i, and s_i identify the worker, episode, and measurement run. Records retain their order within each run. A scheduled request is accepted only when a worker is available; otherwise it is dropped. Dropped requests remain in the raw attempt trace for capacity accounting but do not enter a latency distribution. The calibration experiments use two GPUs and two workers per deployment.

Fixed and Normal profiles. Let $\widehat{\mu}_d$ and $\widehat{\sigma}_d$ be the mean and population standard deviation of action latency over accepted records. Fixed assigns every request the measured mean,

$$\Delta t_i = \widehat{\mu}_d. \tag{13}$$

Normal draws requests independently and clips negative durations:

$$x_i \overset{\mathrm{iid}}{\sim} \mathcal{N}(\widehat{\mu}_d, \widehat{\sigma}_d^2), \qquad \Delta t_i = \max(0, x_i). \tag{14}$$

Both methods use the rounded action latency as the process busy time. They therefore model capacity with one duration rather than fitting action latency and process busy time separately.

IID profile. IID fits a worker-specific empirical joint distribution of action latency and process busy time, denoted $\widehat{F}_{d,j}^{\mathrm{all}}$, and draws

$$(\Delta t_i, \Delta t_i^{\mathrm{busy}}) \sim \widehat{F}_{d,j_i}^{\mathrm{all}}. \tag{15}$$

Draws are independent across requests. Within one request, the two durations are sampled jointly using empirical inverse CDFs and their measured worker-specific rank correlation. Thus IID preserves the fitted marginal shapes and within-request dependence while removing dependence between consecutive requests.

Temporal profile. Temporal retains the same concurrent-inference interface but adds normal and slow periods. A slow period is a sequence of requests during which one or more inference processes repeatedly remain busy for a long time. For worker j, the slow threshold is

$$\Delta t^{\mathrm{thr}}_{d,j} = 1.25 \min_{(s,e)} \mathrm{median}\{\Delta t^{\mathrm{busy}}_i : i \in \mathcal{D}_{d,s,e,j}\}. \tag{16}$$

Records above this threshold are marked slow. Within a worker's ordered stream, marked records separated by at most 30 submitted records belong to the same local interval. The union of local intervals across workers defines shared slow periods. Normal-period distributions are fitted from records outside these intervals.

For a worker j in measured slow period b, let $q_{b,j}$ be the fraction of its requests marked slow. The period-specific mixture is

$$\widehat{F}^{\mathrm{period}}_{d,j,b} = (1 - q_{b,j})\widehat{F}^{\mathrm{base}}_{d,j} + q_{b,j}\widehat{F}^{\mathrm{slow}}_{d,j,b}. \tag{17}$$

Here $\widehat{F}^{\mathrm{base}}_{d,j}$ is the worker's normal-period joint distribution and $\widehat{F}^{\mathrm{slow}}_{d,j,b}$ describes its slow requests. The sampler fits the frequency of entering a slow period from the measured normal-to-slow and normal-to-normal transition counts. For each simulated episode, it draws

$$p^{\mathrm{slow}}_e \sim \mathrm{Beta}\big(N_{\mathrm{normal}\to\mathrm{slow}} + \tfrac{1}{2}, N_{\mathrm{normal}\to\mathrm{normal}} + \tfrac{1}{2}\big)\,. \tag{18}$$

A Bernoulli draw with this probability can start a slow period while the state is normal. Its duration is drawn from the empirical distribution of measured slow-period lengths. During the period, worker-specific slow fractions and levels are sampled from measured periods and held fixed for that period. This preserves consecutive slow requests without replaying a recorded action or trajectory.

Clock conversion and request capacity. For deployment-specific frame duration $T_{\mathrm{frame},d}$, both durations are converted using the rule from Eq. 1:

$$\Delta k_i = \left\lceil \frac{\Delta t_i}{T_{\mathrm{frame},d}} \right\rceil, \qquad \Delta k^{\mathrm{busy}}_i = \left\lceil \frac{\Delta t^{\mathrm{busy}}_i}{T_{\mathrm{frame},d}} \right\rceil. \tag{19}$$

Action readiness uses Δk_i; worker availability uses $\Delta k^{\mathrm{busy}}_i$. The simulator accepts or drops requests using the same rule as the measured deployment.

Scope of the comparison. The four profiles do not isolate only the order of requests. Fixed and Normal use one duration for both action readiness and process busy time, IID samples their joint distribution independently across requests, and Temporal also models consecutive slow requests. The methods therefore differ in which delays they generate, how they limit concurrent requests, and whether slow requests occur consecutively. Our validation asks which complete profile best reproduces hardware-timed return; it does not attribute the observed gap to temporal ordering alone.

E Simulated Versus Measured Timing: Deployments and Protocol

This comparison contrasts assigned timing in simulation with measured inference timing on physical GPUs; the task environment remains digital in both conditions. This section describes the frozen deployment artifacts used by these experiments. Their measurement definition and model versions are historical; Appendix D specifies the current profiling and sampling implementation and distinguishes it from these artifacts.

Comparison protocol. Each device label in this comparison denotes a complete measured deployment instance rather than a nominal GPU model alone. Across instances, we use the same OpenVLA policy and checkpoint configuration, environment semantics, observation-stride policy, and two-GPU, two-worker inference configuration. The policy was trained for a fixed latency of $\Delta k_{\mathrm{train}} = 2$ environment frames. Because the same model exhibits different measured end-to-end action-delivery latency on different instances, we calibrate the environment rate separately for each

instance. For an action latency Δt_i in milliseconds and a calibrated environment rate f_d in frames per second, the corresponding frame duration and discrete delay are

$$T_{\mathrm{frame},d} = \frac{1000}{f_d}, \qquad \Delta k_i = \left\lceil \frac{\Delta t_i}{T_{\mathrm{frame},d}} \right\rceil .$$

Among integer rates that preserve the observation stride and satisfy the 80% two-worker utilization cap, we select the rate that maximizes the number of profile samples mapped to $\Delta k_{\mathrm{train}}$. Ties are resolved by the mean absolute frame error, proximity to the mean-latency rate $1000\Delta k_{\mathrm{train}}/\widehat{\mu}_d$, where $\widehat{\mu}_d$ is the measured mean action latency in milliseconds, and then the lower rate. The resulting instance-specific rates normalize physical time to the common frame-relative latency used during training; they are calibrated environment rates, not GPU throughput measurements or different task settings.

For each deployment instance, real execution and simulation use the same calibrated rate, observation schedule, and two-worker capacity. Simulation samples timing values from that instance's fitted profile on a virtual frame clock. Consequently, the physical speed of the machine running the simulator affects only evaluation time in wall-clock units; it does not enter the simulated action delay. This yields a matched comparison between real execution on an instance and simulation using timing values sampled from that instance's fitted profile.

Measured deployment conditions. The historical profiled quantity is a composed action-delivery latency. It combines CUDA-synchronized policy inference with measured pipeline stages from observation capture through IPC, preprocessing, inference, action decoding, and result delivery until the action is made available to the real-time scheduler. This quantity sums instrumented stages rather than taking one uninterrupted elapsed-time measurement; uninstrumented coordinator waiting is not included. It excludes the subsequent environment-step execution. CPU, memory, GPU placement, power configuration, and the software runtime can all affect this measured latency, so we report them as part of each deployment condition.

Rented-infrastructure reproducibility. The deployment machines were rented from third-party GPU infrastructure. We selected and verified the GPU model and count and kept the policy and experimental configuration fixed, while recording the host topology, visible memory, CPU allocation, power limits, and runtime exposed by each instance. Provider-managed frequency or power policies and transient resource contention may not be fully observable. These factors can change the absolute latency obtained on a new rental even when its nominal GPU model and count match. Reproduction on a new instance should therefore re-profile its end-to-end latency and recalibrate its environment rate rather than reuse the FPS reported here. Sampling from the same fitted profile preserves the latency condition; it does not require the simulator host to reproduce the source host topology.

