

Fewer Steps, Better Actions: Rethinking Flow-Matching Inference for VLA Policies

Zhipeng Tang^{♣◇} Xinda Chen^{♣◇} Weining Rao^{♣◇} Xiao Li^{♣◇}
Wenting Tan[♣] Yuning Wang[△] Xiao Shi^{♣*} Xiaofang Zhao[♣]

[♣]Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China

[◇]University of Chinese Academy of Sciences, Beijing, China

[△]Zhongke Haichuan Intelligent

{tangzhipeng23s, chenxinda24z, shixiao}@ict.ac.cn

Abstract. Vision-language-action (VLA) policies based on flow matching generate action chunks through repeated evaluations of an action expert. Increasing the number of integration steps raises inference cost, but does not necessarily improve closed-loop success. We propose Coda, which reallocates part of this integration budget to a single learned endpoint correction. A frozen policy first completes a few-step noise-to-action trajectory; a lightweight Transformer then predicts a demonstration-supervised residual using the candidate action, source noise, and shared observation-prefix cache. Only the corrector is trained. On 50 RoboTwin Easy tasks, five-step Coda improves success from 71.64% to 74.68% over the matched five-step baseline, while reducing forward latency by 30.2% relative to the default ten-step policy. A two-step configuration achieves 71.88% success with a $2.12\times$ speedup. An independent 13-task control shows a 5.69-percentage-point gain at nearly equal latency, supporting correction as an effective alternative to additional integration. The same design also improves frozen official SmolVLA, raising two-step success from 60.8% to 69.4%. These results show that endpoint correction improves the quality-latency trade-off of frozen flow-matching policies.

1 Introduction

Vision-language-action (VLA) policies map images, language instructions, and robot states to actions, providing a unified interface for multitask manipulation [1–4]. Flow-matching policies such as π_0 , $\pi_{0.5}$, and SmolVLA first encode the observation condition and then repeatedly invoke an action expert (AE) to integrate noise into an action chunk [5–7]. Each step depends on the output of the previous one, so action-chunk generation requires multiple serial AE forwards. The number of function evaluations (NFE) directly drives the serial AE cost [8, 9]. In our A100 profiling, the AE stage accounts for about three quarters of the default ten-step forward pass. Reducing this repeated computation is therefore a major opportunity for lowering model-forward latency.

Do more integration steps lead to more successful actions? In our 50-task evaluation, increasing the number of integration steps from one to five raises success from 59.76% to 71.64%, whereas

*Corresponding author.

This work has been submitted to the IEEE for possible publication. Copyright may be transferred without notice, after which this version may no longer be accessible.

further increasing it to ten steps yields 71.52%. This plateau motivates our central question: *if the benefit of continued serial integration is limited, can part of its time and compute budget be reassigned to a single learned correction?*

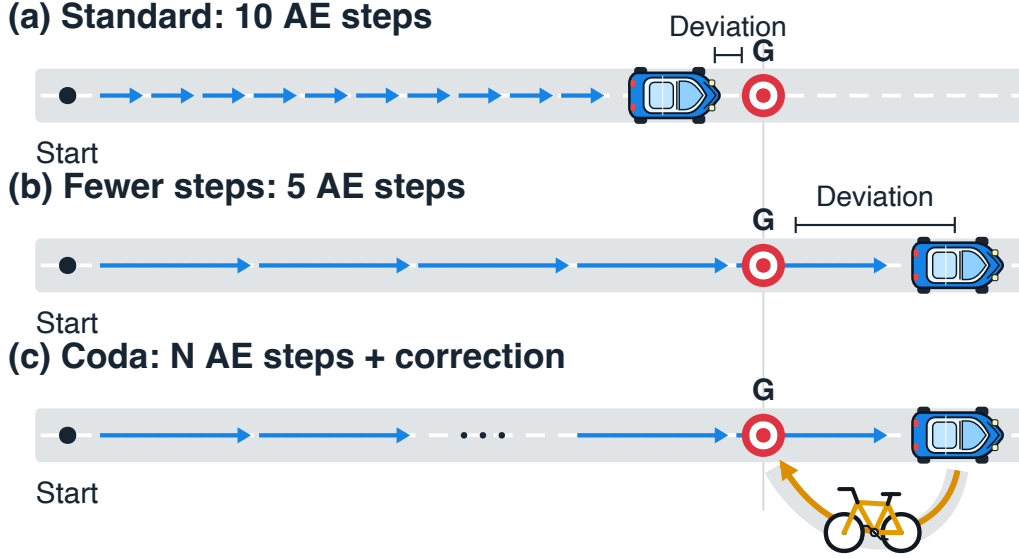


Figure 1: Compute allocation. Ten-step and few-step generation both cover the full noise-to-action interval. Coda reallocates part of the serial integration budget to one learned endpoint correction; the target and offsets are schematic.

Few-step generation and correction play complementary roles. Reducing ten Euler steps to five increases the step size from $1/10$ to $1/5$, advancing farther along the velocity field at each step. Both settings cover the complete noise-to-action interval. This coarser integration saves time but may change the final action. Coda uses the few-step candidate and spends part of the saved budget on one learned endpoint correction. Fig. 1 illustrates this change in computation using a car and a bicycle.

A preliminary endpoint intervention further motivates this design. On `hanging_mug`, mixing an expert-reference action with source noise at $t = 0.1$ and applying one AE step succeeds in 9/10 trials, compared with 5/50 for the five-step baseline (Fig. 6a). This privileged probe suggests that endpoint information can improve the final action. Coda learns such a correction from demonstrations, so deployment requires only the policy’s own candidate and observation.

We propose Coda, which separates action generation from endpoint correction. Named after the concluding passage of a musical composition, Coda adds a single learned correction after the complete few-step generation trajectory. The original policy uses N integration steps to generate a candidate a_θ , and an independent Transformer predicts a residual $\widehat{\Delta a}$. The executed action is

$$a_{\text{exec}} = a_\theta + \widehat{\Delta a}. \quad (1)$$

The corrector is trained to predict the difference between the demonstrated action and the candidate produced by the frozen generator. During inference, it reuses the key-value (KV) cache from the observation encoding, which limits the additional computation [10]. Only the corrector is trained; the generator remains frozen.

