ACTIONCACHE: TRAINING-FREE ACCELERATION FOR VISION-LANGUAGE-ACTION MODELS WITH ACTION CACHING AND REFINEMENT

Anonymous authors
Paper under double-blind review

ABSTRACT

Vision-Language-Action (VLA) models is a promising approach for generalizable robotic manipulation. Recently, VLA models that employ flow-matching-based action heads have shown remarkable success, as they can generate precise and smooth action sequences while capturing multimodal action distributions. However, these action heads require iterative denoising, which acts as a major computational bottleneck in inference, posing a critical challenge for real-time VLA deployment. To address this challenge, we propose *ActionCache*, a plug-and-play external cache that opportunistically reuses past intermediate actions. Specifically, ActionCache stores intermediate actions with compact multimodal keys and retrieves them from similar past contexts to warm-start generation from the vicinity of target actions. This mechanism reduces the number of required denoising steps, leading to substantially lower inference latency. Experimental results in simulation and real-world environments demonstrate that ActionCache maintains high task success rates in a low-latency regime, achieving action head inference acceleration of up to $10.44\times$ and $40.17\times$ for representative flow-matching-based VLA, $\pi_{0.5}$ and GR00T-N1.6, respectively.

1 INTRODUCTION

Vision-Language-Action (VLA) models have emerged as a promising foundation for generalist robot policies that map visual observations and language instructions directly to low-level control (Kawaharazuka et al., 2025; Ma et al., 2026). Most recent VLA models consist of a Vision-Language Model (VLM) backbone and a flow-matching-based (hereafter, flow-based) action head, which generates continuous action sequences (Shukor et al., 2025; Bjorck et al., 2025; Black et al., 2026). By operating directly in a continuous action space, these flow-based VLA models can avoid action quantization artifacts, capture multimodal action distributions, and produce smooth and precise trajectories.

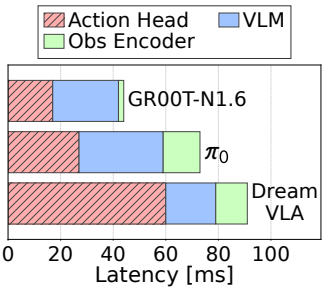


Figure 1: **Inference latency breakdown for representative VLA models on NVIDIA RTX 4090 GPU.** Action head accounts for 37-66% of the end-to-end latency.

Despite their strong performance, real-time deployment of flow-based VLA models remains challenging because the action head iteratively evaluates a generative model to transform noise into a structured action trajectory at every replanning step. Although some prior acceleration methods mainly optimize the VLM backbone (Xu et al., 2025; Yang et al., 2025), the action head remains a major computational bottleneck, accounting for over 65% for DreamVLA (Zhang et al., 2025) and 39% for GR00T-N1.6 (Bjorck et al., 2025), as shown in Figure 1.

Recently, warm-start methods have been proposed to exploit temporal continuity by initializing refinement from recent actions, predicted actions, or trajectory-level priors (Duan et al., 2025; Li et al., 2026; Chen et al., 2025; Høeg et al., 2025). These methods are typically framed as improving local temporal smoothness by breaking the Markov Decision Process underlying action generation,

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.

in which the model is conditioned solely on the current observation. As a byproduct, they can also shorten the effective distance traveled during the iterative refinement by starting closer to the target action trajectory, rather than forcing the model to synthesize an action chunk from noise. However, this effect has not been systematically studied as a general mechanism for output reuse beyond temporal continuity. Moreover, existing warm-start methods often depend on learned predictors or explicit temporal modeling rather than offering a plug-and-play solution for pretrained flow-based VLAs.

This paper proposes *ActionCache*, a plug-and-play external cache that turns past action generation into reusable computation. Robotic control often involves recurring visual states, task phases, and language goals, which can induce similar conditional transport paths toward structured action chunks. ActionCache captures this redundancy by storing generated action chunks with compact multimodal keys and retrieving them when the current context is likely to induce a similar conditional generation path. This reframes warm-starting from a local temporal-continuity heuristic into an output-level retrieval problem. Rather than learning a predictor for the next action or relying solely on previous timesteps, ActionCache opportunistically reuses policy outputs that can initialize refinement closer to the current conditional flow than unstructured noise. In this way, ActionCache substantially reduces the number of required refinement steps in a completely training-free manner, i.e., without retraining the policy, modifying the action head, or adding a learned warm-start module.

Our contributions are as follows:

- We introduce ActionCache, a training-free and plug-and-play external cache that accelerates the flow-based action head of VLA models by reusing and refining past action generations.
- We develop a model-agnostic cache design that stores intermediate action chunks with compact multimodal keys. It enables reusable computation across timesteps and episodes while retaining the pretrained policy through a conservative fallback mechanism.
- We validate ActionCache in simulation and real-world robot manipulation tasks, showing substantial improvements in the latency–success-rate trade-off with up to 10.44× action-head acceleration for $\pi_{0.5}$. We further provide detailed analyses that characterize the factors governing effective action reuse.

2 Background and Related Work

Vision-Language-Action Models. Vision-Language-Action (VLA) models (Kawaharazuka et al., 2025; Ma et al., 2026) generate robotic actions based on visual and linguistic inputs. Most VLA models build on a Vision-Language Model (VLM) (Zhang et al., 2024a) backbone and attach an action head that predicts robot actions from the representation extracted by the VLM. Based on the action generation mechanism of their action heads, VLA models can be categorized into two classes: autoregressive and diffusion-based models. The former discretizes continuous actions into action tokens and predicts them sequentially (Kim et al., 2024), while the latter generates continuous action chunks through an iterative generative process, often using diffusion (Yang et al., 2023; Ho et al., 2020) or flow matching (Lipman et al., 2023). Many recent VLA models employ diffusion-based action heads, particularly flow-matching-based ones, to model continuous actions more expressively. These *flow-based* VLAs have achieved remarkable performance across diverse and complex robotic tasks, positioning them as a prominent class of recent VLA models. Motivated by this trend, this paper focuses on the flow-based VLA architecture family.

At each control timestep t, a flow-based VLA passes a context representation $\boldsymbol{c}_t$ from the VLM backbone to the flow-based action head[1]. Conditioned on $\boldsymbol{c}_t$, the action head generates a continuous action chunk by evolving $\boldsymbol{A}_t^\tau := [\boldsymbol{a}_t^\tau, \ldots, \boldsymbol{a}_{t+H-1}^\tau]$ from $\tau = 0$ to $\tau = 1$, where τ is the flow time, and H is the action horizon (the number of actions in the chunk). Starting from Gaussian noise $\boldsymbol{A}_t^0 \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{I})$, the action head iteratively transforms the intermediate action chunk as

$$\boldsymbol{A}_t^{\tau+\Delta\tau} = \boldsymbol{A}_t^\tau + \boldsymbol{V}_\theta(\boldsymbol{A}_t^\tau, \tau, \boldsymbol{c}_t)\Delta\tau. \tag{1}$$

[1]Note that the exact form of $\boldsymbol{c}_t$ is architecture-dependent; it may be obtained from the VLM's final output embeddings (Bjorck et al., 2025) or from internal representations such as KV caches (Shukor et al., 2025; Black et al., 2025; 2026).

Here, $\boldsymbol{V}_\theta$ is the velocity field predicted by the action head, and $\Delta\tau$ is the numerical integration step size. After integration to $\tau = 1$, the final action chunk is obtained as $\boldsymbol{A}_t^1 = [\boldsymbol{a}_t^1, \ldots, \boldsymbol{a}_{t+H-1}^1]$.

Plug-and-Play Acceleration of Flow-based VLA Models. As shown in Equation (1), generating each action chunk in a flow-based VLA requires multi-step flow integration, where the action head repeatedly evaluates the velocity field $\boldsymbol{V}_\theta$. Thus, the number of flow matching steps directly determines the number of action head forward passes and is a major bottleneck of inference latency. For instance, in the inference of representative flow-based VLAs, the action head accounts for more than 37% of the end-to-end latency as shown in Figure 1, making it a bottleneck for real-time robot control.

To alleviate this bottleneck, prior works have explored plug-and-play acceleration methods that can be applied to pretrained VLAs without costly retraining. EfficientVLA (Yang et al., 2025) reduces redundant computation across the VLA pipeline by pruning less informative language layers, selecting a compact subset of visual tokens, and caching intermediate features in the diffusion-based action head. While effective, such methods primarily reduce the per-step computational cost or reuse features within the original sampler. They do not fundamentally eliminate the need for multi-step flow integration, and the action head must still be invoked repeatedly. This limitation motivates us to implement a training-free acceleration strategy that more directly reduces the iteration cost of action heads while preserving the behavior of the pretrained VLA.