Table 6: Hardware used for the five profiling deployments. Each row describes the complete two-GPU host, including GPU memory and power settings, CPU and memory allocation, and GPU-to-NUMA placement. Because these host settings differ, the deployments do not isolate GPU model as the only variable.

Deployment	GPU memory	Power limits (GPU 0/GPU 1)	Host CPU and allocation	Host-visible RAM	GPU-to-NUMA mapping
RTX 3090	24,576 MiB	270/270 W; default 420/350 W	EPYC 7642 48-Core; 2 sockets / 192 logical CPUs / 8 NUMA; quota 73.728 CPUs	503.69 GiB	0→3; 1→2
RTX 4090 (48 GB)	49,140 MiB	450/450 W; default 450/450 W	EPYC 7B13 64-Core; 1 socket / 128 logical CPUs / 1 NUMA; quota n/r	503.51 GiB	0→0; 1→0
RTX 5090	32,607 MiB	600/575 W; default 600/575 W	EPYC 9734 112-Core; 2 sockets / 224 logical CPUs / 8 NUMA; quota n/r	755.49 GiB	0→2; 1→0
L40S	46,068 MiB	350/350 W; default 350/350 W	EPYC 9334 32-Core; 2 sockets / 128 logical CPUs / 2 NUMA; quota n/r	1007.54 GiB	0→1; 1→1
A100 SXM4 80 GB	81,920 MiB	275/275 W; default 275/275 W	EPYC 7742 64-Core; 1 socket / 128 logical CPUs / 4 NUMA; quota 61.440 CPUs	503.71 GiB	0→2; 1→1

Note. Each deployment used two GPUs and two inference workers. Power limits and NUMA mappings follow GPU 0/GPU 1 order. RAM is host-visible; quota is the cgroup CPU allocation; n/r means not recorded. The 48 GB RTX 4090 reports 49,140 MiB/GPU.

Table 7: Software and calibrated environment rate for each profiling deployment. Hardware-timed and simulated evaluation use the same calibrated rate within a row. The final column gives the number of valid action-latency records used to fit that deployment's profile.

Deployment	Driver	Core runtime	Kernel	Calibrated env. FPS	Valid latency records
RTX 3090	595.58.03	CUDA 12.6 PyTorch 2.6.0+cu126 Triton 3.2.0	Linux 6.8.0-134-generic	22	$N_d = 251{,}324$
RTX 4090 (48 GB)	580.95.05	CUDA 12.6 PyTorch 2.6.0+cu126 Triton 3.2.0	Linux 6.8.0-90-generic	24	$N_d = 1{,}172{,}204$
RTX 5090	570.153.02	CUDA 12.8 PyTorch 2.7.1+cu128 Triton 3.3.1	Linux 6.8.0-60-generic	30	$N_d = 1{,}014{,}625$
L40S	560.35.03	CUDA 12.6 PyTorch 2.6.0+cu126 Triton 3.2.0	Linux 6.8.0-59-generic	38	$N_d = 788{,}377$
A100 SXM4 80 GB	570.211.01	CUDA 12.6 PyTorch 2.6.0+cu126 Triton 3.2.0	Linux 6.8.0-124-generic	23	$N_d = 1{,}270{,}442$

Note. All rows use Ubuntu 24.04, Transformers 4.57.0, and FlashAttention 2.8.3.post1. The calibrated environment rate is selected per deployment instance. N_d denotes valid, applied, non-dropped end-to-end latency records pooled over five profiling runs of 100 episodes each; it is not the number of evaluation episodes.

F Small-Policy Latency-Training Sweep

We run a controlled small-policy sweep over five non-mixed latency profiles and two mixed-latency training profiles across Flappy Bird, Demon Attack, and Deadly Corridor (Figure 8). Columns denote the training profile and rows denote the evaluation profile. Cell labels show mean return; colors use a linear scale relative to each task's smallest fixed train/eval delay baseline: (1,1) for Flappy Bird and (2,2) for the other tasks, in raw frames. Thus, the heatmaps support within-task comparisons rather than direct cross-task comparisons of return magnitude.

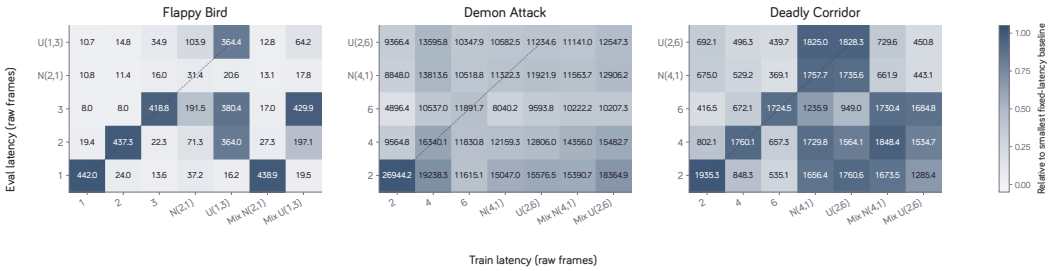

Figure 8: No training profile gives the highest return across all tasks and evaluation profiles in this small-policy sweep. Columns specify the training profile and rows specify the evaluation profile; the dashed diagonal marks cells that use the same single profile for training and evaluation. $\mathrm{N}(\mu, \sigma)$ is a clipped Normal distribution over frame delays, and $\mathrm{U}(a, b)$ samples integer delays uniformly from a to b; Mix denotes training across multiple listed conditions. Cell labels are mean returns over three training seeds. Within each task, color divides the cell value by the result obtained when the smallest fixed delay is used for both training and evaluation: $(1, 1)$ for Flappy Bird and $(2, 2)$ for the other tasks. Episode limits are 3,600 steps for Flappy Bird, 36,000 for Demon Attack, and 30,000 for Deadly Corridor.

The figure abbreviates a normal delay distribution as N and a uniform distribution as U. For example, N(2,1) specifies Gaussian mean 2 and standard deviation 1 before clipping, while U(1,3) samples between 1 and 3 in the plotted delay units. Mix labels denote training across multiple latency conditions. These are synthetic delay families, distinct from the deployment-fitted profiles of §3.

The resulting train–evaluation matrices show heterogeneous transfer rather than a uniform robustness improvement. Some fixed, stochastic, and mixed profiles retain performance across parts of the evaluation grid, but the transfer pattern depends on the environment and on the particular training distribution; no single profile consistently dominates the others. We therefore treat this sweep as exploratory evidence about possible training effects, rather than as a general small-policy scaling rule.

G Temporal Resolution, Context Span, and Memory Interfaces

The observation-stride sweep (Figure 9) changes the number of environment frames between observation/action updates. The context-window sweep (Figure 10) instead changes the number of retained observation frames, denoted C.

In both figures, panels (a)–(c) show Flappy Bird, Demon Attack, and Deadly Corridor, respectively. Evaluation horizons are 36,000 steps for Flappy Bird and Demon Attack and 30,000 steps for Deadly Corridor. The plots label the two profiles as Fixed delay and Uniform delay. The fixed reference is $\Delta k = 2$ for Flappy Bird and $\Delta k = 4$ for Demon Attack and Deadly Corridor. The matched uniform profile uses bounds of half and one-and-a-half times the fixed reference, expressed in frame units: 1 to 3 for Flappy Bird and 2 to 6 for the other two environments. Shaded regions show 95% percentile bootstrap confidence intervals, obtained by resampling training seeds. This statistical use of bootstrap differs from the initial supervised training stage with the same name.

The sweeps show environment- and profile-dependent optima. They do not support a monotonic rule that more frequent observations or longer histories always improve performance. Frame stacking in these small policies also does not isolate the effect of adding or removing recurrence.

G.1 VLA Memory Interfaces

The left side of Figure 7 in the main text shows inference-time input and memory interfaces. Here *KV replay* reprocesses saved observation sequences during supervised training to reconstruct attention history; it does not refer to latency-profile replay. The plotted KV result uses packed replay, which processes the rollout jointly and lets later action losses update earlier cached features. Serial replay processes frames one at a time and detaches past cache states, blocking those gradients. Both use the same inference-time cache. WanOFT is a separate temporal world-model architecture.