The corrector conditions on the candidate action, observation context, and source noise. Its single forward pass is separate from integration, so the generation step count can be chosen to meet the available inference budget.

We evaluate both closed-loop success and measured inference latency. On the 50-task main panel, five-step Coda gains 3.04 percentage points over five-step generation and is $1.43\times$ faster than the default ten-step policy. In an independent 13-task panel, five-step Coda outperforms six-step generation by 5.69 points at nearly identical latency. A second implementation on frozen official SmolVLA tests whether this design extends beyond $\pi_{0.5}$.

Our contributions are:

1. **A compute-allocation strategy for few-step generation.** We show that higher NFE substantially increases serial AE computation but provides limited additional closed-loop benefit in the evaluated setting. Based on this observation, Coda replaces part of the integration budget with a single endpoint correction.
2. **A configurable corrector for frozen flow-matching policies.** Coda keeps the original policy frozen, reuses its observation-prefix representation, and learns the residual between the generated candidate and the demonstrated action. A single corrector can be applied across test-time NFE values, and the same design is implemented on both $\pi_{0.5}$ and SmolVLA.
3. **An evaluation of both action quality and inference cost.** On the 50-task RoboTwin suite, Coda improves mean success over matched-NFE baselines, while selected few-step configurations reduce latency relative to the default ten-step policy. Cross-NFE experiments, transfer to SmolVLA, and comparisons with Mix-AE, continued AE training, and an MLP corrector characterize how endpoint correction uses the available inference budget.

2 Related Work

Generative action policies and deployment latency Diffusion models, score-based SDEs, and flow matching model continuous action distributions through iterative transformations [8, 9, 11–13]. Diffusion Policy, 3D Diffusion Policy, ACT, Octo, and RDT-1B combine generative action modeling with action-chunk execution for visuomotor control, bimanual manipulation, and multitask policies [14–18]. RT-1, RT-2, and OpenVLA instead use autoregressive or discrete-action VLA interfaces, while Open X-Embodiment provides cross-embodiment training data [1–4]. π_0 , $\pi_{0.5}$, and SmolVLA combine vision-language conditioning with flow-matching action experts and generate action chunks through repeated action-expert (AE) evaluations; $\pi_{0.5}$ uses a PaliGemma-based vision-language prefix [5–7, 19]. Their planning latency therefore includes both observation-prefix encoding and NFE-proportional AE computation. Real-Time Execution of Action Chunking Flow Policies mitigates chunk-arrival delay through asynchronous generation, overlapping execution, and chunk alignment, while retaining the underlying iterative action generator [20]. Coda addresses a different layer: how to allocate inference compute between candidate generation and endpoint correction.

Few-step generation and sampling acceleration For diffusion sampling, DDIM uses a non-Markovian sampling process and DPM-Solver uses a dedicated high-order ODE solver to reduce sampling steps [21, 22]; progressive distillation, consistency models, latent consistency models,

and shortcut models support few- or one-step generation through distillation or new training objectives [23–26]. In visuomotor control, Consistency Policy applies consistency distillation to few-step inference [27]. In flow matching and robotic action generation, Rectified Flow improves few-step transport by straightening transport trajectories, while L1 Sample Flow combines an integration step with direct sample prediction under an L_1 objective [9, 28]. These approaches primarily improve the solver, alter the generation trajectory, or retrain a few-step generation objective.

Generation sources, residual policies, and post-generation correction Action-to-action flow matching encodes proprioceptive and action history into the generation source, changing the action distribution from its starting point [29]; ResVLA anchors generation at a predicted low-frequency intent and models the remaining high-frequency dynamics with a residual diffusion bridge [30]. Classical residual policy learning adds a learned action correction outside a fixed controller [31, 32], a form also related to deep residual connections [33].

Position of this work Coda applies supervised residual learning to a frozen flow-matching VLA at a specific interface: the endpoint of a complete few-step trajectory. It reuses the prefix KV cache, conditions on the generated candidate, and decouples the integration budget from the correction forward. Our focus is the resulting compute allocation: whether a learned endpoint update delivers better closed-loop quality per unit inference time than additional calls to the original action expert.