Diffusion Caching on Image Generation. In image generation, prior work on efficient diffusion inference has shown that caching can reduce denoising costs by reusing previously computed outputs, latent states, or intermediate features. They retrieve intermediate latent states (Agarwal et al., 2024) or final images (Xia et al., 2025) of similar previous prompts stored in caches to serve as warm starts, skipping part of the denoising process. Intermediate features across adjacent denoising steps have also been reused to avoid redundant computation (Ma et al., 2024). While these caching methods are training-free and effective for image generation, they cannot be applied directly to flow-based VLA inference, where cache retrieval must be conditioned on multimodal and embodied contexts, including visual observations, language instructions, robot states, and task progress. To the best of our knowledge, ActionCache is the first method that introduces the caching paradigm to flow-based VLA and optimizes both inference speed and accuracy.

Warm-Starting for Diffusion Policies. Existing warm-starting methods for diffusion-based robot policies (Chi et al., 2023) mainly exploit local temporal continuity within a rollout by reusing previous action predictions or temporally adjacent trajectories (Li et al., 2026; Høeg et al., 2025). These methods often need training of predictors or explicit temporal modeling. STEP (Li et al., 2026) introduces a specialized predictor to generate a temporally and spatially consistent warm-start initialization from the temporally adjacent action trajectory. SDP (Høeg et al., 2025) trains a diffusion policy to handle action chunks with mixed noise levels, enabling temporal streaming synthesis and accelerated inference. In contrast, ActionCache aims to reuse action generation across temporally distant but semantically and visually related contexts *without training*. It enables acceleration by caching action-relevant generation states that are shared across episodes, rather than relying solely on short-term continuity between consecutive control steps.

3 METHOD

3.1 FRAMEWORK OVERVIEW

To address the high inference latency of the iterative action refinement in the flow-based action head, we propose *ActionCache*, a plug-and-play action memoization framework. Figure 2 illustrates our framework. Given the current observation at a control timestep t, the VLM backbone produces a condition $\boldsymbol{c}_t$ for the action head. Leveraging the VLM output embeddings generated as a byproduct of conditioning, ActionCache constructs a cache *key* $\boldsymbol{k}_t$, a compact representation that extracts multimodal information from the backbone features in a training-free manner, and uses it to query existing cache entries from past successful action generations.

Each cache entry contains a cache key $\boldsymbol{k}_t$, an action chunk $\boldsymbol{A}_t^\tau$ generated by the base policy, and optional metadata $\boldsymbol{m}_t$ for cache management such as reference counts. Importantly, the cached value is the action chunk itself—not internal states, KV cache, or visual token representation–and is

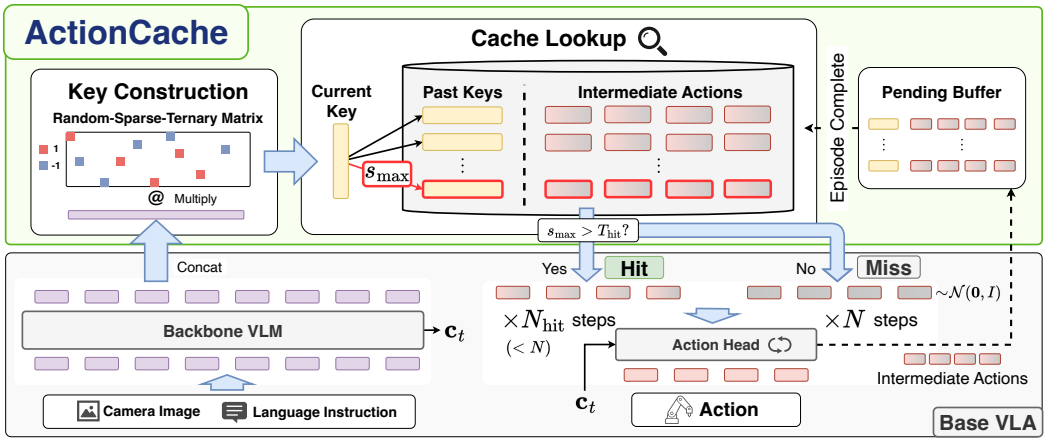


Figure 2: **ActionCache Framework.** ActionCache is an external cache that stores intermediate noisy actions from past generations. On a cache hit, ActionCache initializes from retrieved past similar actions, reducing the number of denoising steps.

therefore defined in the action space rather than tied to a particular backbone architecture. Moreover, it leaves the backbone and action-head weights unchanged. Therefore, ActionCache can be attached to a pretrained VLA policy without additional training.

At inference time, the current key is used to retrieve a candidate cache entry together with a similarity score. ActionCache then branches according to a conservative hit/miss rule. If the similarity of the retrieved entry exceeds the hit threshold T_{hit}, the lookup is treated as reliable, and the retrieved action chunk is used as an initialization for zero- or few-step action generation. Otherwise, the retrieved candidate is rejected, and the model falls back to the standard full-step generation process from pure Gaussian noise $\boldsymbol{A}_t^{\tau=0} \sim \mathcal{N}(\mathbf{0}, I)$. This fallback strategy ensures that the VLA model retains its original robustness and generalization capabilities in unseen or complex situations where the cache is not sufficiently populated.

3.2 Cache Representation and Retrieval for Multimodal Contexts

A useful cache key should be expressive enough to capture action-relevant multimodal context while remaining inexpensive to compute and search for at every control step. We therefore construct keys from VLM output embeddings produced during the backbone forward pass. This choice does not require an additional VLM computation, and in architectures where some VLM output tokens are not directly consumed by the action head, it also reuses features that would otherwise be discarded. Moreover, these embeddings encode visual and linguistic context after backbone processing, making them a more action-relevant retrieval signal than raw image or language features alone. We empirically compare this design with alternative feature sources in Appendix A.

Let $\boldsymbol{h}_t \in \mathbb{R}^D$ be the concatenated VLM output embeddings. For models such as GR00T-N1.6 (Bjorck et al., 2025) whose VLM outputs do not include robot proprioceptive state, $\boldsymbol{h}_t$ is a concatenation of the VLM output features and the encoded robot state features $\boldsymbol{r}_t$. Since D can be very large, using $\boldsymbol{h}_t$ directly as a cache key would increase memory footprint and lookup cost. Therefore, we reduce its dimensionality by projecting it into a compact key $\boldsymbol{k}_t \in \mathbb{R}^d$ using a fixed sparse ternary random projection matrix (Li et al., 2006):

$$\boldsymbol{k}_t = \boldsymbol{R}\boldsymbol{h}_t, \qquad \boldsymbol{R} = (r_{ij}) \in \{-1, 0, 1\}^{d \times D}. \tag{2}$$

Here, each row of $\boldsymbol{R}$ contains $pD/2$ entries of 1, $pD/2$ entries of -1, and the remaining $(1-p)D$ entries of 0. The position of these values is chosen uniformly at random for each row. $\boldsymbol{R}$ is created once and kept fixed. This compression approach requires no fitting data, retraining, or online update. It therefore preserves the plug-and-play nature of ActionCache while reducing storage and similarity search costs.

Given the current key $\boldsymbol{k}_t$, ActionCache retrieves the nearest cache entry $\boldsymbol{A}_{i^\star}$ using cosine similarity: $i^\star = \arg\max_i \cos_\text{sim}(\boldsymbol{k}_t, \boldsymbol{k}_i)$. The retrieved action is considered a hit only when the Top-1 sim-

ilarity $s_{\max} = \cos_\text{sim}(\boldsymbol{k}_t, \boldsymbol{k}_{i^\star})$ exceeds the threshold T_{hit}. On a hit, the retrieved action chunk is either executed directly or used as the initialization for a small number of refinement steps under the current condition $\boldsymbol{c}_t$.

3.3 Cache Population and Management

During a cache miss, we extract the intermediate noisy action at denoising step $N - N_{\text{hit}}$ from its full generation trajectory. Here, N denotes the total number of function evaluations (NFE), and N_{hit} represents the NFE on a cache hit. This intermediate action $\boldsymbol{A}_t^\tau$ at $\tau = (N - N_{\text{hit}})\Delta\tau$, paired with its corresponding key $\boldsymbol{k}_t$, is temporarily stored in a pending buffer. If the episode completes successfully, these pending pairs $\{\boldsymbol{k}_t, \boldsymbol{A}_t^\tau\}$ are committed to the main cache for future reuse; otherwise, they are discarded.

Once the cache reaches its maximum capacity, an existing cache entry is evicted and replaced with a new one according to a cache replacement policy. This process allows ActionCache to adapt to changing environments and new tasks. In many routine scenarios, tasks exhibit temporal locality, meaning that related actions are likely to occur again in the near future. This property makes Least Recently Used (LRU) a reasonable policy (Agarwal et al., 2024). However, LRU can suffer from cache pollution, where a newly inserted but rarely reused entry evicts a more reusable one. This effect becomes more pronounced in a capacity-limited scenario, making Least-Frequently-Used (LFU) a better choice, as we show in the results shortly (Section 4.3).

