Dejavu: Towards Experience Feedback Learning for Embodied Intelligence Appendix

A. Overview

This appendix is organized as follows. Section B describes the detailed settings of our method, baselines, and experiments. Section C elaborates on the rationale, motivation, and intuition behind EFN. Section D discusses the method and its broader implications. Section E presents supplementary experimental results and analysis. Section F provides additional visualizations of EFN in action.

B. Details of Method and Experiment

This section provides implementation and experimental details; the rationale behind these design choices is discussed in Section C.

B.1. Experience Bank Implementation Details

Step-level representation. Recall that EFN operates on step-level transitions from rollouts $\tau = (s_1, a_1, \ldots, s_T, a_T)$ and augments a frozen VLA policy using an experience bank. Conceptually, each stored step $E_{m,n}$ from rollout m at time n contains

$$E_{m,n} = \big(\ell_{\tau_m},\, \mathbf{F}_{m,n},\, \mathbf{k}_{m,n},\, \mathbf{a}^{(0)}_{m,n}\big), \tag{B.1}$$

where ℓ_{τ_m} is the instruction embedding for rollout τ_m, $\mathbf{F}_{m,n} \in \mathbb{R}^{L\times C}$ is the VLA vision encoder feature map for frame $s_{m,n}$, $\mathbf{k}_{m,n} \in \mathbb{R}^{d_k}$ is the mean–max fused retrieval key defined in main-paper Sec. 3.2, and $\mathbf{a}^{(0)}_{m,n}$ is the raw action produced by the frozen base policy at that step.

In our implementation, the bank is stored as a line-based JSON file together with accompanying feature files, where each JSON line encodes one step and includes a rollout identifier, a step index, and paths to the visual and action features. The corresponding visual feature file stores the precomputed retrieval key $\mathbf{k}_{m,n}$ as a d_k-dimensional vector (e.g., `visual_embed_meanmax` with $d_k = 4096$), together with any additional visual features that may be needed. The action file stores the base policy's latent action representation (e.g., a $4\times d_a$ tensor of action tokens) and the associated discrete token IDs (`generated_ids`) from which the continuous control command $\mathbf{a}^{(0)}_{m,n}$ is decoded.

Rollout-wise organization and indexing. Experiences are inserted into the bank at every non-blank step where the robot executes a control action. We explicitly omit the initial waiting steps during simulator loading and robot warm-up from both training and evaluation; these steps are never written to the JSON file and therefore never appear as candidates in retrieval.

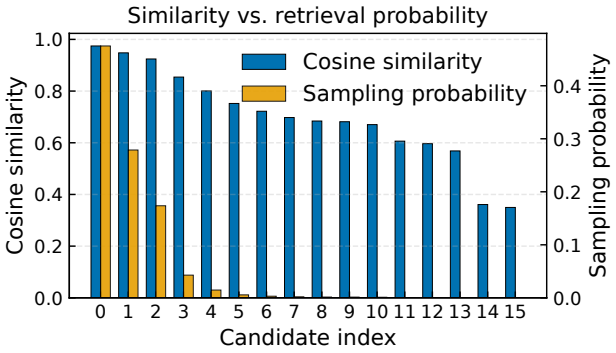


Figure B.1. Cosine similarities (left bars) and corresponding softmax sampling probabilities (right bars) over the top candidates in the experience bank. Higher similarity induces higher sampling probability while still preserving exploration.

The JSON file implicitly groups steps by `rollout_id` and `step_idx`. At load time, we construct two data structures: (i) a mapping

$$\texttt{index} : (\texttt{rollout\_id}, \texttt{step\_idx}) \mapsto E_{m,n}, \tag{B.2}$$

implemented as a hash table from the pair (m, n) to the corresponding metadata record, and (ii) a dense key matrix

$$\mathbf{K} \in \mathbb{R}^{N\times d_k}, \tag{B.3}$$

obtained by stacking all keys $\mathbf{k}_{m,n}$ for the N stored steps. This enables constant-time lookup of a particular step and efficient vectorized similarity computation against all keys.

For convenience, we also expose a "successor" operator $E_{m,n+1}$ that returns the immediate next step of a given transition whenever it exists:

$$E_{m,n+1} = \texttt{index}\big(\texttt{rollout\_id} = m,\ \texttt{step\_idx} = n+1\big). \tag{B.4}$$

We use $E_{m,n+1}$ as the semantic target successor frame when shaping the progress reward: the retrieved step provides a reference for "what should happen next" if the agent were to follow a previously successful trajectory.