Coda inference dataflow

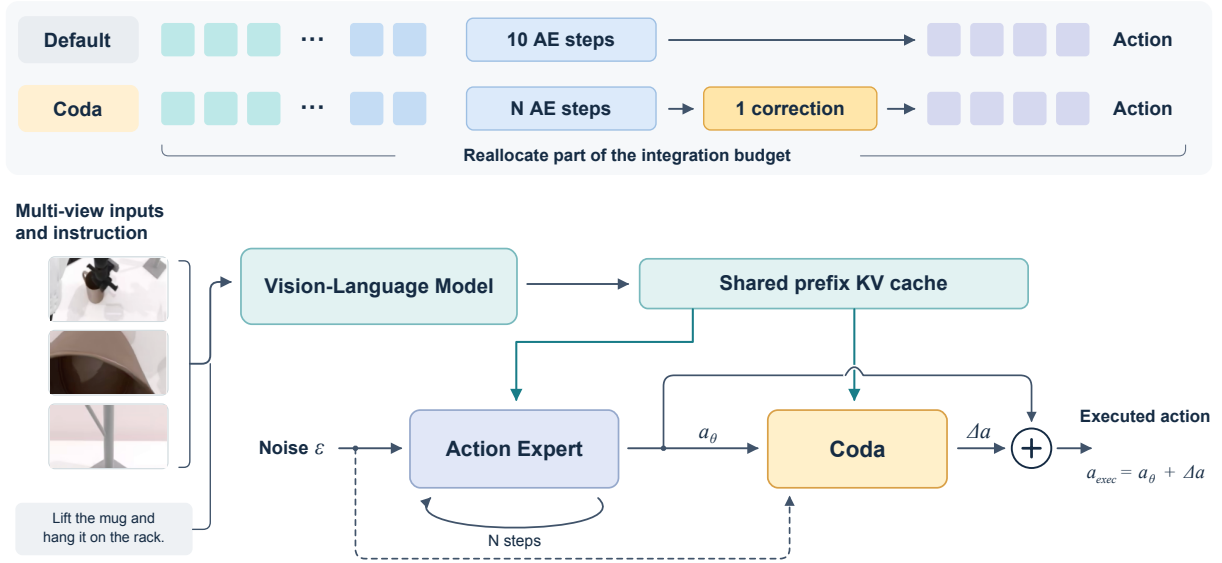


Figure 2: Coda inference. A frozen VLM encodes the three camera views and instruction once; in $\pi_{0.5}$, discretized robot state is also included in the text prefix. Both modules reuse the prefix KV cache. The frozen AE completes N Euler steps, and Coda uses the candidate and the same source noise to predict a residual in one forward. The residual is added to the candidate before execution. A line bridge denotes a crossing without a connection. Only Coda is trainable.

3 Method

3.1 Few-step generation on frozen $\pi_{0.5}$

Let o denote the observation and instruction, and let $a^* \in \mathbb{R}^{H \times D}$ be an expert action chunk, where H is the prediction horizon and D is the action dimension. The source noise is $\varepsilon \sim \mathcal{N}(0, I)$. We use the convention that $t = 1$ is noise and $t = 0$ is action. The flow-matching training interpolation is

$$x_t = t\varepsilon + (1 - t)a^*, \quad (2)$$

with target velocity $\varepsilon - a^*$ [8, 9, 13]. Let $z_o = f_{\text{VLM}}(o)$ be the frozen observation representation. Inference uses N Euler steps:

$$x^{[i+1]} = x^{[i]} - \frac{1}{N} v_\theta(x^{[i]}, 1 - i/N, z_o), \quad (3)$$

where $i = 0, \dots, N - 1$, $x^{[0]} = \varepsilon$, and the final candidate is $a_\theta = x^{[N]}$. Reducing N increases the integration step size. Coda and its matched-NFE baseline use the same integration process. Fig. 2 shows this N -step path to a_θ and the residual suffix attached after the candidate.

3.2 Candidate-conditioned residual learning

The residual module conditions on the frozen observation prefix, the candidate action, and its source noise:

$$\widehat{\Delta a} = r_\phi(z_o, a_\theta, \varepsilon, t_{\text{res}}). \quad (4)$$

On $\pi_{0.5}$, the candidate action and noise are projected separately and added to form the input to a Transformer [34] suffix. As shown in Fig. 2, the residual suffix reads the VLM prefix KV cache, while the prefix does not read the residual suffix. We set $t_{\text{res}} = 0.1$ as fixed time conditioning through AdaRMS modulation, matching the near-endpoint time used in GT-mix. This fixed conditioning value is used by the corrector after generation reaches $t = 0$. SmolVLA omits this time conditioning. The residual Transformer suffix is initialized from the corresponding frozen AE weights, and its output layer is zero-initialized so that the initial correction is zero and training starts from $a_{\text{exec}} = a_\theta$.

Training updates only residual parameters ϕ ; the VLM and AE remain frozen throughout. With sg denoting stop-gradient, the supervision target and loss are

$$\Delta a^* = a^* - \text{sg}(a_\theta), \quad (5)$$

$$\mathcal{L}_{\text{res}} = \|r_\phi(z_o, \text{sg}(a_\theta), \varepsilon, t_{\text{res}}) - \Delta a^*\|_1. \quad (6)$$

This objective learns the total displacement from the generated candidate to the demonstrated action, including both policy bias and the effect of coarse integration. The $\pi_{0.5}$ residual is trained on candidates with $N_{\text{train}} = 5$ and the same weights are reused at different test NFE values.

For SmolVLA, we retain the same candidate-plus-residual form and train an independent corrector on the frozen SmolVLA policy. This implementation uses $N_{\text{train}} = 10$, omits the time conditioning, and applies causal attention to residual tokens. Residual weights are not shared across policies.

3.3 Direct execution and inference cost

At inference, Coda outputs the corrected action according to (1), executes the first K actions of the predicted chunk, and then replans from a new observation. The executed horizon is K ; evaluation uses $K = 10$ for $\pi_{0.5}$ and $K = 50$ for SmolVLA (Table VII). Each replan encodes the observation prefix once and links generation and correction with the same source noise. Expert actions are used only as training supervision.

If prefix, one AE forward, and one residual-forward time are T_{prefix} , T_{AE} , and T_{res} , respectively, then

$$T_{\pi_{0.5}\text{-FT}}(N) \approx T_{\text{prefix}} + NT_{\text{AE}}, \quad (7)$$

$$T_{\text{Coda}}(N) \approx T_{\text{prefix}} + NT_{\text{AE}} + T_{\text{res}}. \quad (8)$$

Thus the net saving from ten-step generation to five-step Coda is approximately $5T_{\text{AE}} - T_{\text{res}}$. The speedup comes from reducing repeated AE calls; matched-NFE quality improvement incurs one correction forward. Below, NFE counts AE calls only, and Coda’s single residual forward is reported separately.

4 Experiments

4.1 Experimental setup

Main evaluation. We evaluate on 50 RoboTwin bimanual manipulation tasks [35, 36]. The base $\pi_{0.5}$ policy is fine-tuned for 60k steps on demo_clean_50 and then frozen ($\pi_{0.5}\text{-FT}$). Coda is trained for 3k steps on this checkpoint. Global batch sizes are 48 and 960, respectively, giving 2.88M presented samples for each stage. Peak AdamW learning rates are 2.5×10^{-5} and 5×10^{-5} , with 3000 and 150 warmup steps and cosine decay. Execution uses the first $K = 10$ actions before replanning.