ActionCache can be viewed as a generalization of purely temporal warm starting. By modifying the buffering and replacement policy (e.g., to FIFO), it can supply the necessary inputs to a temporal continuity framework, although such work requires an additional learned module to complete warm-starting. ActionCache extends this perspective by introducing an indexed memory that can retrieve action chunks across timesteps and episodes. This broader reuse requires cache quality assessment and filtering, but it also allows the policy to benefit from prior successful behavior beyond the immediately preceding control step in a plug-and-play manner.

4 Experiments

4.1 Evaluation Settings

Implementation details. We evaluate two state-of-the-art flow-based VLA models: $\pi_{0.5}$ (Black et al., 2025) and GR00T-N1.6 (Bjorck et al., 2025). For GR00T-N1.6, instead of raw VLM output embeddings, which do not include robot-state features, we use a concatenation of encoded VLM output embeddings and encoded robot-state features as a source of cache key. We set the dimension of the cache key to $d = 500$ with the non-zero value density of the sparse ternary random matrix $p = 0.01$. Since d and p are insensitive to success rate (see Appendix D), we select sufficiently large values that can maintain efficient cache lookup. The main comparison reports steady-state performance, while Section 4.4 separately evaluates cache warmup and long-span behavior. To evaluate steady-state performance of ActionCache, we fill the cache by running the rollouts with $T_{\text{hit}} = 1$, and then start evaluation with a fixed T_{hit}. T_{hit} is systematically selected by profiling the similarity distribution of the warmup phase and identifying the shoulder of the hit-rate curve before it plateaus. By default, T_{hit} is set to 0.85 for $\pi_{0.5}$ and 0.65 for GR00T-N1.6. Warmup and test episodes use disjoint random seeds, ensuring no cached action is generated under an identical task configuration (e.g., object placement) to any test episode. The default cache size is set to $3{,}000$ entries managed by the LRU policy.

For both models, we set the action execution horizon to 10. Unless otherwise specified, we use these hyperparameters as the default configuration throughout our experiments.

Simulation Evaluation. To validate the efficacy of our framework in various task settings, we use VLABench (Zhang et al., 2024b) and LIBERO (Liu et al., 2023) benchmarks. VLABench is a large-scale robot manipulation benchmark that contains 100 tasks with $2{,}000+$ objects and evaluates visual/spatial understanding and common-sense/world-knowledge application. We evaluate on the 10 primitive tasks of varying difficulty from VLABench, which are released as an official unified dataset on Hugging Face. Throughout warmup and test time, these tasks are executed in a round-robin manner, which means the cache contains entries from all the tasks. The latency is measured

Table 1: **Overall performance comparison on VLABench.** ActionCache preserves success rate in low-NFE and low-latency regime, while the simple NFE reduction and prior efficient methods exhibit substantial degradation. The latency of ActionCache includes key projection and cache lookup overhead. SR(%) and Lat.(ms) denote success rate and action head latency, respectively.

	$\pi_{0.5}$				GR00T-N1.6			
Method	**NFE**	**SR(↑)**	**Lat.(↓)**	**Speedup**	**NFE**	**SR(↑)**	**Lat.(↓)**	**Speedup**
Base model	10†	$38.8_{\pm 3.0}$	18.8	1.00×	4†	$\mathbf{34.0}_{\pm 1.2}$	24.1	1.00×
	1	$6.8_{\pm 2.9}$	2.5	7.52×	1	$24.6_{\pm 1.4}$	6.9	3.49×
EfficientVLA	2	17.9	5.2	3.62×	2	19.7	15.3	1.58×
	1	7.3	3.6	5.22×	1	8.2	9.2	2.62×
Falcon	–	13.1	13.0	1.45×	–	26.1	18.8	1.28×
	–	7.6	6.8	2.76×	–	19.3	15.6	1.54×
ActionCache (Ours)	2	$39.4_{\pm 2.8}$	5.5	3.42×	2	$33.9_{\pm 0.9}$	13.6	1.77×
	1	$\mathbf{41.0}_{\pm 1.2}$	3.6	5.22×	1	$32.1_{\pm 3.2}$	7.5	3.21×
	0	$40.9_{\pm 2.8}$	**1.8**	**10.44**×	0	$30.8_{\pm 1.8}$	**0.6**	**40.17**×

† Default NFE for each base model.

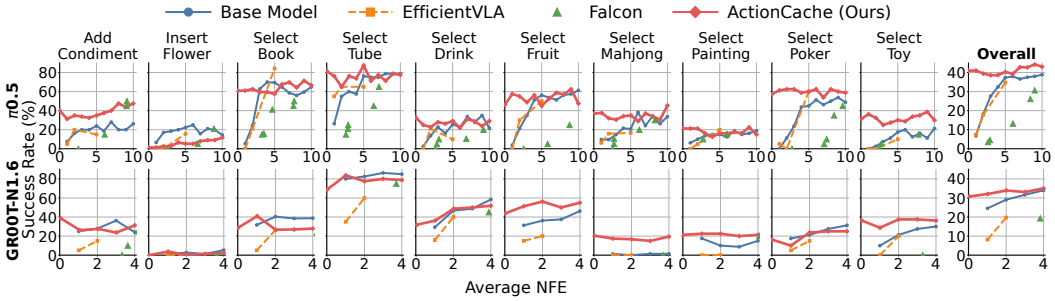


Figure 3: **Success rate and average NFEs on VLABench.** ActionCache provides a better trade-off of NFE and success rate.

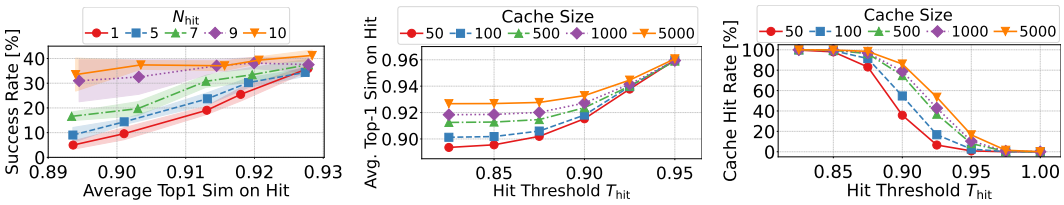
on a single NVIDIA RTX 5090 GPU with 32 GB of GPU memory. More details and the results for the LIBERO benchmark are reported in Appendix G.

Real-World Evaluation. We deploy the $\pi_{0.5}$ model on SO-101 (Cadene et al., 2026), an open-sourced 6-DoF robotic arm. We evaluate on three tasks: `pick_sausage`, `pick_and_close`, and `push_button`. These tasks cover evaluation of robustness to distractor objects and spatial perturbations, multi-step long-horizon manipulation, and instruction following through visual-language grounding. Detailed task descriptions and fine-tuning settings are provided in Appendix C.

4.2 Success Rate and Latency Evaluation

Overall Performance Comparison on VLABench. To validate the efficacy of ActionCache, we compare with a simple denoising step reduction model and two prior training-free acceleration methods, EfficientVLA (Yang et al., 2025) and Falcon (Chen et al., 2025). We measure success rate with a total of 800 episodes for base models and ActionCache, and 200 episodes for other methods. We set the cache size to 10,000 for $\pi_{0.5}$ and 30,000 for GR00T-N1.6, and evaluate its steady-state performance. Table 1 summarizes the number of function evaluations (NFE), success rate and inference latency of the action head in the low-latency (low-NFE) regime. Note that the NFE of ActionCache is equal to the N_{hit} value.

Although the base models drop their success rates severely at $\text{NFE} = 1$, ActionCache mitigates the degradation in the low-NFE regime. This result demonstrates that by starting from the vicinity of the target action rather than from pure noise, ActionCache can generate a high-fidelity action even with a minimal NFE. Compared to EfficientVLA, which is a prior plug-and-play acceleration method, ActionCache achieves a better trade-off between success rate and latency. This is because EfficientVLA's acceleration is limited by its reliance on pure noise initialization and its inability to skip the entire execution of a denoising step. Falcon, a warm-starting method for diffusion policies (Chi



(a) (b) (c)

Figure 4: **(a): Average Top-1 similarity vs success rate for various numbers of denoising steps** N_{hit}. As the Top-1 similarity increases, the success rates improve. **(b): Average Top-1 similarity vs hit thresholds for various cache sizes.** As the hit threshold increases, average Top-1 similarity also increases. **(c): Cache hit rate vs various hit thresholds for various cache sizes.** A larger cache yields a higher cache hit rate.

et al., 2023), exhibits a less favorable latency–success-rate trade-off likely due to the overhead of selecting an initial trajectory and a limited correctionability of recent-trajectory warm-starting under deterministic flow integration. Furthermore, especially on $\pi_{0.5}$, ActionCache maintains a high success rate at $\text{NFE} = 0$ with negligible latency overhead induced by key projection and cache lookup. By directly reusing past actions in similar contexts, ActionCache achieves a drastic speedup, suggesting that past actions can be retrieved and reused without modification.