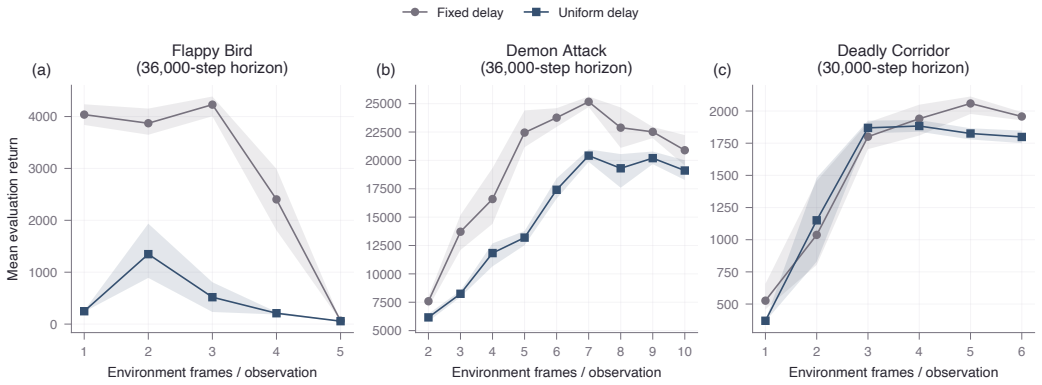

Figure 9: The observation stride that gives the highest return depends on the task and latency profile. The horizontal axis is the number of environment frames between policy observations and action updates, so a smaller stride means more frequent control. Fixed delay is 2 frames for Flappy Bird and 4 frames for the other tasks. Uniform delay samples 1–3 frames for Flappy Bird and 2–6 frames for the other tasks. Each point averages the mean returns of five independently trained policies; shading is a 95% bootstrap confidence interval obtained by resampling those five training seeds. Each policy is trained and evaluated with the same profile.

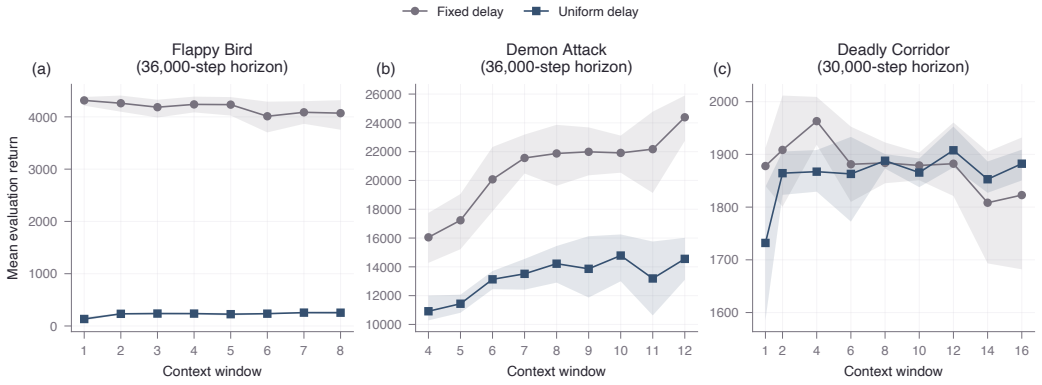

Figure 10: Increasing visual-history length does not improve all tasks under latency. The horizontal axis is context length C, implemented by stacking the latest C observation frames. Fixed delay is 2 frames for Flappy Bird and 4 frames for the other tasks. Uniform delay samples 1–3 frames for Flappy Bird and 2–6 frames for the other tasks. Each point averages the mean returns of five independently trained policies; shading is a 95% bootstrap confidence interval obtained by resampling those five training seeds. Each policy is trained and evaluated with the same profile.

Table 8: Raw returns for the visual-history comparison in Figure 7. No interface has the highest mean on all three tasks. Training and evaluation use the same fixed raw-frame delay, shown in each column header. Cells report mean episode return ± the Student-t 95% confidence-interval half-width from at least 20 evaluation episodes; these intervals measure episode variation for one training run per cell. Bold marks the largest mean in each task. The OpenVLA methods use checkpoint 4000; WanOFT uses a separate backbone and task-specific checkpoints. A hyphen means that Ghost trail was not evaluated on Deadly Corridor.

Method	Flappy Bird (Δk=3)	Demon Attack (Δk=6)	Deadly Corridor (Δk=6)
Single frame	368.9 ± 25.9	5671.8 ± 126.3	232.0 ± 51.3
Four separate frames	352.1 ± 28.2	6474.6 ± 117.8	227.1 ± 33.9
4-frame grid (2×2)	343.1 ± 29.5	5783.2 ± 212.5	**271.1 ± 56.7**
KV memory (packed replay)	382.2 ± 25.1	**6483.5 ± 109.9**	229.2 ± 33.2
Ghost trail	**426.9 ± 13.7**	5688.7 ± 312.7	-
WanOFT	425.3 ± 33.1	4608.8 ± 714.8	238.9 ± 74.2

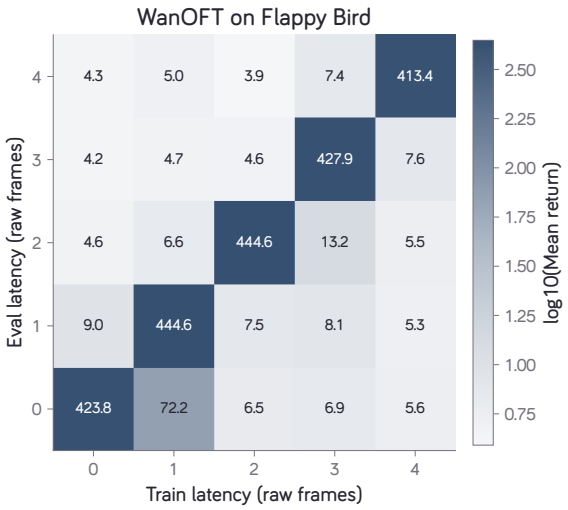

Figure 11: WanOFT performs well mainly when its training and evaluation delays match, despite receiving five observation frames. Columns are fixed training delays and rows are fixed evaluation delays, both from 0 to 4 raw frames on Flappy Bird. Cell labels give mean episode return over 20 evaluations, and color shows $\log_{10}(\max(\bar{R}, 1))$. Each policy receives the current frame and the preceding four frames, uses one training seed, and is evaluated for at most 3,600 steps per episode. At 30 FPS, one raw frame is 33.33 ms. Uncertainty is not shown.

Table 8 reports the unnormalized values underlying the right side of Figure 7, using the corrected evaluator and raw-frame horizon. Earlier standalone evaluations supplied a single image to some multi-frame checkpoints; a subsequent Demon Attack batch used a horizon four times too short. Those invalidated results are excluded. The retained table is based on the replacement evaluations documented in the experiment archive.

H WanOFT Latency Transfer and Timing Probes

The five-frame WanOFT policy loses performance when training and evaluation use different delays (Figure 11). Its five-frame input alone does not make the policy robust to a change in action delay.

The linear-probe experiment trains a linear classifier on frozen features from a model trained with a fixed delay of two frames. It asks whether a simple classifier can read action timing from those features without updating the policy. The plotted feature labels are *DiT temporal delta*, a 9,216-dimensional representation from the diffusion-transformer backbone, and *action query*, a 3,072-dimensional representation associated with action prediction. Each input control trains its own probe on five episodes and evaluates on two disjoint episodes. The controls shuffle frame order, repeat the current frame, or remove the latency suffix. We train a separate probe for each input condition. Comparing the ordered-frame input with each control therefore measures how much usable timing information remains after the frames are changed; it does not measure whether one probe can handle a different input at test time.

The timing labels count decision steps until the next teacher flap action or since the previous one, grouped into classes 0, 1, 2, 3, and $4+$. The last class groups distances of at least four steps. Class F1 is the harmonic mean of precision and recall; Macro-F1 averages class F1 values equally, so frequent timing classes receive no extra weight. Higher values indicate better decoding, with 1 denoting perfect predictions.