Retrieval over the bank. Given a current observation s_t, we compute the vision encoder feature map $\mathbf{F}_t$ and derive a query key $\mathbf{q}_t$ via the same mean–max fusion and ℓ_2-normalization used for $\mathbf{k}_{m,n}$ (main-paper Sec. 3.2). We then compute cosine similarities between $\mathbf{q}_t$ and all stored keys:

$$s_i = \cos(\mathbf{q}_t, \mathbf{k}_i), \quad i \in \{1, \dots, N\}. \tag{B.5}$$

In the probabilistic variant used for EFN training, we follow the top-k sampling scheme described in the main text: we first select a shortlist $\mathcal{N}_k(\mathbf{q}_t)$ of the k highest-similarity keys and sample an index using a temperature-scaled softmax,

$$p(i \mid \mathbf{q}_t) = \frac{\exp\left(s_i/\tau\right)}{\sum_{j \in \mathcal{N}_k(\mathbf{q}_t)} \exp\left(s_j/\tau\right)}, \quad i \in \mathcal{N}_k(\mathbf{q}_t), \tag{B.6}$$

with a fixed temperature $\tau > 0$ (we use $\tau = 0.05$ in all experiments). As shown in Figure B.1, this "retrieve-then-sample" design encourages exploration among near-matches while still biasing toward the most semantically similar experiences.

Usage during training and deployment. During offline pretraining of EFN, the experience bank is constructed from a mixture of successful, near-successful, and occasionally failed rollouts, as discussed in Appendix D. At each environment step, EFN queries the bank, obtains a reference step $E_{m,n}$ and its successor $E_{m,n+1}$, and uses the successor observation as the semantic target for the similarity-shaped progress reward. At deployment, the bank continues to grow: we append only the trajectories that reach the goal, thereby gradually refining the set of available references without ever updating the frozen VLA backbone weights.

Even failed or near-success rollouts can still contribute meaningful similarity structure, since the reward is defined on feature transitions rather than binary outcome labels (see Sec. D.1 for analysis).

B.2. Reward Shaping and SAC Training Details

This subsection details the implementation of the similarity terms in main-paper Eq. (13) and the Soft Actor–Critic (SAC) [5] training procedure used for EFN.

Similarity function $\mathrm{sim}(\cdot,\cdot)$**.** For reward shaping, we instantiate $\mathrm{sim}(\cdot,\cdot)$ as a cosine-based similarity on the per-patch vision tokens. Given two feature maps $X, Y \in \mathbb{R}^{256\times4096}$ from the frozen VLA backbone, we first ℓ_2-normalize each token along the channel dimension and form the cosine matrix

$$\begin{aligned} S &= \text{token\_cosine\_matrix}(X, Y) \\ &= \mathrm{norm}(X)\,\mathrm{norm}(Y)^\top \in [-1,1]^{256\times256}. \end{aligned} \tag{B.7}$$

We then compute a Sinkhorn-regularized matching [3] between the two token sets with a Gibbs kernel $K = \exp(S/\varepsilon_{\mathrm{OT}})$, uniform row and column marginals, and n_{iter} Sinkhorn iterations. The resulting transport plan $P \in \mathbb{R}^{256\times256}$ satisfies approximately uniform marginals and we define the scalar score

$$\mathrm{score}(X, Y) \;=\; \sum_{i,j} P_{ij}\, S_{ij}, \tag{B.8}$$

followed by a linear rescaling to $[0, 1]$:

$$\mathrm{sim}(X, Y) \;=\; \tfrac{1}{2}\left(\mathrm{clip}(\mathrm{score}(X, Y), -1, 1) + 1\right). \tag{B.9}$$

In all experiments, we set $\varepsilon_{\mathrm{OT}} = 0.05$ and $n_{\mathrm{iter}} = 50$. No additional learned projection is applied before computing similarity; we rely on the backbone's vision tokens and Sinkhorn normalization to provide a stable semantic similarity in $[0, 1]$. The same primitive $\mathrm{sim}(\cdot,\cdot)$ is used for $s_t^{\mathrm{next}}, s_t^{\mathrm{cur}}, s_t^{\mathrm{stay}}$ in main-paper Eq. (13).

Reward decomposition and default weights. Given the three similarities

$$\begin{aligned} s_t^{\mathrm{next}} &= \mathrm{sim}(\mathbf{F}_{t+1}, \hat{\mathbf{F}}^{+}), \\ s_t^{\mathrm{cur}} &= \mathrm{sim}(\mathbf{F}_t, \hat{\mathbf{F}}), \\ s_t^{\mathrm{stay}} &= \mathrm{sim}(\mathbf{F}_{t+1}, \mathbf{F}_t). \end{aligned} \tag{B.10}$$

we define the auxiliary scalars

$$a_t = s_t^{\mathrm{next}}, p_t = s_t^{\mathrm{next}} - s_t^{\mathrm{cur}}, m_t = 1 - s_t^{\mathrm{stay}}, n_t = [\,\varepsilon - p_t\,]_+, \tag{B.11}$$