Figure 3 shows the trade-off between average NFE and success rate for each task in VLABench. Overall, ActionCache achieves a better trade-off compared to the base model and the other prior methods across a broad range of tasks, especially in the low-NFE regime.

4.3 Impact of Retrieval Quality on Success Rate

Retrieval Quality and Success Rate. We investigate the relationship between the quality of retrieved actions and task success rate with varying hyperparameters for ActionCache. The quality of retrieved actions is measured by average Top-1 cosine similarity on cache hits, which we control by varying the cache size from 50 (114 KB) to 5,000 (11.4 MB).

From Figure 4a, we can observe that when the average Top-1 similarity on cache hits is high enough, the model achieves high success rates comparable to that of the full-step generation, regardless of N_{hit}. This observation indicates that ActionCache requires either a sufficiently large N_{hit} or a high average Top-1 similarity to achieve a high task success rate. Since increasing N_{hit} negates the computational benefits of ActionCache, it is necessary to increase the Top-1 similarity on retrieval for a good trade-off between success rate and latency.

Hit threshold T_{hit} serves as a key parameter to control success rate. Figure 4b shows the relationship between the average Top-1 similarity and T_{hit}, where a higher T_{hit} leads to a higher average Top-1 similarity. With a sufficiently large cache, the average Top-1 similarity saturates at lower thresholds because the nearest-neighbor similarity distribution is already concentrated above them, making the hit criterion non-selective. Since T_{hit} enforces a lower bound on the similarity of accepted retrievals, it provides a practical control knob for the quality of retrieved actions and, consequently, the expected success rate. This allows us to tune the trade-off of cache capacity and latency benefit as below with a target success rate.

Finally, Figure 4c shows the cache hit rate with varying T_{hit} and cache sizes. We observe that increasing the cache size monotonically improves the cache hit rate, which is directly translated into lower latency. Overall, these results indicate that ActionCache provides highly flexible knobs to adjust accuracy-efficiency-capacity trade-offs, depending on the task difficulty, success rate, target latency, and allocated memory.

Comparison of Cache Replacement Policies. As established in Section 4.3, the quality of retrieved actions is important to achieve a high success rate in ActionCache. With a large cache size, we can retrieve sufficiently high-quality actions from the cache with a higher cache hit rate, whereas in a restricted cache size, the replacement policy is crucial for a higher cache hit rate under a specific hit threshold T_{hit}. To this end, we investigate the impact of different cache replacement policies on the

cache hit rate. We compare five policies: Random, Least Recently Used (LRU), Least Frequently Used (LFU), First In First Out (FIFO), and an oracle algorithm (Belady, 1966), which assumes future knowledge of queries and serves as the ideal replacement policy.

As shown in Figure 5, in limited capacity, the LFU policy demonstrates the highest cache hit rate among the other policies. In a typical caching mechanism, the LRU policy is likely to achieve a high hit rate due to its ability to exploit temporal locality. While LRU is susceptible to cache pollution by *one-hit wonder* actions, such as highly specific and difficult grasps, LFU prioritizes robust and highly reusable actions that are shared across tasks or episodes. Although LFU captures temporal locality less effectively than LRU, retaining task- or episode-agnostic highly reusable actions in the cache contributes greatly to the hit rate in those settings. Notably, the FIFO policy, which implicitly measures the probability of encountering a good starting point (specified by T_{hit}) of traditional warm-starting methods for temporal continuity, is no better than random replacement in a plug-and-play setting, suggesting a missed opportunity for caching.

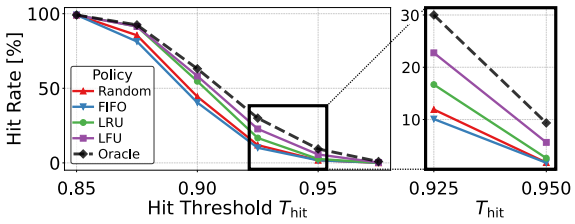


Figure 5: **Hit rate comparison for cache replacement policies under a restricted cache size of 100.** LFU policy demonstrates the highest hit rate.

4.4 Cache Population and Long-Span Deployment

The main experiments evaluate ActionCache after cache warmup to characterize its steady-state performance. Here, we study its behavior during the transition from cache population to long-span deployment. As the cache is populated, reusable actions gradually accumulate; once the cache reaches capacity, newly generated entries continuously replace existing ones according to the replacement policy. We conduct a long-span experiment on VLABench with $\pi_{0.5}$ and track the moving averages of task success and cache hit rate. We set the hit threshold to $T_{\text{hit}} = 0.88$ and $N_{\text{hit}} = 1$ so that cache misses and replacements continue to occur after the cache is populated.



Figure 6: **Moving average of task success rate and cache hit rate (window size=50).**

Figure 6 shows the evolution of task success rate and cache hit rate from the warmup phase through long-span deployment. After cache population, the hit rate increases and the average NFE correspondingly decreases, bumping up action head speedup from $1\times$ (warmup) to $4.5\times$ (steady). Then, these metrics stabilize, and both the success rate and acceleration remain stable even while cache entries are continuously replaced. This experiment therefore complements the steady-state evaluation by showing transitions from cache warmup to sustained accelerated operation.

4.5 Evaluation on Real-World

To validate the efficacy of ActionCache in a real-world environment, we evaluate $\pi_{0.5}$ on three robotic arm manipulation tasks. For ActionCache, we set the cache size to 300 for the `push_button` task and 1,000 for the others. T_{hit} is set to 0.875 for `pick_sausage` and 0.85 for the others.

Success Rate and Inference Latency. Table 2 summarizes the results. Overall, ActionCache achieves a success rate comparable to the base model while reducing action-head latency, improving overall latency, and demonstrating practical applicability in real-world settings. Table 2 also shows that ActionCache can be introduced with only a negligible latency overhead. Notably, the `push_button` task achieves over 94% hit rate while maintaining a 100% success rate. Despite its longer horizon resulting in a smaller number of cached episodes, the `pick_and_close` task

Table 2: **Success rate and latency breakdown of $\pi_{0.5}$ in the real-world environment.** Overall and Action head latency includes ActionCache's overhead. ActionCache maintains success rate while reducing action head latency with negligible overhead. SR and HR denote success rate and hit rate.

Task	Method	NFE	SR (%)	HR (%)	Latency (ms)			
					Emb.	VLM	Action head	Overall
`pick_sausage`	Base Model	10	90	-	24.22	22.21	56.70 (1.00×)	103.13 (1.00×)
	ActionCache	1	88	41.8	24.26	22.24	35.06 (**1.62**×)	81.90 (**1.26**×)
`pick_and_close`	Base Model	10	88	-	24.23	22.22	55.79 (1.00×)	102.24 (1.00×)
	ActionCache	1	90	82.8	24.29	22.26	14.67 (**3.80**×)	61.56 (**1.66**×)
`push_button`	Base Model	10	100	-	24.31	22.29	59.65 (1.00×)	106.26 (1.00×)
	ActionCache	1	100	94.4	24.33	22.32	9.53 (**6.26**×)	56.46 (**1.88**×)

exhibits a higher cache hit rate than the `pick_sausage` task. Since the `pick_sausage` includes randomly positioned task-irrelevant distractor objects, this result suggests that visual clutter and observation perturbations might affect the key matching process. More detailed analyses about key similarity are provided in Appendix F.

4.6 End-to-end Performance and Combination with prior VLM acceleration

We study end-to-end performance by coupling ActionCache with VLA-Cache (Xu et al., 2025), a prior *VLM backbone acceleration method* which reduces redundant computation by adaptively reusing visual token KV cache. For this experiment, we use $\pi_{0.5}$ and three tasks of varying difficulty: `select_tube`, `select_painting` and `add_condiment`. More details, including VLA-Cache settings, are provided in Appendix C. As shown in Table 3, VLA-Cache and ActionCache individually accelerate the VLM and action head without dropping the success rate, although accelerating the action head provides substantially larger end-to-end gains than accelerating the VLM backbone alone.

Table 3: **End-to-end $\pi_{0.5}$ performance comparison of ablation on ActionCache and VLA-Cache.**

VLA-Cache	Action Cache	T_{hit}	SR (%)	Latency (ms)		
				VLM	Head	Overall
–	–	–	40.7	27.6	58.4	97.4 (1.00×)
✓	–	–	**41.4**	24.0	58.8	94.2 (1.03×)
–	✓	0.85	<u>41.2</u>	27.6	**6.8**	<u>45.9</u> (<u>2.12</u>×)
✓	✓	0.85	30.0	**23.9**	7.4	**42.8** (**2.28**×)
		0.91	39.1	**23.9**	21.6	57.0 (1.71×)

We further examine whether VLM-side acceleration provides additional gains on top of ActionCache. Directly combining both methods in the same T_{hit} ($= 0.85$) further reduces latency, but degrades the success rate, likely because reused visual tokens in VLA-Cache distort the similarity distribution. A recalibrated T_{hit} ($= 0.91$) largely restores success rate, but reduces cache hits and resulting speedup. This suggests that jointly accelerating the VLM and action head requires improving key quality without sacrificing cache reuse. Under the training-free constraint, ActionCache *alone* provides the best latency at comparable success rate, while future work can design VLM acceleration methods that preserve key reliability.