For each NFE in $\{1, 2, 3, 4, 5, 10\}$, both methods complete 50 valid episodes on each task (2500 episodes per method). Additional stable seeds fill the six tasks whose default initialization pools contain fewer than 50 valid episodes. Success is the equally weighted task mean. We obtain the 95% interval for the five-step gain by resampling paired task differences 100000 times [37]. The source-noise stream is determined by the environment seed and replan index. Complete execution and environment settings appear in the appendix.

Additional evaluations. Official SmolVLA remains frozen, and a separate corrector is trained for 18716 steps on its official 10-task set, using batch size 256 and peak learning rate 10^{-4} . Evaluation uses 50 episodes per task and execution horizon $K = 50$. The no- ϵ input variant uses checkpoint 18000. An independent 13-task panel contains 621 valid episodes per method and compares Transformer correction, MLP correction, and additional AE integration at nearby inference budgets.

Latency. The main operating points use an A100 80GB PCIe, batch size one, SDPA, and no compilation. We report the median of 40 CUDA-synchronized model forwards after 10 warmups, including prefix encoding, all AE calls, and correction. These timings exclude environment, communication, and robot execution. All $\pi_{0.5}$ main points come from the same measured NFE sweep. The independent 13-task panel uses an A100 40GB and its own observation cache and timing protocol.

Table I: Success on 50 Easy tasks (%). Each method has 50 episodes per task at every NFE. Bold indicates the higher mean at each NFE.

AE-NFE	1	2	3	4	5	10
$\pi_{0.5}$ -FT	59.76	67.40	71.28	69.96	71.64	71.52
Coda	69.80	71.88	73.64	73.12	74.68	72.60

4.2 More integration has diminishing closed-loop returns

The frozen baseline improves from 59.76% at one step to 71.64% at five steps, but reaches 71.52% at ten steps (Table I). Over the latter interval, measured latency rises from 144.7 to 233.0 ms, a 61.0% increase (Fig. 3). Thus, increasing integration resolution beyond five steps adds substantial serial computation without improving mean success. This observation motivates using part of that computation for an endpoint update learned directly from demonstration actions.

Five-step Coda reaches 74.68%, a 3.04-percentage-point improvement over five-step generation, with a task-bootstrap interval of [1.04, 5.08]. Across tasks, 27 improve, 10 tie, and 13 decline. The same corrector yields positive mean gains at every tested NFE, from 10.04 points at one step to 1.08 points at ten steps. Five steps give the highest success among the tested Coda configurations.

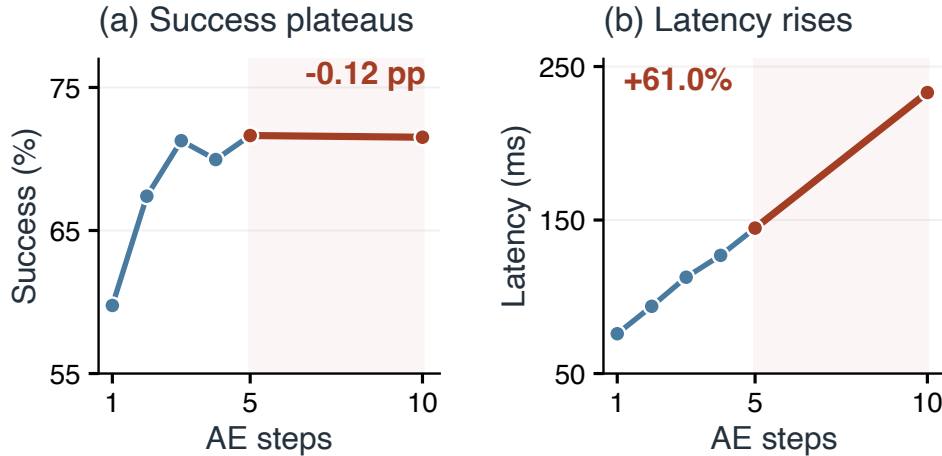


Figure 3: Diminishing returns of frozen $\pi_{0.5}$ on the 50-task panel. (a) Success versus AE steps. (b) Measured forward latency versus AE steps. The highlighted 5-to-10-step segment changes success by -0.12 percentage points while increasing latency by 61.0%.

4.3 Correction is a better use of comparable inference compute

The independent 13-task panel compares correction with additional integration at nearly equal latency. Six-step $\pi_{0.5}$ -FT reaches 71.69% at 218.14 ms. Five-step Coda reaches 77.38% at 218.59 ms: a 5.69-point improvement at nearly equal latency (Fig. 4). The paired 95% interval is [2.62, 8.77] points when resampling episodes within fixed tasks and [1.54, 10.15] under two-stage task-and-episode resampling. This comparison supports allocating the next unit of computation to endpoint correction rather than another AE step.

The same Coda point also exceeds ten-step generation by 4.37 points while using 31.5% less forward time. An MLP corrector reaches 73.63% at 195.63 ms, providing a cheaper intermediate option. The MLP and Transformer correctors offer different quality–latency choices at the same integration step count.

Table II: Independent 13-task compute-allocation panel. Each method has 621 valid episodes; timing uses an A100 40GB.