with tolerance $\varepsilon > 0$ and $[x]_+ = \max(x, 0)$. The shaped reward in main-paper Eq. (13) is then

$$\begin{aligned} r_t \;=\; & w_{\mathrm{abs}}\, a_t + w_{\mathrm{prog}}\, [p_t]_+ + w_{\mathrm{mot}}\, m_t \\ & - \; w_{\mathrm{lazy}}\left(s_t^{\mathrm{next}}\, n_t\, s_t^{\mathrm{stay}}\right) - \lambda_{\mathrm{time}}, \end{aligned} \tag{B.12}$$

which is finally clipped to $[-1, 1]$ for numerical stability. Unless otherwise stated we use

$$w_{\mathrm{abs}} = 0.1, w_{\mathrm{prog}} = 20, w_{\mathrm{mot}} = 0.3, w_{\mathrm{lazy}} = 20, \tag{B.13}$$

$$\varepsilon = 5 \times 10^{-3}, \quad \lambda_{\mathrm{time}} = 0.01. \tag{B.14}$$

These values give a reward distribution with non-trivial variance while keeping r_t in a moderate range for SAC. In practice, we tune the weights as follows. We first fix w_{abs} to a small constant that preserves the absolute similarity bonus without dominating the other terms, then adjust w_{prog} so that genuine progress toward the retrieved successor frame is reliably rewarded. If the agent tends to move without approaching the target view, we slightly reduce w_{prog} and increase w_{mot}. If the policy learns to idle at viewpoints that already match the target frame, we increase w_{lazy} (and,

if necessary, ε) to strengthen the anti-idling penalty. The per-step time cost λ_{time} is kept small and is only increased when trajectories become unnecessarily long. The residual ℓ_2 penalty from the semantic reward in main-paper Sec. 3.3 is disabled in our final experiments, that is, we set $\lambda_{\text{res}} = 0$ and let the tanh-squashed residual range control the magnitude of $\Delta\mathbf{a}_t$.

SAC hyperparameters. We use a standard off-policy Soft Actor–Critic setup on top of the shaped reward in Eq. (B.12). The discount factor is $\gamma = 0.98$. The SAC temperature α is adapted online to match a target entropy. We maintain $\log\alpha$ as a learnable scalar and update it with

$$\mathcal{L}_\alpha = -\big(\log\alpha\big)\,\big(\log\pi_\phi(\Delta\mathbf{a}_t|\mathbf{c}_t) + \mathcal{H}_{\text{target}}\big), \tag{B.15}$$

where $\mathcal{H}_{\text{target}}$ is proportional to the dimensionality of the residual action. Since $\Delta\mathbf{a}_t$ lives in a 4×4096 latent space, we set

$$\mathcal{H}_{\text{target}} = -\kappa\cdot(4\cdot4096), \quad \kappa = 0.25, \tag{B.16}$$

which encourages a moderately stochastic residual policy while still exploiting the shaped reward.

Both actor and critics are trained with Adam optimizers [7]. We use learning rates

$$\text{lr}_{\text{actor}} = 3\times10^{-4}, \text{lr}_{\text{critic}} = 3\times10^{-4}, \text{lr}_\alpha = 3\times10^{-4}, \tag{B.17}$$

with default Adam momentum parameters and no weight decay. The replay buffer capacity is 2×10^5 transitions, and we start SAC updates once the buffer contains at least one mini-batch. Each update samples a batch of size $B = 32$ and performs one critic update, one actor update, one temperature update, and a soft target update. The target critic $Q_{\bar\theta}$ is updated by Polyak averaging with coefficient

$$\tau = 0.005, \quad \bar\theta \leftarrow (1-\tau)\,\bar\theta + \tau\,\theta \tag{B.18}$$

after every gradient step. The discount γ and τ are kept fixed across all experiments, and we verify that moderate changes around these values have little qualitative effect on EFN's behavior.

Training loop. Algorithm B.1 summarizes the training loop that combines retrieval, reward shaping, and SAC. We emphasize that the frozen VLA backbone and experience bank are treated as environment-side components; gradients never flow into the backbone or the bank.

This procedure realizes the conceptual objective in main-paper Sec. 3.3: EFN learns a residual policy that steers $\mathbf{F}_{t+1}$ toward the retrieved successor frame $\hat{\mathbf{F}}^+$ while discouraging degenerate idling and overly long trajectories.