5 Conclusion and Limitations

In this paper, we propose ActionCache, a plug-and-play external cache that accelerates action heads, a major bottleneck in flow-based VLA inference. Through our experiments, we demonstrate that ActionCache can drastically improve the latency–success-rate trade-off of action head compared to the base model and prior acceleration methods.

Limitations. Our framework introduces extra hyperparameters, T_{hit} and N_{hit}, in addition to accuracy-insensitive ones d and p (see Appendix D). While T_{hit} and N_{hit} can be tuned before deployment based on training/warmup information and our hyperparameter analysis in Section 4.3, further optimization and dynamic adaptation of these may improve the trade-off between accuracy and acceleration. We leave the design of effective hyperparameter optimization strategies for future work.

AI USE STATEMENT

We used generative AI tools to refine hypotheses, provide feedback on research methodology and experiments, implement methods, and assist in interpreting results. We did not use generative AI tools for other tasks requiring disclosure, and the remaining required-disclosure tasks are not applicable to this work. Additionally, we used generative AI tools to identify and summarize relevant literature, assist with writing and drafting parts of the paper, and create code for generating figures. All AI-assisted outputs were reviewed and verified by the authors for correctness and appropriateness before being incorporated into the work. We take responsibility for the final content of this work, including text, claims, code, and artifacts produced with the aid of generative AI.

ETHICS STATEMENT

As with VLA models in general, unexpected actions may lead to unsafe robot behavior. In our method, incorrect action retrieval may also result in unintended motions. Appropriate safety measures should therefore be employed when deploying the method on physical robots.

REPRODUCIBILITY STATEMENT

We provide implementation details, experimental settings, and hyperparameters in the main text and appendices. The code used in our experiments is included in the supplementary material to facilitate reproducibility.

REFERENCES

Shubham Agarwal, Subrata Mitra, Sarthak Chakraborty, Srikrishna Karanam, Koyel Mukherjee, and Shiv Kumar Saini. Approximate caching for efficiently serving text-to-image diffusion models. In *USENIX Symposium on Networked Systems Design and Implementation*, 2024.

L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 1966.

Johan Bjorck, Fernando Castañeda, Nikita Cherniadev, Xingye Da, Runyu Ding, Linxi, Yu Fang, Dieter Fox, Fengyuan Hu, Spencer Huang, Joel Jang, Zhenyu Jiang, Jan Kautz, Kaushil Kundalia, Lawrence Lao, Zhiqi Li, Zongyu Lin, Kevin Lin, Guilin Liu, Edith LLontop, Loic Magne, Ajay Mandlekar, Avnish Narayan, Soroush Nasiriany, Scott Reed, You Liang Tan, Guanzhi Wang, Zu Wang, Jing Wang, Qi Wang, Jiannan Xiang, Yuqi Xie, Yinzhen Xu, Zhenjia Xu, Seonghyeon Ye, Zhiding Yu, Ao Zhang, Hao Zhang, Yizhou Zhao, Ruijie Zheng, and Yuke Zhu. GR00T N1: An open foundation model for generalist humanoid robots. *arXiv preprint arXiv:2503.14734*, 2025.

Kevin Black, Noah Brown, James Darpinian, Karan Dhabalia, Danny Driess, Adnan Esmail, Michael Robert Equi, Chelsea Finn, Niccolo Fusai, Manuel Y. Galliker, Dibya Ghosh, Lachy Groom, Karol Hausman, brian ichter, Szymon Jakubczak, Tim Jones, Liyiming Ke, Devin LeBlanc, Sergey Levine, Adrian Li-Bell, Mohith Mothukuri, Suraj Nair, Karl Pertsch, Allen Z. Ren, Lucy Xiaoyang Shi, Laura Smith, Jost Tobias Springenberg, Kyle Stachowicz, James Tanner, Quan Vuong, Homer Walke, Anna Walling, Haohuan Wang, Lili Yu, and Ury Zhilinsky. $\pi_{0.5}$: A vision-language-action model with open-world generalization. In *Conference on Robot Learning*, 2025.

Kevin Black, Noah Brown, Danny Driess, Adnan Esmail, Michael Equi, Chelsea Finn, Niccolo Fusai, Lachy Groom, Karol Hausman, Brian Ichter, Szymon Jakubczak, Tim Jones, Liyiming Ke, Sergey Levine, Adrian Li-Bell, Mohith Mothukuri, Suraj Nair, Karl Pertsch, Lucy Xiaoyang Shi, James Tanner, Quan Vuong, Anna Walling, Haohuan Wang, and Ury Zhilinsky. π_0: A vision-language-action flow model for general robot control. *arXiv preprint arXiv:2410.24164*, 2026.

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Yash Katariya, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018.

Remi Cadene, Simon Alibert, Francesco Capuano, Michel Aractingi, Adil Zouitine, Pepijn Kooijmans, Jade Choghari, Martino Russi, Caroline Pascal, Steven Palma, Dana Aubakirova, Mustafa Shukor, Jess Moss, Alexander Soare, Quentin Lhoest, Quentin Gallouédec, and Thomas Wolf. LeRobot: An open-source library for end-to-end robot learning. In *International Conference on Learning Representations*, 2026.

Haojun Chen, Minghao Liu, Chengdong Ma, Xiaojian Ma, Zailin Ma, Huimin Wu, Yuanpei Chen, Yifan Zhong, Mingzhi Wang, Qing Li, and Yaodong Yang. Falcon: Fast visuomotor policies via partial denoising. In *International Conference on Machine Learning*, 2025.

Cheng Chi, Siyuan Feng, Yilun Du, Zhenjia Xu, Eric Cousineau, Benjamin Burchfiel, and Shuran Song. Diffusion policy: Visuomotor policy learning via action diffusion. In *Robotics: Science and Systems*, 2023.

Yuqing Du, Ksenia Konyushkova, Misha Denil, Akhil Raju, Jessica Landon, Felix Hill, Nando de Freitas, and Serkan Cabi. Vision-language models as success detectors. In *Conference on Lifelong Learning Agents*, 2023.

Yufei Duan, Hang Yin, and Danica Kragic. Real-time iteration scheme for diffusion policy. In *International Conference on Intelligent Robots and Systems*, 2025.

Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models. *Advances in Neural Information Processing Systems*, 2020.

Sigmund H Høeg, Yilun Du, and Olav Egeland. Fast policy synthesis with variable noise diffusion models. In *International Conference on Robotics and Automation*, pp. 4821–4828, 2025.

Kento Kawaharazuka, Jihoon Oh, Jun Yamada, Ingmar Posner, and Yuke Zhu. Vision-language-action models for robotics: A review towards real-world applications. *IEEE Access*, 2025.

Moo Jin Kim, Karl Pertsch, Siddharth Karamcheti, Ted Xiao, Ashwin Balakrishna, Suraj Nair, Rafael Rafailov, Ethan P Foster, Pannag R Sanketi, Quan Vuong, Thomas Kollar, Benjamin Burchfiel, Russ Tedrake, Dorsa Sadigh, Sergey Levine, Percy Liang, and Chelsea Finn. OpenVLA: An open-source vision-language-action model. In *Conference on Robot Learning*, 2024.

Jinhao Li, Yuxuan Cong, Yingqiao Wang, Hao Xia, Shan Huang, Yijia Zhang, Ningyi Xu, and Guohao Dai. STEP: Warm-started visuomotor policies with spatiotemporal consistency prediction. *arXiv preprint arXiv:2602.08245*, 2026.

Ping Li, Trevor J. Hastie, and Kenneth W. Church. Very sparse random projections. In *SIGKDD Conference on Knowledge Discovery and Data Mining*, 2006.

Anthony Liang, Yigit Korkmaz, Jiahui Zhang, Minyoung Hwang, Abrar Anwar, Sidhant Kaushik, Aditya Shah, Alex S. Huang, Luke Zettlemoyer, Dieter Fox, Yu Xiang, Anqi Li, Andreea Bobu, Abhishek Gupta, Stephen Tu, Erdem Biyik, and Jesse Zhang. Robometer: Scaling general-purpose robotic reward models via trajectory comparisons. In *Robotics: Science and Systems*, 2026.

Yaron Lipman, Ricky T. Q. Chen, Heli Ben-Hamu, Maximilian Nickel, and Matthew Le. Flow matching for generative modeling. In *International Conference on Learning Representations*, 2023.