We extract the DiT hidden states supplied to the action head with the policy frozen in evaluation mode. For each five-frame input, we group the spatiotemporal tokens into two temporal groups and average the spatial tokens within each group, obtaining $h_0, h_1 \in \mathbb{R}^{3072}$. The DiT temporal-delta feature concatenates these vectors and their difference, $[h_0; h_1; h_1 - h_0]$, yielding a 9,216-dimensional representation. To obtain the action-query feature, we average all DiT tokens, apply the policy's learned linear projection into action queries, and select the 3,072-dimensional query corresponding to the first action in the predicted chunk, before the MLP action head. For each probe, feature normalization statistics are computed on the probe-training split and reused for validation.

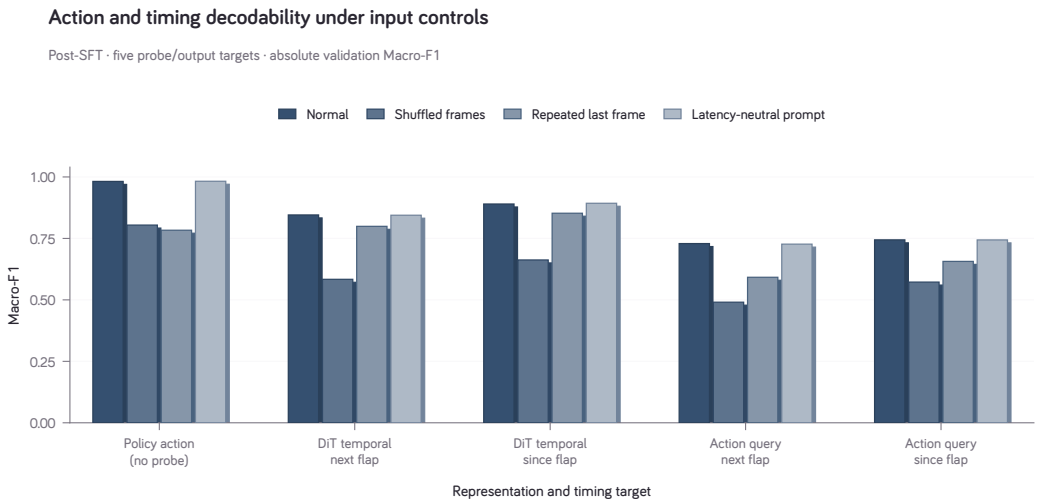

Figure 12: A linear classifier predicts flap timing more accurately from WanOFT features when the five input frames remain in temporal order. Bars report validation Macro-F1, the mean F1 score across timing classes, for predicting the number of decision steps to the next teacher flap or since the previous flap. Policy action is the model's direct output; all other bars use a separate linear classifier trained on frozen features for the displayed input condition. DiT refers to features from the diffusion-transformer backbone. Shuffling frame order reduces post-SFT next-flap Macro-F1 from 0.845 to 0.584 for DiT features and from 0.729 to 0.491 for action-query features. The experiment measures what timing information a classifier can read from the representation, not whether the policy uses that information to act. Results use a single fixed delay of 2 frames, five training episodes, and two validation episodes, so they do not test generalization across latency values.

With ordered post-SFT context, next-flap Macro-F1 is 0.845 for the DiT temporal-delta representation and 0.729 for the action query. Shuffling reduces these to 0.584 and 0.491; repeating the current frame gives 0.799 and 0.592. Removing latency text changes the selected metrics by at most 0.003. Since the data contain only one latency condition, this control cannot establish that the representations encode distinct latency values.

The bucket analysis separates time to the next flap and time since the last flap into decision-step distances 0, 1, 2, 3, and $4+$. It provides a representation-level account of local timing, while the rollout latency grid shows that this information is not sufficient for robust transfer.

I Inference-Time Action Review with GPT-6 Astra

This exploratory study asks whether a frontier reasoning model can supply latency awareness at inference time, without latency training. Sections 4.1 and 5 adapt the VLA through training and timing inputs. Here we ask a complementary question: given the delay and recent observations, can a foundation-model reviewer identify and correct an action that will arrive too late? We use GPT-6 Astra (OpenAI, 2026), abbreviated as Astra, to review OpenVLA action chunks. Inspired by the hybrid action-review interface in GPT-as-Policy (Su et al., 2026), the reviewer tracks the task throughout an episode. Both models' weights remain fixed during evaluation.

We apply this interface to delayed, overlapping action chunks. At each observation selected for review, OpenVLA predicts a chunk of eight actions, using the committed action prefix from §3.1 as context. Astra assesses what earlier actions achieved and whether the new chunk will still be appropriate when it arrives. It receives recent top and wrist camera images, measured robot state, a visual estimate of the ball's state with uncertainty, and a preview of robot motion fitted from past execution.

The evaluation software invokes Astra only at prespecified review events. At each event, it also supplies a candidate correction computed from the robot and ball geometry. Astra returns a textual assessment, then either keeps the OpenVLA chunk or makes a correction within predefined action limits. The returned chunk becomes context for the next OpenVLA prediction. Later observations and the executed-action record show Astra what the earlier actions actually achieved.

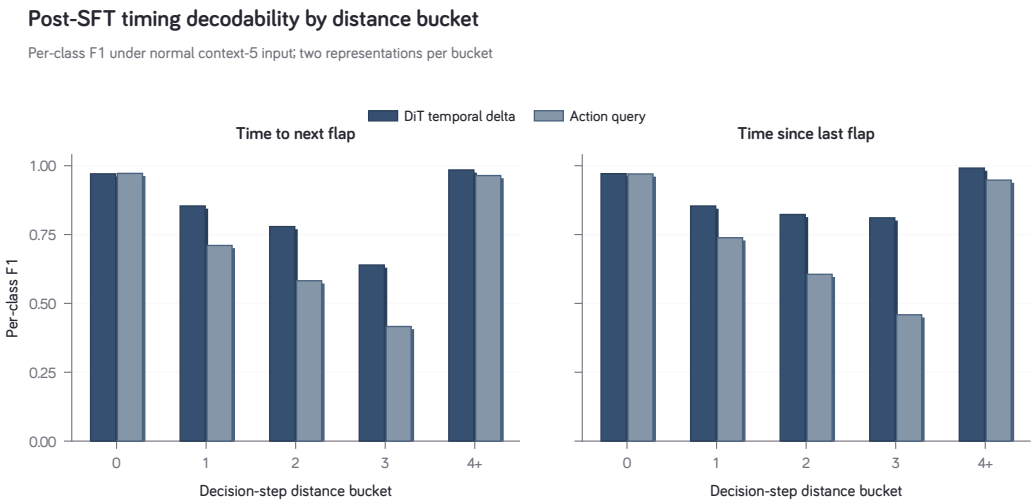

Figure 13: After supervised fine-tuning, a linear classifier predicts action timing more accurately from diffusion-transformer (DiT) features than from action-query features. The left panel predicts decision steps to the next teacher flap; the right panel predicts steps since the previous flap. Bars are per-class F1 scores for distances 0, 1, 2, 3, and $4+$, where $4+$ combines all distances of at least four steps. DiT features have higher F1 in the 1-, 2-, and 3-step classes in both panels. Results use ordered five-frame input, one split with five training episodes and two validation episodes, and no confidence intervals. Class frequencies are imbalanced.

The diagnostic supplies a known delay and pauses the simulator for review. We use a fixed 100 ms action delay, rather than asking Astra to estimate the deployment's latency profile. Simulated time pauses during Astra API calls, so the reviewer's own latency does not advance the environment. This setting tests action review under a known delay; it does not evaluate the combined system with hardware-timed reviewer latency.