Method	AE-NFE	Success (%)	Latency (ms)
$\pi_{0.5}$ -FT	5	70.40	193.39
$\pi_{0.5}$ -FT	6	71.69	218.14
$\pi_{0.5}$ -FT	10	73.02	318.98
Coda-MLP	5	73.63	195.63
Coda-Transformer	5	77.38	218.59

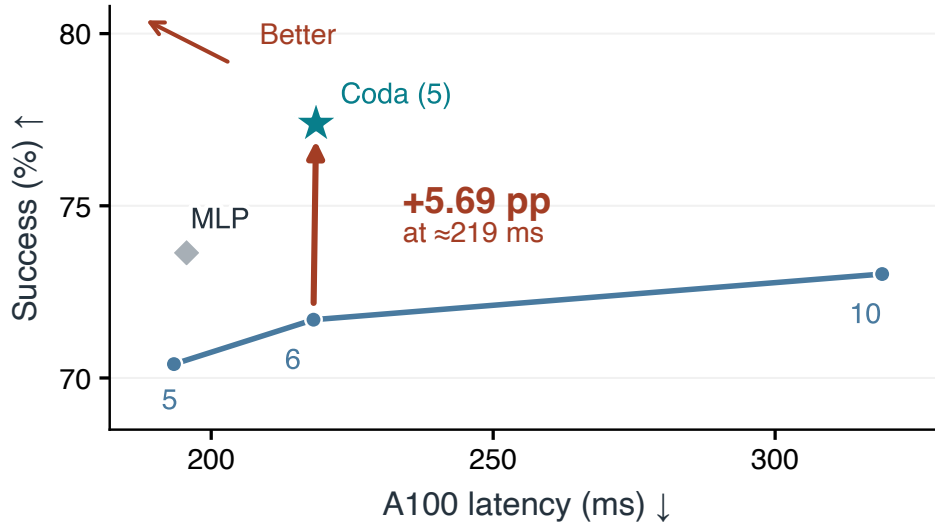


Figure 4: Where to spend the inference budget. The baseline line connects measured 5-, 6-, and 10-step policies. At approximately 219 ms, Coda improves success by 5.69 points over six-step generation. Upper-left is better; this panel uses the independent 13-task protocol in Table II.

4.4 Coda improves the deployment trade-off

Fig. 5 maps the main evaluation to measured success–latency operating points. Each line connects measured configurations of one method; step counts identify the deployment budget.

For $\pi_{0.5}$, five-step Coda reaches 74.68% at 162.6 ms, compared with 71.52% at 233.0 ms for the default ten-step policy. It therefore improves success by 3.16 points while reducing latency by 30.2% ($1.43\times$ speedup). Two-step Coda offers a lower-cost point: 71.88% at 109.9 ms, a $2.12\times$ speedup over the default. It also exceeds the success of three-step generation (71.28% at 112.7 ms) at slightly lower latency.

Table III separates the cost of generation and correction. At five steps, Coda adds 17.9 ms to the 144.7 ms baseline. Reducing AE calls from ten to five more than covers that addition. Prefix

encoding takes approximately 56 ms and correction approximately 18 ms, so these two costs become a larger fraction of the total at low NFE. The resulting operating points support choosing a two-step configuration for lower latency or a five-step configuration for higher success, using the same trained corrector.

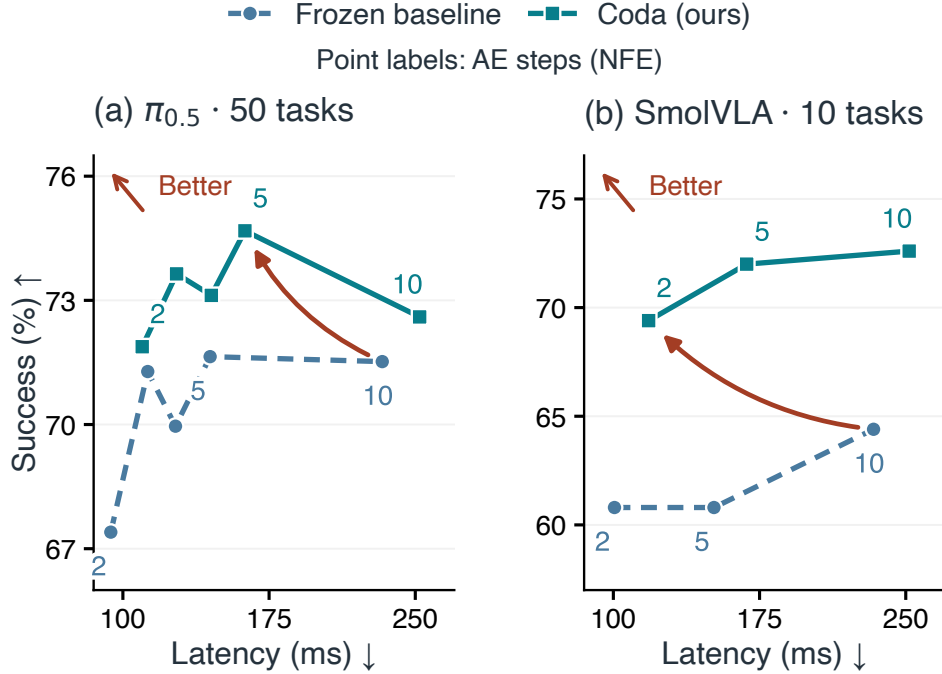


Figure 5: Deployment landscapes on A100 80GB: (a) $\pi_{0.5}$, 50 tasks; (b) frozen official SmolVLA, 10 tasks. Upper-left is better. Point labels (2, 5, 10) indicate the number of AE integration steps (NFE); Coda adds one correction forward at each setting. Lines connect measured points at $\text{NFE} \geq 2$, and arrows highlight improvements over the default ten-step policy. One-step success is retained in Tables I and IV. The panels use separate host policies and evaluation protocols.

Table III: Measured median forward latency (ms). P/A/R denote prefix, AE, and residual stage times. Speedup is relative to the same-GPU ten-step baseline. Bold highlights Coda deployment points.

GPU	Method	N	P	A	R	Total	Speedup
A100	$\pi_{0.5}$ -FT	10	55.8	175.6	0.0	233.0	1.00×
	$\pi_{0.5}$ -FT	5	55.8	88.0	0.0	144.7	1.61×
	Coda	5	55.7	88.0	17.8	162.6	1.43×
	$\pi_{0.5}$ -FT	3	57.1	54.3	0.0	112.7	2.07×
	Coda	2	55.9	35.3	17.9	109.9	2.12×
4090	$\pi_{0.5}$ -FT	10	62.7	182.1	0.0	246.5	1.00×
	Coda	5	61.7	88.6	17.8	169.2	1.46×
	Coda	2	61.9	35.6	17.9	116.2	2.12×

Stage times and total latency are measured separately; medians are not additive.