Bo Liu, Yifeng Zhu, Chongkai Gao, Yihao Feng, qiang liu, Yuke Zhu, and Peter Stone. LIBERO: Benchmarking knowledge transfer for lifelong robot learning. In *Neural Information Processing Systems Datasets and Benchmarks Track*, 2023.

Xinyin Ma, Gongfan Fang, and Xinchao Wang. DeepCache: Accelerating diffusion models for free. In *Computer Vision and Pattern Recognition*, 2024.

Yecheng Jason Ma, Joey Hejna, Chuyuan Fu, Dhruv Shah, Jacky Liang, Zhuo Xu, Sean Kirmani, Peng Xu, Danny Driess, Ted Xiao, Osbert Bastani, Dinesh Jayaraman, Wenhao Yu, Tingnan Zhang, Dorsa Sadigh, and Fei Xia. Vision language models are in-context value learners. In *International Conference on Learning Representations*, 2025.

Yueen Ma, Zixing Song, Yuzheng Zhuang, Jianye Hao, and Irwin King. A survey on vision–language–action models for embodied AI. *IEEE Transactions on Neural Networks and Learning Systems*, 2026.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. *PyTorch: an imperative style, high-performance deep learning library*. 2019.

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

Yuchen Xia, Divyam Sharma, Yichao Yuan, Souvik Kundu, and Nishil Talati. MoDM: Efficient serving for image generation via mixture-of-diffusion models. In *International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, 2025.

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

Ling Yang, Zhilong Zhang, Yang Song, Shenda Hong, Runsheng Xu, Yue Zhao, Wentao Zhang, Bin Cui, and Ming-Hsuan Yang. Diffusion models: A comprehensive survey of methods and applications. *ACM Computing Surveys*, 2023.

Yantai Yang, Yuhao Wang, Zichen Wen, Luo Zhongwei, Chang Zou, Zhipeng Zhang, Chuan Wen, and Linfeng Zhang. EfficientVLA: Training-free acceleration and compression for vision-language-action models. In *Advances in Neural Information Processing Systems*, 2025.

Jingyi Zhang, Jiaxing Huang, Sheng Jin, and Shijian Lu. Vision-language models for vision tasks: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2024a.

Shiduo Zhang, Zhe Xu, Peiju Liu, Xiaopeng Yu, Yuan Li, Qinghui Gao, Zhaoye Fei, Zhangyue Yin, Zuxuan Wu, Yu-Gang Jiang, and Xipeng Qiu. VLABench: A large-scale benchmark for language-conditioned robotics manipulation with long-horizon reasoning tasks. *arXiv preprint arXiv:2412.18194*, 2024b.

Wenyao Zhang, Hongsi Liu, Zekun Qi, Yunnan Wang, XinQiang Yu, Jiazhao Zhang, Runpei Dong, Jiawei He, He Wang, Zhizheng Zhang, Li Yi, Wenjun Zeng, and Xin Jin. DreamVLA: A vision-language-action model dreamed with comprehensive world knowledge. In *Advances in Neural Information Processing Systems*, 2025.

A Ablation Study on Cache Key Source

Table 4: **Ablation of Cache Key Source.** Using VLM output embeddings as a cache key source outperforms the VLM input embeddings across various cache hit rates.

Source	VLM-in			**VLM-out**		
HR(%)	99.9	83.0	30.0	99.8	83.7	17.9
SR(%)	$7.3_{\pm 0.8}$	$8.1_{\pm 1.9}$	$19.7_{\pm 1.7}$	$\mathbf{34.7}_{\pm 3.9}$	$\mathbf{37.2}_{\pm 3.2}$	$\mathbf{39.7}_{\pm 3.1}$

ActionCache utilizes output embeddings of the VLM as an action-relevant cache key source. To validate this design choice, we investigate how the source of cache key impacts overall success rate on VLABench. As shown in Table 4, utilizing VLM output embeddings significantly outperforms the VLM input counterpart by up to 29.1% under roughly the same cache hit rate. This performance gap indicates that VLM outputs more effectively encode action-relevant multimodal context than VLM inputs.

B Case Study of Cross-Task Action Reuse

We conduct a case study to examine whether ActionCache can reuse actions beyond the task used for cache population.

To investigate the potential for cross-task action reuse, we conduct a zero-shot cross-task experiment on VLABench with $\pi_{0.5}$. In this experiment, we first populate the cache with the `select_fruit` task and then evaluate the model's performance on other target tasks. For more fine-grained analysis of action reuse across different tasks, we use a task progress score for evaluation, which reflects the completion of intermediate sub-goals within a task, rather than a binary success rate. During the evaluation of the target task, we disabled cache updates to ensure that any hits represent cross-task action sharing. The hit threshold T_{hit} is set to 0.925 during the target task evaluation. Results are aggregated over 120 episodes for each task.

Target Task	Progress Score (%)
`select_painting`	20.0 (base: 21.0)
`select_toy`	51.7 (base: 50.5)

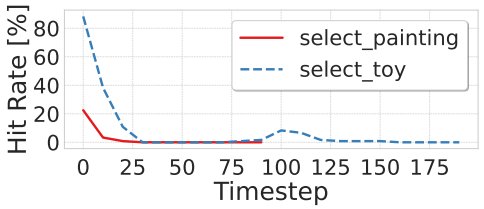


Figure 7: **Progress score on the target tasks and cache hit rate across rollout timesteps.** ActionCache maintains the base model's progress scores in unseen tasks. Cache hit rate reaches over 80% at the starting stage.

Progress Score and Hit Rate. Figure 7 shows that ActionCache maintains the base model's progress score while providing a meaningful cache hit rate at a certain phase. These results demonstrate that ActionCache can retrieve useful actions for tasks that were not used to populate the cache, while safely falling back to the original model when uncertain or unpopulated. Notably, the hit rate reaches over 80% at the initial stage of episodes in `select_toy` task. These results provide evidence that some reusable action patterns, particularly early-stage reaching motions, can transfer across related tasks. We do not interpret this experiment as demonstrating robustness to arbitrary task or environment shifts. Under distribution shifts that reduce cache-key similarity, ActionCache rejects more retrieved candidates, falls back to full-step generation, and adapts to it based on the cache replacement policy.

C Implementation Details for Simulation/Real-world Evaluations

C.1 Model Checkpoints

We utilized some publicly available model checkpoints for our experiments. For $\pi_{0.5}$, we used `VLABench/pi05-primitive-10task`[2] for finetuned checkpoints, and directly applied them to the simulation experiments. For GR00T-N1.6, we finetuned `nvidia/GR00T-N1.6-3B`[3] with the training scripts on the official repository[4], using a dataset: `VLABench/vlabench_primitive_ft_lerobot`[5]. We finetuned this model for 100,000 training steps with global batch size of 640, using 8 H100 GPUs. We also finetuned `lerobot/pi05_base`[6] for the real-world experiments.

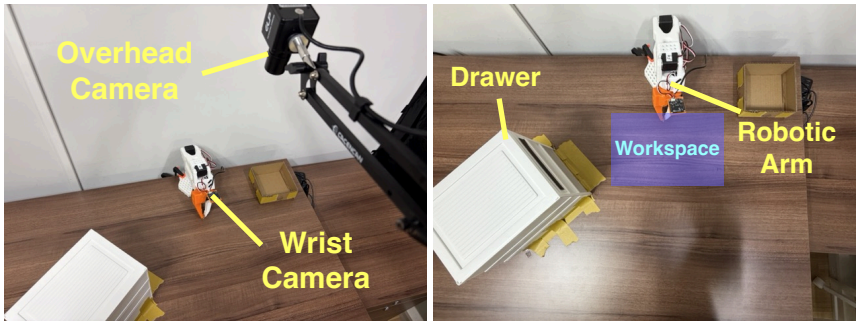


Figure 8: **Real-world Experimental Environment.** We use 6-DoF SO-101 robotic arm with an overhead and wrist-mounted camera.

C.2 Hyperparameter Settings

Table 5 summarizes the default hyperparameters for ActionCache specified in Section 4.1, and some other specific values used in experiments. Since VLM output embeddings of GR00T-N1.6 do not contain robot-state features, we utilized concatenation of VLM output and robot-state features as key source.

Among the evaluated hyperparameter configurations for Falcon (Chen et al., 2025), we select two points on the latency–success-rate Pareto frontier, one from the higher-latency region and one from the lower-latency region. These correspond to the first and second Falcon rows in Table 1 respectively.

C.3 Simulation Evaluation

Random Seed Configuration and Statistical Aggregation. For simulation experiments, we controlled two types of seeds: a task-setting seed and an action-noise sampling seed. Across all experiments on VLABench and LIBERO, the task-setting seed and episode ID uniquely determine the task configuration—including the target object, initial object poses, and instruction—ensuring that no evaluation episode exactly matches those used for cache prefill. We repeat each evaluation using four noise-sampling seeds, with 200 episodes per seed, and report the mean and standard deviation of the four resulting success rates in Table 1. Since the success rate is relatively insensitive to the noise-sampling seed and our goal is to obtain a coarse characterization of the latency–success-rate trade-off, we evaluate EfficientVLA and Falcon using a single noise-sampling seed, corresponding to 200 episodes in total.