Astra describes timing risks, but we observe few successful recoveries. We test this interface on Intercept Grab Fast from MIKASA-Robo (Cherepanov et al., 2026a). We first use a checkpoint trained without latency. Only one rollout succeeds in more than 25 trials. We then use a checkpoint trained at the same fixed delay and examine selected policy failures. Astra's logged textual assessments track when each chunk will execute, relate earlier actions to observed outcomes, and sometimes predict that the robot will miss the moving ball. However, we observe very few successful recoveries. Figure 14 shows one such case: Astra selects a geometry-based candidate that slows the robot arm within the permitted action range, and the reviewed rollout grasps the ball while its paired baseline fails.

Turning timing assessments into effective control remains difficult. Astra's logged textual assessments correctly describe a known delay and its consequences, yet the reviewed policy does not achieve reliable recovery or a measurable average success-rate gain. A correct latency description is insufficient for latency-conditioned control: a policy benefits from being told the latency when it has been trained under that latency, whereas a reviewer that describes the delay correctly still rarely converts the description into successful actions. A real-time evaluation would also need to measure the reviewer's latency and account for it in action scheduling, using the timing framework of §3.

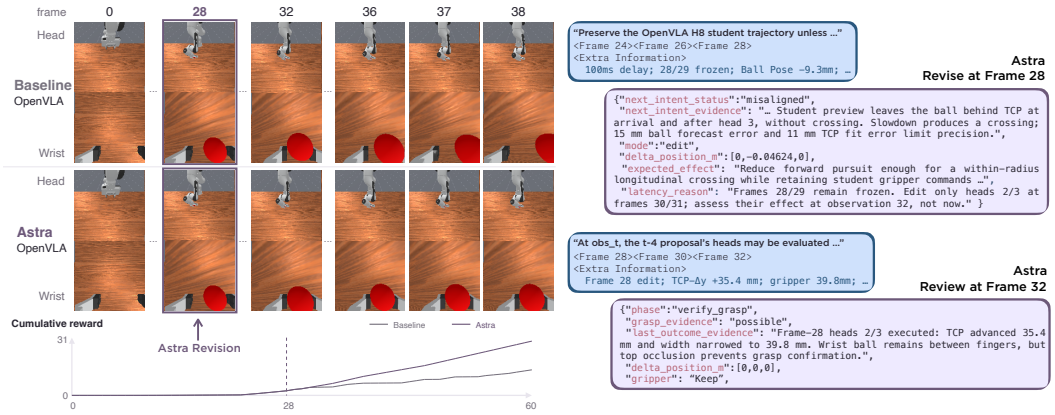

Figure 14: One selected MIKASA episode shows GPT-6 Astra choosing a correction after the unreviewed OpenVLA action misses the object. The left side compares head and wrist views from the baseline rollout with the reviewed rollout; the plotted value is reward accumulated up to each frame. OpenVLA predicts an action chunk, a geometry module proposes a slower correction, and Astra (OpenAI, 2026) selects whether to apply it given a stated 100 ms delay. The abridged interaction on the right shows the correction at frame 28 and review of the possible grasp at frame 32. This is a qualitative example, not an estimate of average success rate. Simulation pauses during the Astra API call, so the example does not test real-time intervention.

J TEACHER TRAINING AND EVALUATION

Each task in the 12-task benchmark uses a shared zero-latency teacher and, where applicable, separate teachers trained under the measured deployment profiles of OpenVLA-OFT, $\pi_{0.5}$, and GR00T. Balance and MIKASA InterceptGrabFast use the zero-latency teacher for action-chunk DAgger.

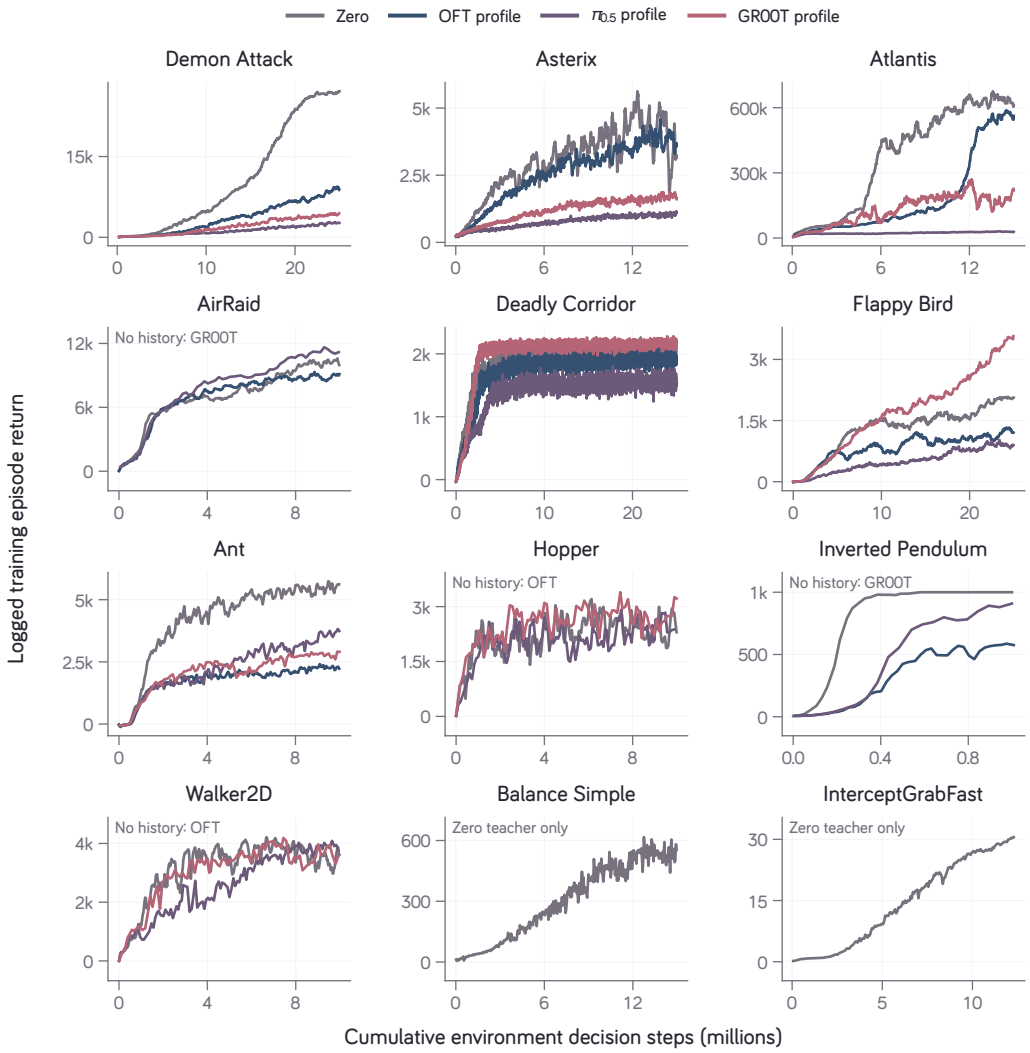

Figure 15: Learning curves for the task-specific RL teachers used to label VLA training data. The horizontal axis is the number of environment steps and the vertical axis is mean episode return. Each curve is one training run under either zero latency or the displayed deployment profile. Environment rates and episode limits differ across conditions, so differences in curve height should not be interpreted as the isolated effect of latency.