4.5 The design transfers to frozen SmolVLA

On official SmolVLA, Coda raises success from 60.8% to 69.4% at two steps, from 60.8% to 72.0% at five steps, and from 64.4% to 72.6% at ten steps (Table IV). The two-step corrected policy runs at 118.1 ms, compared with 233.5 ms for ten-step official generation: a $1.98\times$ speedup with a 5.0-point gain. Fig. 5b shows the same quality–latency improvement on SmolVLA.

Table IV: SmolVLA success on 10 Easy tasks (%). Each configuration completes 500 episodes. AE-cont and Coda (+ ε) use checkpoint 18716; the no- ε variant uses checkpoint 18000.

AE-NFE	1	2	5	10
Official frozen	35.0	60.8	60.8	64.4
AE-cont	48.2	64.6	70.8	69.2
Coda (+ ε)	55.8	69.4	72.0	72.6
Coda (no ε)	58.2	71.4	75.2	74.8

The SmolVLA corrector is trained on ten-step candidates and directly reused at one, two, and five steps. The $\pi_{0.5}$ corrector is trained at five steps and reused over the full tested range. This cross-NFE reuse allows deployment budgets to change without training a new corrector for each setting.

AE continuation is an alternative use of additional training compute. Under the matched SmolVLA training budget, continuation reaches 64.6/70.8/69.2% at two/five/ten steps, below the corresponding Coda results. The independently trained no- ε configuration reaches 58.2/71.4/75.2/74.8%, indicating that source-noise input is unnecessary for this SmolVLA configuration. The deployment plot uses the noise-conditioned model consistently across all budgets.

4.6 Direct execution preserves the learned correction

We compare direct execution with Mix-AE, which sends the corrected action through one additional frozen AE update. It first constructs

$$\tilde{x}_{0.1} = 0.1\varepsilon + 0.9(a_\theta + \widehat{\Delta a}), \quad (9)$$

then integrates from $t = 0.1$ to zero using the original velocity field. The same source noise is reused in this remapping.

On 128 held-out mix50 demonstration frames, mean absolute error over 12 arm joints is 5.73×10^{-3} for five-step generation, 5.64×10^{-3} for ten-step generation, 5.54×10^{-3} for Mix-AE, and 4.31×10^{-3} for Coda. Direct execution reduces error by 24.8% relative to the five-step baseline (Fig. 6b). The extra AE update moves the result back toward baseline error, supporting the choice to execute the supervised displacement directly.

The endpoint probe in Fig. 6a complements this comparison. On hanging_mug, five-step generation succeeds in 5/50 trials, Coda in 12/50, and GT-mix in 9/10. GT-mix uses a test-time expert reference and serves as a privileged diagnostic; Coda learns its correction from training demonstrations.

4.7 Response to visual randomization

On the 50-task Hard protocol (demo_randomized_50, minimal shader), both methods retain $K = 10$. Across 2088 valid episodes per method, with 23–49 episodes per task, five-step generation and Coda

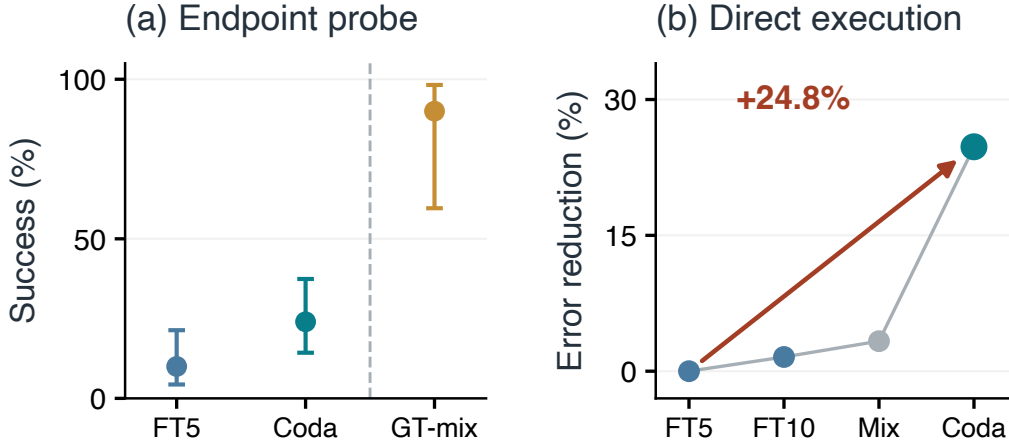


Figure 6: Endpoint diagnostics. (a) Hanging-mug success with 95% Wilson intervals; the dashed divider separates the privileged GT-mix probe from deployable policies. (b) Reduction in mean open-loop action error relative to FT5; Mix denotes Mix-AE. Direct Coda execution preserves the largest reduction.

reach 24.42% and 23.86%. The paired difference is -0.56 points with a 95% interval of $[-2.71, 1.64]$. The Easy-task benefit does not yield a clear gain under this visual randomization.

5 Discussion And Limitations

The near-equal-latency comparison shows that endpoint correction can improve closed-loop success more than an additional integration step. Results on two frozen policies support allocating inference compute between generation and correction.

The residual target combines policy error and the effects of coarse integration. Separating their contributions requires additional controls. The architecture and input comparisons also vary in capacity, training run, or checkpoint choice. In the historical $\pi_{0.5}$ continuation run, the recorded 2398 episodes give 75.62% task-mean success, but two tasks use $K = 16$ rather than the main protocol’s $K = 10$. This protocol difference prevents a matched comparison with Coda.

Gains vary across tasks, and the tested visual randomization yields no clear average improvement. Future work should test whether pre-execution signals can predict when correction helps and validate the measured forward-time gains in a real-robot control loop. The present evaluation is in simulation, and model-forward latency excludes communication and physical execution.