[2] https://huggingface.co/VLABench/pi05-primitive-10task
[3] https://huggingface.co/nvidia/GR00T-N1.6-3B
[4] https://github.com/NVIDIA/Isaac-GR00T/tree/n1.6-release
[5] https://huggingface.co/datasets/VLABench/vlabench_primitive_ft_lerobot
[6] https://huggingface.co/lerobot/pi05_base

Table 5: **Default hyperparameters and model-specific implementation details of ActionCache.** Unless otherwise specified, these settings are used throughout our experiments.

Quantity	Symbol	$\pi_{0.5}$	GR00T-N1.6	Description
Cache key source	–	VLM output	VLM output + robot-state	Features used as input to the random projection.
Cache key source dimension	D	1,982,464	593,408	Dimension of the concatenated feature vector before random projection. (model specific)
Cache-key dimension	d	500	500	Dimension of the projected cache key.
Nonzero density	p	0.01	0.01	Fraction of nonzero entries in the sparse ternary random projection matrix.
Default hit threshold	T_{hit}	0.85	0.65	Minimum cosine similarity required to accept a retrieved cache entry.
Default cache capacity	C	3,000	3,000	Maximum number of cache entries.
Replacement policy	–	LRU	LRU	Default cache replacement policy.
Full-generation NFE	N	10	4	The official number of function evaluations.
Action chunk horizon	H	10	50	Number of actions generated in each action chunk.
Action execution horizon	–	10	10	Number of actions executed before replanning.

Evaluation Task on VLABench. For our experiments on VLABench, we used 10 primitive tasks of varying difficulty from VLABench, which are released as an official unified dataset on Hugging Face. All the task names are as follows: `add_condiment`, `insert_flower`, `select_book`, `select_chemistry_tube` (denoted as `select_tube` in Section 4.2), `select_drink`, `select_fruit`, `select_mahjong`, `select_painting`, `select_poker`, and `select_toy`.

C.4 Real-world Training and Evaluation

Real-world Environment. As depicted in Figure 8, we use an overhead camera and a wrist-mounted camera with a resolution of 480×640, resized to 224×224 before inputting them to the model.

Detailed Task Description. We defined three tasks: `pick_sausage`, `pick_and_close` and `push_button`. In `pick_sausage`, the robot picks up a toy sausage from workspace containing multiple toy foods, and place it on a fixed pan. Two distractor objects are placed to assess the robustness and spatial understanding of ActionCache in perturbations of object layout. In `pick_and_close`, the robot picks up a white cube, put it inside a drawer, and close the drawer. This task provides a multi-step and long-horizon evaluation. In `push_button`, three colored buttons (red, green, and blue) are placed in a randomized spatial arrangement, and the robot is instructed to press the button of the specified color. This task evaluates the model's instruction following and visual/linguistic grounding ability.

Training Dataset Construction and Task Settings. To construct fine-tuning datasets for each task, we record 100 successful episodes and 50 additional episodes consisting of recovery trajectories from near-failure states.

For `pick_and_close` task, we predetermined five fixed locations to place a white cube and record 20 successful demonstrations per location to ensure comprehensive spatial coverage. For each lo-

cation, we add rotational perturbations to the cube; $0°$, $\pm 30°$ and $\pm 45°$. In each evaluation episode, one of the five locations is uniformly selected and the same rotational perturbations are applied. A trial is deemed successful if the robot put the cube into the drawer and close it.

For `pick_sausage` task, we randomly choose the location of the sausage and add rotational perturbations; $0°$, $45°$, $90°$, $135°$ and $180°$. The location of the other distractor objects are also randomly selected. The task setting on evaluation is largely consistent with that used for fine-tuning dataset collection; however, object positions are randomized, and none of the evaluated configurations exactly matches a configuration included in the fine-tuning dataset. A trial is deemed successful if the robot put the sausage on the pan.

For `push_button` task, three colored buttons (red, green, and blue) are placed at three predefined positions in front of the robotic arm. At the beginning of each trial, the assignment of colors to these positions is uniformly randomized, and the target button color specified in the language instruction is also sampled uniformly at random. A trial is deemed successful if the robot presses the button whose color matches the instruction.

Along with the success rate, we measure the inference wall-clock time to accurately evaluate the latency-performance trade-off. We implemented our evaluation code with PyTorch.

Training Budget. For each task, we fine-tuned $\pi_{0.5}$ for $5{,}000$ steps with a global batch size of 128. We used LeRobot framework with 4 H100 (96GB memory) GPUs for fine-tuning.

Evaluation Settings. Before success rate measurement, we prefill the cache with $T_{\text{hit}} = 1$. Once the cache becomes full, we seamlessly move on to the final evaluation with T_{hit} value predetermined for each task. Specifically, we set $T_{\text{hit}} = 0.875$ for the `pick_sausage` task to maintain base model's success rate, and $T_{\text{hit}} = 0.85$ for the others. Since the `push_button` task is relatively short-horizon, we set the cache size to 300 instead of $1{,}000$ for the other tasks. We set both the action chunk and execution horizon to 50, since it is the default value for the base model (`lerobot/pi05_base`). Each success rates and latencies reported in Table 2 is aggregated over 50 episodes.

Success Verification and Cache Update. During our real-world evaluation, the cache was updated online using human-verified episode success/failure. Success was determined solely by whether the predefined final task state described in task settings was achieved. Note that this success signal is required only for cache admission and does not lie on the action-generation critical path. In practical deployments, it may be provided by task-specific completion signals, an external success detector (e.g., SuccessVQA (Du et al., 2023), GVL (Ma et al., 2025), or ROBOMETER (Liang et al., 2026)), or an operator. When successful trajectories with known outcomes are already available, e.g., from policy fine-tuning, the same cache-admission procedure can also be applied offline before deployment, while we did not do that as we used disjoint task-setting seeds to avoid overlap between cache construction and evaluation episodes.

C.5 VLA-Cache Configuration and Evaluation Settings

Table 6: **Hyperparameter settings used for VLA-Cache.** Default values from the official VLA-Cache GitHub repository are shown in parentheses.

Hyperparameter	Value
Threshold of patch-wise cosine similarity	0.996 (0.996)
Token pruning layer locations	2, 6, 9, 11, 13, 15 (2, 6, 9, 11)
Reuse-ratio schedule growth factor	1.00 (0.55)

For evaluation of ActionCache with VLA-Cache in Section 4.6, we use the hyperparameter settings shown in Table 6. The default settings yielded little VLM speedup, likely due to differences in the model and benchmark; we therefore use more aggressive visual token reuse parameters that still preserve the success rate.

C.6 Latency Measurement

In both simulation and real-world experiments, we used either JAX (Bradbury et al., 2018) or PyTorch (Paszke et al., 2019) implementation. For $\pi_{0.5}$, the latencies in Table 1 are measured using a JAX implementation, while those in other experiments and GR00T-N1.6 are measured using a PyTorch implementation. In our setup, the PyTorch implementation is less optimized for inference than the JAX implementation, which results in the higher baseline latencies in these experiments.

D Hyperparameter Sensitivity Analysis

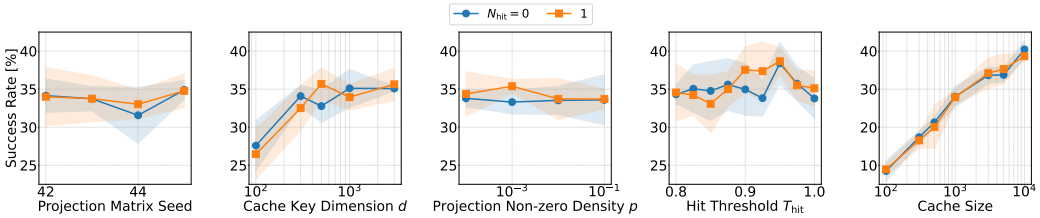


Figure 9: **Task success rates for hyperparameter changes.** Only one parameter varies and the others are fixed. Cache key dimension d and cache size mainly affect the success rate, while the others do not.

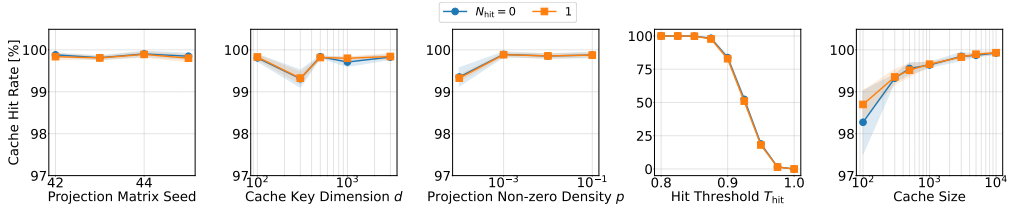


Figure 10: **Cache hit rates for hyperparameter changes.** Only one parameter varies and the others are fixed. The hit threshold T_{hit} mainly affects the cache hit rate.