J.1 TEACHER EVALUATION RETURNS

Task	Zero → Zero	OpenVLA $P \to P$	$\pi_{0.5}$ $P \to P$	GR00T $P \to P$
Games				
Demon Attack	8,727.15 (100)	1,525.00 (100)	447.30 (100)	663.25 (100)
Asterix	6,041.00 (100)	3,941.00 (100)	1,141.00 (100)	1,852.50 (100)
Atlantis	88,613.00 (100)	38,411.00 (100)	27,780.00 (100)	29,064.00 (100)
AirRaid	2,557.50 (10)	3,242.50 (10)	3,785.00 (10)	2,572.50 (10)
Deadly Corridor	2,116.79 (100)	1,933.82 (100)	1,397.40 (100)	2,155.71 (100)
Flappy Bird	444.60 (100)	1,233.51 (100)	1,194.79 (100)	1,995.24 (100)
Embodied Control				
Ant	5,011.32 (10)	2,301.03 (10)	4,497.52 (10)	3,199.56 (10)
Hopper	2,682.09 (10)	3,657.61 (10)	2,616.17 (10)	3,814.84 (10)
Inverted Pendulum	1,000.00 (100)†	791.90 (10)	1,000.00 (10)	1,000.00 (10)
Walker2D	3,743.51 (10)	4,102.87 (10)	3,305.17 (10)	4,228.54 (10)
Balance Simple	866.62 (64)	N/A	N/A	N/A
InterceptGrabFast	36.82 (100)	N/A	N/A	N/A

Table 9: Evaluation returns of the RL teachers used to label the 12 benchmark tasks. Zero → Zero trains and evaluates without delay. $P \to P$ trains and evaluates with the measured profile for the architecture named in the column. Each cell reports mean episode return, followed by the number of evaluation episodes in parentheses. N/A marks Balance and MIKASA, which use the zero-latency teacher for action-chunk DAgger. Figure 15 shows the corresponding learning curves.

Games. Flappy, Demon Attack and Deadly Corridor teacher evaluations use 100 episodes with actual seeds 104771–104870. Zero caps are 3,600/14,400/30,000 frames; profile caps are 18,000/14,400/3,600, respectively. Profile teachers are the training-selected best checkpoints of 25M-decision-step, seed-42 runs; Demon profile evaluation uses the 16-way CuLE backend and Deadly uses native 35/8.75 Hz. Atlantis/Asterix zero teachers use 100 seeds 104771–104870 and a 14,400-frame cap, while profile teachers use 100 seeds 1000000–1000099, 60/15 Hz and a 7,200-frame cap; these are selected checkpoints of 15M-step, seed-0 runs.

AirRaid and Gymnasium control. Profile-trained teachers are evaluated over ten episodes. OFT uses 50/12.5 Hz for AirRaid and 10/10 Hz for control; GR00T uses 50/12.5 and 5/5 Hz; $\pi_{0.5}$ uses 10/2.5 and 5/5 Hz. Profile evaluations use IID latency sampling. Zero teachers are evaluated over ten episodes except Inverted Pendulum. † The Inverted Pendulum zero result uses 100 demonstration-collection rollouts, with all 100 attempted episodes retained. The teacher checkpoint is selected at 581,632 training steps; collection uses 25/25 Hz and a 1,000-frame cap.

Action-chunk teachers. Balance uses the H1 FastTD3 checkpoint at 12,801,280 environment steps, evaluated over 64 episodes with seed 715272, 50/50 Hz and a 1,000-frame cap. InterceptGrab-Fast uses the shared PPO teacher's 100-episode strict evaluation on seeds 4242424242–4242424341 at 20/20 Hz; all 100 episodes succeed.

K SCALE-UP VLA TRAINING LOSSES

Figure 16 shows the recorded training loss trajectories for the selected scale-up VLA runs across ten environments.

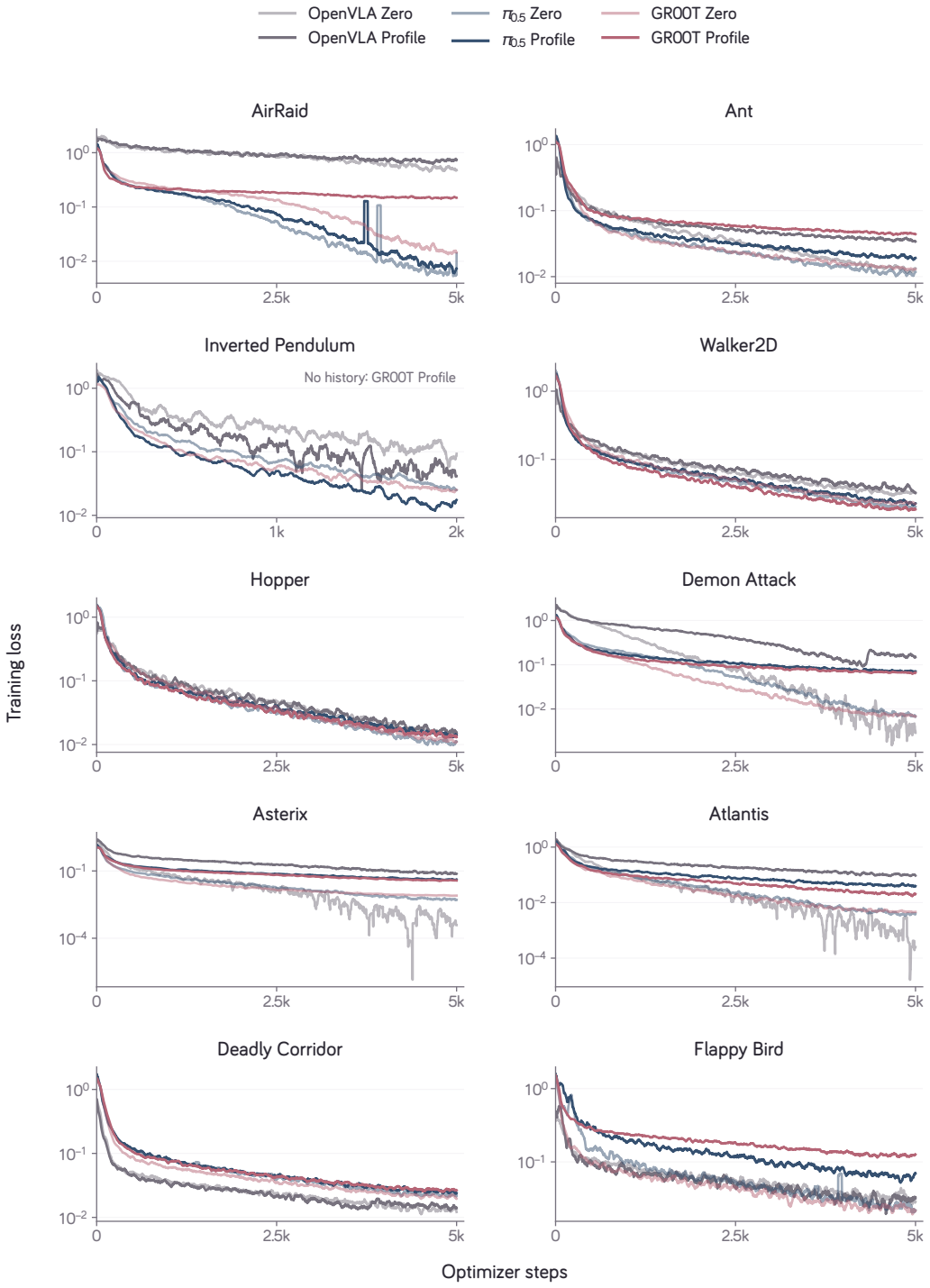

Figure 16: VLA supervised-training loss under zero latency and the measured deployment profile. Each panel corresponds to one task and model; each curve is the 50-point trailing mean from one training run. Loss values can be compared over training within a curve, but their absolute magnitudes are not directly comparable across tasks or model architectures.

L PER-ARCHITECTURE LATENCY TRAINING RESULTS

Figure 17 complements the radar chart in Figure 1 with per-architecture bars for the same tasks. It shows the 12 tasks with complete before/after results for all three architectures. Table 1 reports the raw mean returns for the same 12 tasks. Each before/after pair uses the same zero-latency policy's mean return as its reference; the after-training policy does not supply a new denominator. Appendix C documents the evaluation protocols.