6 Conclusion

Coda reallocates part of a flow-matching policy’s serial integration budget to one learned endpoint correction. With the VLM and action expert frozen, it improves few-step action quality and offers faster, more successful deployment points than the default ten-step policy. The near-equal-latency comparison supports this allocation of compute, and each trained corrector can serve multiple integration budgets.

A Implementation Details

The appendix provides the inference procedure (Table V), training hyperparameters (Table VI), and the training and evaluation environments (Table VII).

A.1 Inference procedure

At test time the VLM and AE are frozen. AE-NFE N counts Euler steps. Coda at NFE = N runs the same N steps as $\pi_{0.5}$ -FT, then one residual forward.

Table V: Inference at AE-NFE N . $\pi_{0.5}$ -FT stops at a_θ ; Coda adds one residual; Mix-AE is ablation only.

Input: observation o , AE-NFE N , recipe $\in \{\pi_{0.5}\text{-FT}, \text{Coda}, \text{Mix-AE}\}$.
Frozen: f_{VLM}, v_θ . On $\pi_{0.5}$, Coda uses residual r_ϕ at $t_{\text{res}} = 0.1$; SmolVLA does not use that time conditioning.

1. Encode o once: $z_o = f_{\text{VLM}}(o)$. Cache prefix key-value tensors.
2. Draw $\varepsilon \sim \mathcal{N}(0, I)$ (use the host policy’s evaluation noise stream). Set $x^{[0]} = \varepsilon$.
3. For $i = 0, \dots, N - 1$: $x^{[i+1]} = x^{[i]} - \frac{1}{N} v_\theta(x^{[i]}, t_i, z_o)$, $t_i = 1 - i/N$.
4. Set $a_\theta = x^{[N]}$.
5. $\pi_{0.5}\text{-FT}$: execute a_θ and stop.
6. **Coda**: $\widehat{\Delta a} = r_\phi(z_o, a_\theta, \varepsilon, t_{\text{res}})$, execute $a_{\text{exec}} = a_\theta + \widehat{\Delta a}$.
7. **Mix-AE** (ablation): form $\tilde{x}_{0.1} = 0.1 \varepsilon + 0.9(a_\theta + \widehat{\Delta a})$ with the same ε as step 2, then let the frozen AE integrate from $t=0.1$ to 0.

Joint commands: linear interpolation, stride 16. $\pi_{0.5}$ eval pairs $\pi_{0.5}\text{-FT}$ /Coda on the same ε ; the stream is indexed by environment seed and replan step. Early failure termination and evaluation video recording are disabled.

A.2 Training hyperparameters

Table VI lists the training hyperparameters used in the main experiments. We use AdamW [38] with cosine learning-rate decay [39]. The two main training stages each present 2.88M samples. The historical continuation run matches residual training GPU-hours.

For $\pi_{0.5}$, training candidates are generated by the frozen policy with $N_{\text{train}} = 5$ Euler steps. On SmolVLA, AE-cont and Coda (+ ε) are independently trained under a matched budget (per-GPU batch 64). The same corrector is evaluated at $\text{NFE} \in \{1, 2, 5, 10\}$.

Table VI: Training hyperparameters. $\pi_{0.5}$ -FT / Coda / Cont use the 50-task RoboTwin clean50 mixture. SmolVLA residual uses the official 10-task set; official SmolVLA stays frozen.

	$\pi_{0.5}$ -FT	Coda	Cont	SmolRes
Init	$\pi_{0.5}$ FT	FT@60k	FT@60k	official
Trainable	VLM+AE	residual only	AE	residual only
Global batch	48	960	48	256
Steps	60k	3k	21k	4 ep
Frames	2.88M	2.88M	1.01M	4 epochs
Peak lr	2.5×10^{-5}	5×10^{-5}	2.5×10^{-5}	1×10^{-4}
Warmup	3k	150	1,050	100
Floor lr	2.5×10^{-6}	5×10^{-6}	2.5×10^{-6}	2.5×10^{-6}
Loss	FM (AE)	$L_1(\Delta a, a^* - \text{sg}(a_\theta))$	FM (AE)	L_1 residual
$N_{\text{train}} / t_{\text{res}}$	–	5 / 0.1	–	10 / unused
Hardware	4×A100	4×A100	4×A100	4×4090

A.3 Training and evaluation environments

Table VII: Environments for every closed-loop and latency number in the main text. Easy and Hard use $\pi_{0.5}$; Smol Easy uses official SmolVLA.

	Easy	Hard	Smol Easy
Suite	RoboTwin 2.0	RoboTwin 2.0	RoboTwin 2.0
Split	demo_clean_50	demo_randomized_50	demo_clean_50
Tasks	50	50	official 10
Shader	default (rt32)	minimal	default
Target n	50	mid-20s–49	50
Seeds	100000+ i	100000+ i	100000+ i
K	10 (all tasks)	10 (all tasks)	50
Exec	lerp / stride 16	lerp / stride 16	30 Hz 480×640
FM ε	det. (env, step)	det. (env, step)	i.i.d.
Cameras	RoboTwin default	RoboTwin default	official four
Metric	unweighted task mean	unweighted task mean	unweighted task mean

Default YAML pools in 100000–100049 are smaller on six tasks: adjust_bottle 35, dump_bin_binbin 32, pick_diverse_bottles 23, place_dual_shoes 46, both shake 31. Extra stable seeds fill each to $n=50$. Unstable Hard resets are skipped. On $\pi_{0.5}$, each replan samples $\varepsilon \sim \mathcal{N}(0, I)$ from seed $\text{env} \times 10007 + \text{step}$; training is unseeded. SmolVLA eval is i.i.d. Latency: batch 1, no compile, SDPA, warmup 10 + 40 CUDA-sync forwards, p50 on A100 80 GB PCIe and RTX 4090; split prefix / AE / residual. Totals are measured independently of the stage medians.