We analyze how hyperparameters affect the task success rates and the cache hit rates of ActionCache. Specifically, we vary only one hyperparameter among the generation seed of the sparse ternary random matrix, key dimension d, non-zero value density of the sparse ternary random matrix p, hit threshold T_{hit} and maximum cache size. The other hyperparameters are fixed to their default value as specified in Section 4.1. To focus on the low-NFE regime, we set $N_{\text{hit}} \in \{0, 1\}$. All experiments are conducted on VLABench with $\pi_{0.5}$.

Figure 9 shows the success rates for each hyperparameter variations. We observe that the overall success rate heavily depends on the cache size. Notably, cache size of $10,000$ reaches the success rate of the full-step $\pi_{0.5}$ model. This result suggests that the cache size is the most important factor in our framework. The cache key dimension affects the success rate only when it is too small, but we found that $d = 300$ is sufficient. In contrast, the generation seed and non-zero value density of the key projection matrix have little effect on the success rate within the evaluated range. These results suggest that we can achieve high performance with relatively small values of d and p, resulting in negligible computational and memory overhead of the key projection in ActionCache. The hit threshold T_{hit} also has little effect in this evaluation setting. This is likely because, with sufficient cache size ($3,000$ for this time), the quality of a Top-1 candidate becomes high enough on average, making the success rate largely insensitive to the choice of T_{hit}.

Figure 10 shows the cache hit rate for each hyperparameter variations. We observe that all hyperparameters except for T_{hit} have little effect on the cache hit rate, which remains close to 100% across the evaluated range. On the other hand, T_{hit} dominates the cache hit rate; the hit rate monotonically decreases from nearly 100% to 0% as T_{hit} increases. Since the cache hit rate directly affects the average inference latency in this framework, T_{hit} plays an important role in the latency–success-rate trade-off.

Table 7: **Success rate comparison of $\pi_{0.5}$ and ActionCache on LIBERO.**

Method	NFE	Success Rate (%)					Hit Rate	Latency (ms)	
		Spatial	Object	Goal	Long	Avg.	(%)	Act. Head	Overall
Base Model	10	97.4	**99.0**	96.6	95.4	**97.1**	–	54.07	101.80
	1	**97.6**	98.0	97.4	94.6	96.9	–	5.77	53.48
ActionCache	2	96.0	98.4	**97.6**	**96.0**	97.0	92.6	14.25	62.08
	1	96.4	**99.0**	96.2	93.6	96.3	91.9	11.31	66.41
	0	89.4	**99.0**	96.4	83.4	92.1	87.7	6.57	54.44

E Warmup Episode Counts

For all experiments in this paper, we warmup the cache with $T_{\text{hit}} = 1$, which is equivalent to running the base model except for the negligible latency overhead of key construction and cache insertion. The cache can therefore be populated during normal base-model operation, without additional costs, dedicated warmup deployment, or success rate drop.

Since only successful episodes are added to the cache, the expected number of episodes needed to fill the cache is approximately

$$N_{\text{warmup}} \approx \frac{C}{\bar{n}_{\text{succ}} s_{\text{base}}},$$

where C is the cache size, $\bar{n}_{\text{succ}}$ is the mean number of action chunks per successful rollout, and $s_{\text{base}} \in (0, 1]$ is the base model's success rate. Because failed episodes contribute no cache entries, the lower base-model success rates on VLABench leads to the larger overall warmup episode counts relative to LIBERO-Long. Across the evaluated settings, however, warmup completes with an average of 11.6–26.7 successful episodes per task, well below the per-task episode counts in fine-tuning datasets for both simulation and real-world settings (see Appendix C). These results show that successful experience on the scale of a fine-tuning dataset is sufficient to populate the caches supporting the reported acceleration.

F Key Similarity Analysis in Real-world

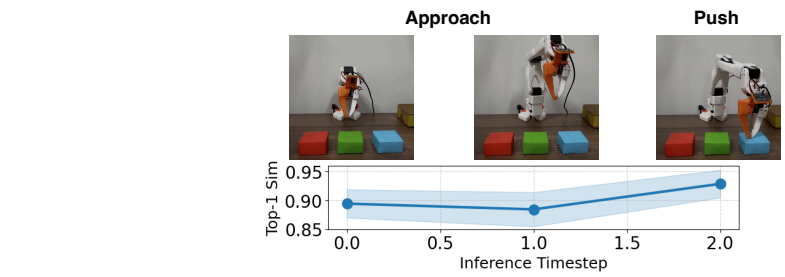
We analyze the temporal evolution of the top-1 cache key similarity in real-robot experiments. Using all data from the real-robot experiments in Section 4.5, we compute the mean and standard deviation of the top-1 similarity at each inference timestep over successful episodes. Failed episodes are excluded because they are substantially longer and too few to provide reliable statistics. Figure 11 shows the temporal evolution of the top-1 similarity for each task, together with intermediate images corresponding to selected timesteps.

For `pick_sausage` and `pick_and_close`, the similarity tends to decrease during phases such as approaching, carrying, and the transition from placing to closing. These phases involve substantial changes in the camera observation, suggesting that the cache keys are sensitive to changes in visible objects and background appearance. In contrast, high similarities are observed near the beginning of episodes and during actions shared across many episodes, such as placing, closing, and pushing. This indicates that ActionCache successfully retains and reuses such common action patterns across episodes.

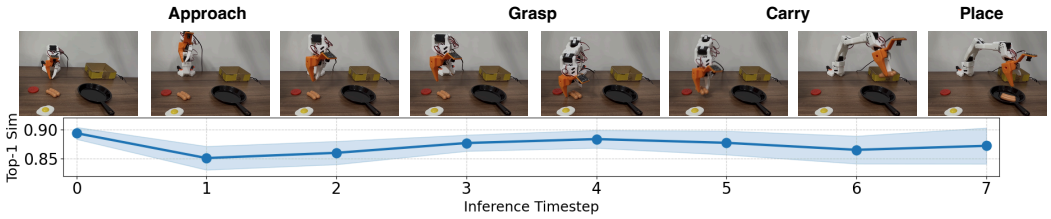
G Results for LIBERO benchmark

LIBERO consists of four task suites (Spatial, Object, Goal and Long) which contain 10 tasks each.

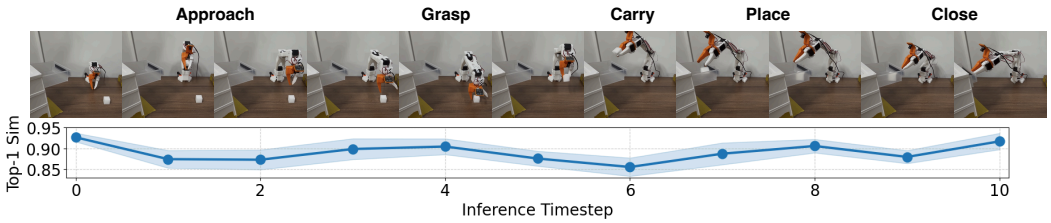
Evaluation Settings. We evaluate $\pi_{0.5}$ on LIBERO benchmark and report its success rate for each task suite at NFE $= 10$ and 1 for the base model, and NFE $= 0 \sim 2$ for ActionCache. We also report cache hit rate and average latency of action head. The reported latency is measured on PyTorch implementation, using NVIDIA RTX 5090 GPU with 32GB of memory. For all task suites, we set the cache size to 10,000 and $T_{\text{hit}} = 0.85$. In this experiment, we leveraged an open weighted



(a) `push_button`



(b) `pick_sausage`



(c) `pick_and_close`

Figure 11: **Visualization of Top-1 similarity and task phase for the three manipulation tasks.**

fine-tuned checkpoint (`lerobot/pi05_libero_finetuned_v044`[7]) from Huggingface, and used LeRobot framework to conduct the experiment. The success rates are the mean value on 500 episodes for each task suites and NFE settings.

Results. As shown in Table 7, we find that the base model retains nearly full success rate even at NFE $= 1$, indicating that one-step inference is already sufficient in this model-benchmark setting. The modest drop of ActionCache at NFE $= 0$ likely reflects LIBERO's sensitivity to small action errors caused by slight mismatches between retrieved and correct ones, as empirically observed in failure cases. However, its near-baseline success rate at NFE $= 1$ indicates that even one refinement step is sufficient to mitigate such errors. Together with the VLABench results in Table 1, where direct NFE reduction substantially degrades the success rate, these results show that ActionCache can accelerate inference while maintaining accuracy across both NFE-insensitive and NFE-sensitive tasks.

[7] https://huggingface.co/lerobot/pi05_libero_finetuned_v044