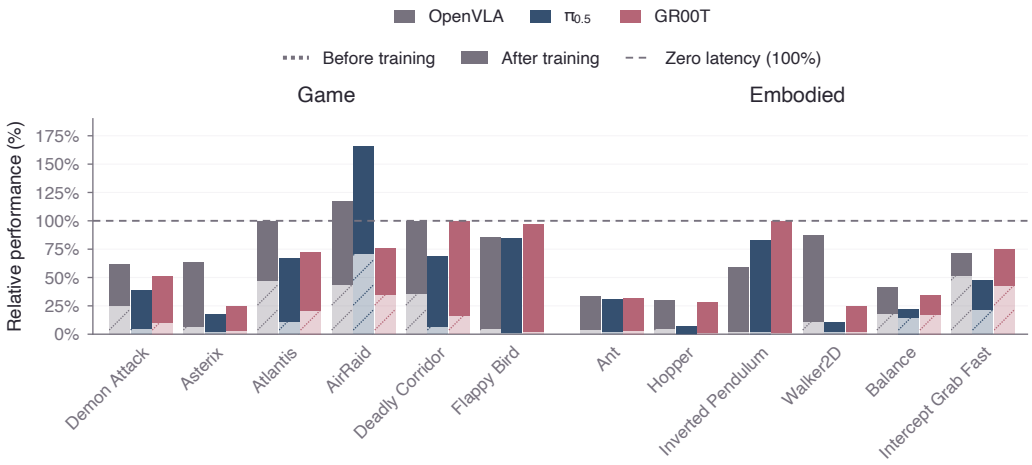

Figure 17: Training with the measured RTX 3090 profile improves real-time return for all 36 task and architecture pairs. For each bar, the top of the pale hatched region is the score before profile training and the top of the solid bar is the score after training; the two regions are not additive. Both scores are divided by the original policy's zero-latency mean on the same task. The dashed 100% line is that reference, so a value above 100% means that the profile-trained policy exceeds its original zero-latency mean. Table 1 reports the unnormalized values. MIKASA uses eight-action DAgger policies ($H = 8$); Balance uses chunks of $H = 8$ or $H = 16$. Bars show means without uncertainty intervals.

M MIKASA TRAINING AND LEARNING CURVES

M.1 ACTION-CHUNK POLICY TRAINING

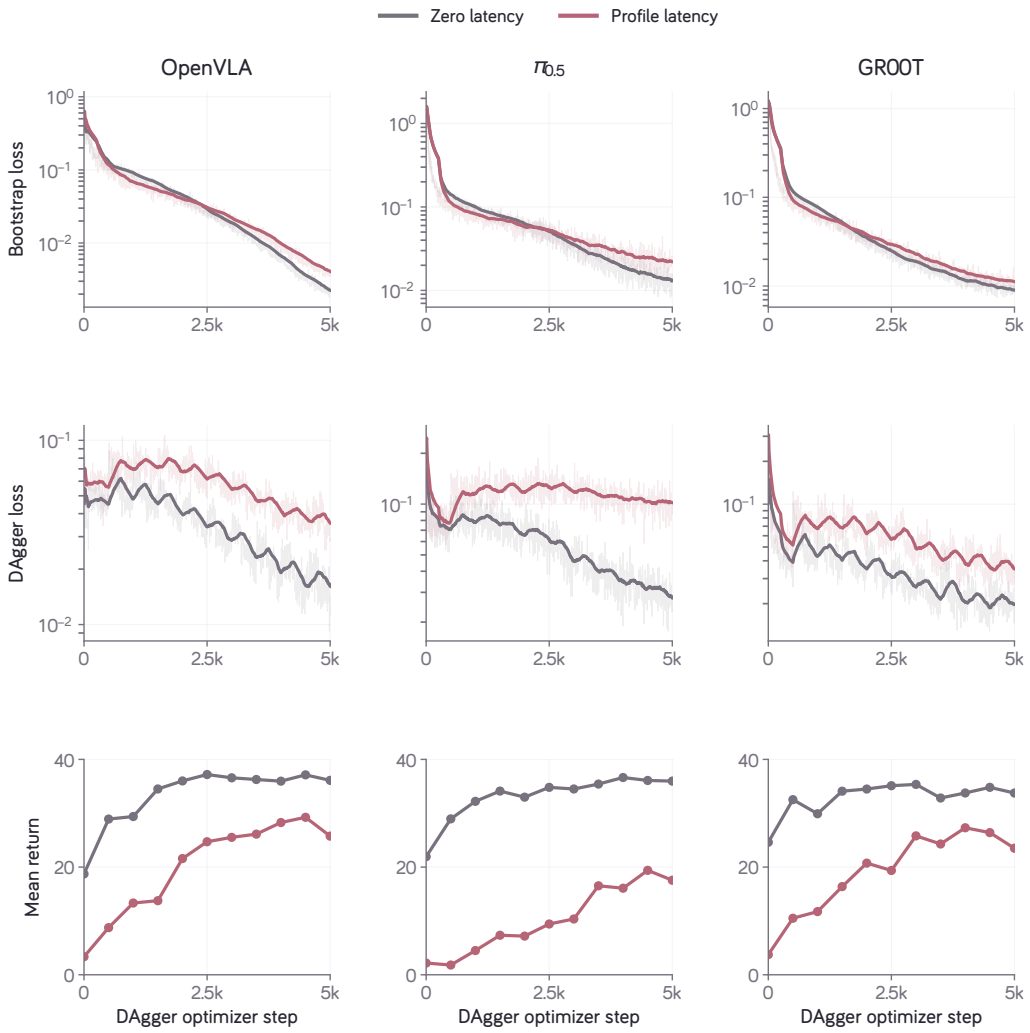

Figure 18: MIKASA eight-action DAgger learning curves under zero and profiled latency. Columns correspond to the three VLA architectures. The top and middle rows show the training losses displayed in each panel; the bottom row shows mean return during training. Each return point averages the same 100 development episodes, except steps 0 and 5,000, which use a separate set of 200 episodes. Each curve is one training run. The endpoint is the final checkpoint, while Figure 1 reports the best checkpoint.

M.2 SINGLE-ACTION POLICY TRAINING

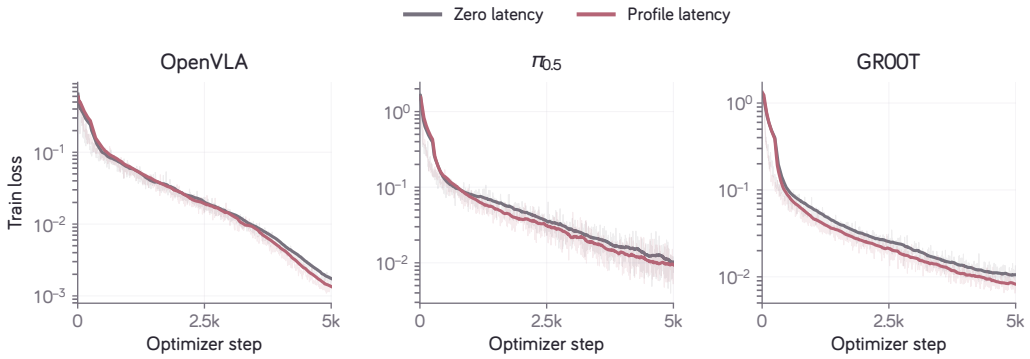

Figure 19: MIKASA single-action supervised-training loss under zero and profiled latency. Columns correspond to the three VLA architectures; gray and rose curves denote the two latency conditions. Each curve is one training run. Figure 2 compares these H1 SFT policies with H8 policies that also receive DAgger, so that comparison does not isolate the effect of action-chunk length.

M.3 LATENCY-ROBUST TRAINING

Figure 6 compares conditioned policies at 3,500 DAgger updates on 100 development episodes with a zero-trained policy at 5,000 updates on 200 held-out episodes. Input, action commitment, and evaluation budgets differ, so this is not a controlled delay ablation.