References

- [1] A. Brohan *et al.*, “RT-1: Robotics transformer for real-world control at scale,” in *Proc. Robot.: Sci. Syst.*, 2023.
- [2] B. Zitkovich *et al.*, “RT-2: Vision-language-action models transfer web knowledge to robotic control,” in *Proc. Conf. Robot Learn.*, 2023, pp. 2165–2183.
- [3] M. J. Kim *et al.*, “OpenVLA: An open-source vision-language-action model,” in *Proc. Conf. Robot Learn.*, 2025, pp. 2679–2713.

- [4] Open X-Embodiment Collaboration, “Open X-Embodiment: Robotic learning datasets and RT-X models,” in *Proc. ICRA*, 2024, pp. 6892–6903.
- [5] K. Black *et al.*, “ π_0 : A vision-language-action flow model for general robot control,” in *Proc. Robot.: Sci. Syst.*, 2025.
- [6] K. Black *et al.*, “ $\pi_{0.5}$: A vision-language-action model with open-world generalization,” in *Proc. Conf. Robot Learn.*, 2025, pp. 17–40.
- [7] M. Shukor *et al.*, “SmolVLA: A vision-language-action model for affordable and efficient robotics,” arXiv preprint arXiv:2506.01844, 2025.
- [8] Y. Lipman, R. T. Q. Chen, H. Ben-Hamu, M. Nickel, and M. Le, “Flow matching for generative modeling,” in *Proc. ICLR*, 2023.
- [9] X. Liu, C. Gong, and Q. Liu, “Flow straight and fast: Learning to generate and transfer data with rectified flow,” in *Proc. ICLR*, 2023.
- [10] R. Pope *et al.*, “Efficiently scaling transformer inference,” in *Proc. Mach. Learn. Syst.*, 2023.
- [11] J. Ho, A. Jain, and P. Abbeel, “Denoising diffusion probabilistic models,” in *Proc. NeurIPS*, 2020.
- [12] Y. Song *et al.*, “Score-based generative modeling through stochastic differential equations,” in *Proc. ICLR*, 2021.
- [13] A. Tong *et al.*, “Improving and generalizing flow-based generative models with minibatch optimal transport,” *Trans. Mach. Learn. Res.*, 2024.
- [14] C. Chi *et al.*, “Diffusion policy: Visuomotor policy learning via action diffusion,” in *Proc. Robot.: Sci. Syst.*, 2023.
- [15] Y. Ze *et al.*, “3D diffusion policy: Generalizable visuomotor policy learning via simple 3D representations,” in *Proc. Robot.: Sci. Syst.*, 2024.
- [16] T. Z. Zhao, V. Kumar, S. Levine, and C. Finn, “Learning fine-grained bimanual manipulation with low-cost hardware,” in *Proc. Robot.: Sci. Syst.*, 2023.
- [17] D. Ghosh *et al.*, “Octo: An open-source generalist robot policy,” in *Proc. Robot.: Sci. Syst.*, 2024.
- [18] S. Liu *et al.*, “RDT-1B: A diffusion foundation model for bimanual manipulation,” in *Proc. ICLR*, 2025.
- [19] L. Beyer *et al.*, “PaliGemma: A versatile 3B VLM for transfer,” arXiv preprint arXiv:2407.07726, 2024.
- [20] K. Black, M. Y. Galliker, and S. Levine, “Real-time execution of action chunking flow policies,” in *Proc. NeurIPS*, 2025.
- [21] J. Song, C. Meng, and S. Ermon, “Denoising diffusion implicit models,” in *Proc. ICLR*, 2021.
- [22] C. Lu *et al.*, “DPM-Solver: A fast ODE solver for diffusion probabilistic model sampling in around 10 steps,” in *Proc. NeurIPS*, 2022.
- [23] T. Salimans and J. Ho, “Progressive distillation for fast sampling of diffusion models,” in *Proc. ICLR*, 2022.
- [24] Y. Song *et al.*, “Consistency models,” in *Proc. ICML*, 2023.
- [25] S. Luo *et al.*, “Latent consistency models: Synthesizing high-resolution images with few-step inference,” arXiv preprint arXiv:2310.04378, 2023.
- [26] K. Frans *et al.*, “One step diffusion via shortcut models,” in *Proc. ICLR*, 2025.

- [27] A. Prasad, K. Lin, J. Wu, L. Zhou, and J. Bohg, “Consistency policy: Accelerated visuomotor policies via consistency distillation,” in *Proc. Robot.: Sci. Syst.*, 2024.
- [28] W. Song *et al.*, “L1 sample flow for efficient visuomotor learning,” arXiv preprint arXiv:2511.17898, 2025.
- [29] J. Jia *et al.*, “Action-to-action flow matching,” in *Proc. Robot.: Sci. Syst.*, 2026.
- [30] Y. Zhong *et al.*, “From noise to intent: Anchoring generative VLA policies with residual bridges,” arXiv preprint arXiv:2604.21391, 2026.
- [31] T. Silver *et al.*, “Residual policy learning,” arXiv preprint arXiv:1812.06298, 2018.
- [32] T. Johannink *et al.*, “Residual reinforcement learning for robot control,” in *Proc. ICRA*, 2019, pp. 6023–6029.
- [33] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 770–778.
- [34] A. Vaswani *et al.*, “Attention is all you need,” in *Proc. NeurIPS*, 2017.
- [35] Y. Mu *et al.*, “RoboTwin: Dual-arm robot benchmark with generative digital twins,” in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, 2025, pp. 27649–27660.
- [36] T. Chen *et al.*, “RoboTwin 2.0: A scalable data generator and benchmark with strong domain randomization for robust bimanual robotic manipulation,” arXiv preprint arXiv:2506.18088, 2025.
- [37] B. Efron, “Bootstrap methods: Another look at the jackknife,” *Ann. Statist.*, vol. 7, no. 1, pp. 1–26, 1979.
- [38] I. Loshchilov and F. Hutter, “Decoupled weight decay regularization,” in *Proc. ICLR*, 2019.
- [39] I. Loshchilov and F. Hutter, “SGDR: Stochastic gradient descent with warm restarts,” in *Proc. ICLR*, 2017.