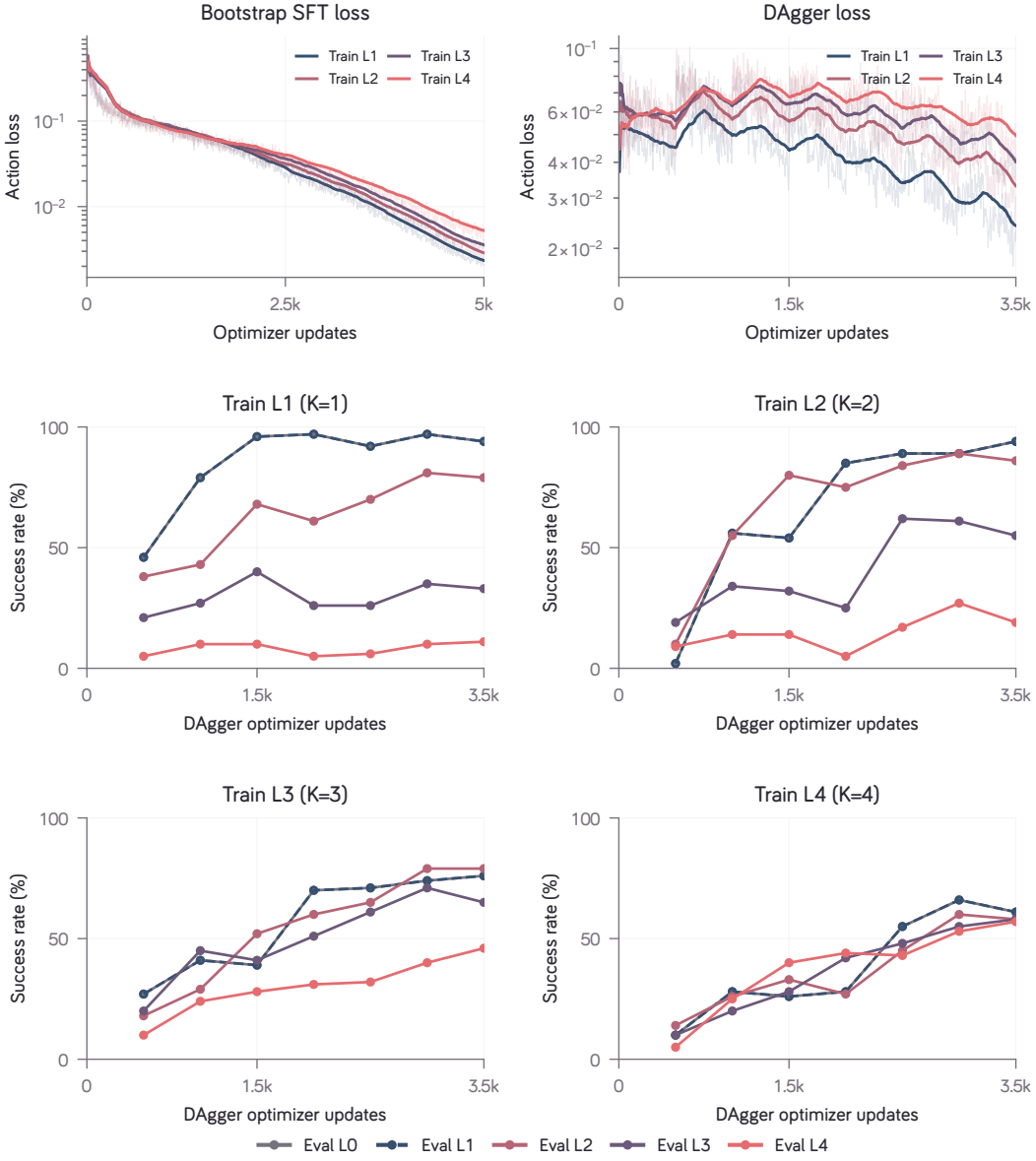

Figure 20: OpenVLA H8 learning curves for MIKASA policies trained at fixed delays of 1, 2, 3, or 4 frames. The top panels show 50-record trailing means of the supervised and DAgger losses on logarithmic axes. The bottom panels correspond to the four training delays; within each panel, line color denotes the evaluation delay. Each success-rate point averages the same 100 development episodes, beginning at update 500. The action prefix contains actions already queued for execution. Its length K equals the policy's training delay and remains fixed when the evaluation delay changes. Supervised training runs for 5,000 updates and DAgger for 3,500; each trajectory is one training run.

N HumanoidBench Balance Simple Bootstrap and DAgger Learning Curves

Figure 21 shows the SFT and DAgger training trajectories for the three action-chunk policies used in the Balance comparison.

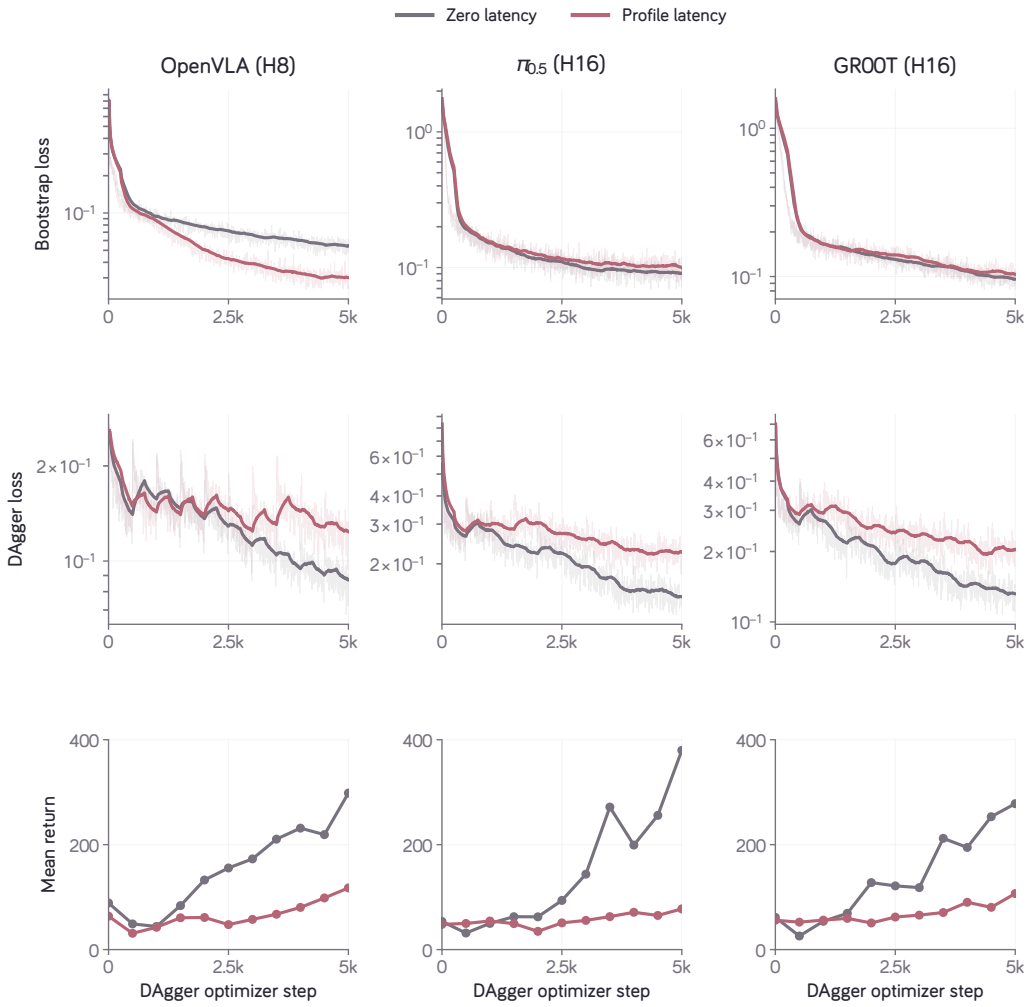

Figure 21: Balance learning curves for action-chunk policies trained with DAgger. Columns show OpenVLA with eight predicted actions per call ($H = 8$), and $\pi_{0.5}$ and GR00T with sixteen ($H = 16$). Gray and rose denote zero-latency training and training with the measured latency profile. The top rows show raw supervised and DAgger losses with 50-point trailing means on separate logarithmic axes; their absolute values are not comparable across architectures or training stages. The bottom row shows mean return on 32 fixed-seed development episodes, from the supervised endpoint at step 0 through 5,000 DAgger steps. All three architectures improve under both latency conditions, with larger increases in the zero-latency runs. Each curve is one training run.