Algorithm B.1 EFN training with reward shaping and SAC

Initialize frozen VLA policy π_0, experience bank $\mathcal{B}$, residual actor π_ϕ, critics $Q_{\theta_1}, Q_{\theta_2}$, target critic $Q_{\bar\theta_i}$, temperature α, and replay buffer $\mathcal{D}$.
for each training epoch **do**
  Reset environment and obtain initial observation s_0; extract vision features $\mathbf{F}_0$ and base action $\mathbf{a}_0^{(0)}$ from π_0.
  while episode not terminated **do**
    Query the experience bank with ($\mathbf{F}_t$, instruction) to retrieve $(\hat{\mathbf{F}}, \hat{\mathbf{a}}, \hat{\mathbf{F}}^+, \ell)$.
    Form context $\mathbf{c}_t = \text{enc}(\mathbf{F}_t, \mathbf{a}_t^{(0)}, \hat{\mathbf{F}}, \hat{\mathbf{a}}, \ell)$ and sample residual $\Delta\mathbf{a}_t \sim \pi_\phi(\cdot|\mathbf{c}_t)$.
    Execute corrected action $\mathbf{a}_t = \mathbf{a}_t^{(0)} + \Delta\mathbf{a}_t$ in the environment and observe next state s_{t+1}, features $\mathbf{F}_{t+1}$, and done flag d_t.
    Compute similarities
    $$s_t^{\text{next}} = \text{sim}(\mathbf{F}_{t+1}, \hat{\mathbf{F}}^+),\ s_t^{\text{cur}} = \text{sim}(\mathbf{F}_t, \hat{\mathbf{F}}),\ s_t^{\text{stay}} = \text{sim}(\mathbf{F}_{t+1}, \mathbf{F}_t),$$
    then construct a_t, p_t, m_t, n_t and reward r_t according to Eq. (B.12).
    Store transition $(\mathbf{c}_t, \Delta\mathbf{a}_t, r_t, \mathbf{c}_{t+1}, d_t)$ in $\mathcal{D}$.
    if $|\mathcal{D}| \geq B$ **then**
      Sample a mini-batch from $\mathcal{D}$ and compute critic targets y_t with the soft Bellman backup using $Q_{\bar\theta_i}$ and α as in main-paper Sec. 3.3.
      Update critics by minimizing $\mathcal{L}_{\text{critic}}$ and update actor by minimizing $\mathcal{L}_{\text{actor}}$.
      Update temperature α toward the target entropy and apply Polyak averaging to refresh $Q_{\bar\theta_i}$.
    end if
    Set $t \leftarrow t+1$.
  end while
end for

B.3. EFN Architecture and Implementation

EFN operates on the latent interface exposed by the frozen vision–language–action (VLA) backbone. At each non-blank step, the backbone provides a set of visual patch embeddings of shape $(256, 4096)$ and a small number of latent action tokens of shape $(4, 4096)$. For the current state and the retrieved experience state, EFN receives both the visual embeddings and the latent tokens and predicts a residual in the same latent space, which is added to the backbone action tokens. All EFN parameters are trained while the backbone remains fixed, so only this residual branch is updated.

To keep the architecture lightweight yet expressive, we first compress the long visual sequence into a few learned latent vectors. This is implemented with a Multihead Attention Pooling (MAP) block inspired by Set Transformers and UniVLA [2]. A bank of learned seed vectors plays the role of latent queries, while the per-patch visual embeddings act as keys and values. A custom attention module projects seeds and visual inputs into multihead query, key and value tensors, performs scaled dot-product atten-

Table B.1. Per-step latency of EFN variants. The full actor–critic configuration is used as reference.

Variant	Latency (ms / step)↓	Δ vs full EFN
policy+critic, 2-layer enc.	48.96	0.00%
policy only, 2-layer enc.	27.89	−43.04%
policy only, 1-layer enc.	26.01	−46.89%
policy only, 2-layer enc., 1536-d	43.48	−11.21%

tion, and projects the attended seeds back to the embedding dimension. Each MAP block is wrapped with RMS normalization and a position-wise feed-forward network. The feed-forward network uses a SwishGLU unit that applies a linear projection to twice the hidden size, splits it into a value and a gate, modulates the value with a SiLU-activated gate, and maps back to the embedding dimension. This yields compact latent representations of the current and retrieved views instead of operating directly on all 256 visual patches.

The EFN policy network uses two MAP blocks to obtain visual latents for the current and retrieved states, each with four latent vectors of dimension 1024. The current and retrieved latent action tokens are projected into the same 1024-dimensional space. For cross-conditioning, EFN concatenates the pooled current visual latents, pooled retrieved visual latents, and retrieved latent tokens along the sequence dimension, yielding a context sequence of length 12. The projected current latent tokens serve as queries. A multihead attention layer with layer-normalized queries, keys, and values computes an attended representation of the four query tokens over this 12-token context. A small feed-forward block with normalization and dropout refines these tokens, and a two-layer Transformer encoder with pre-normalization further processes the four-token sequence to capture interactions among the latent dimensions. A final LayerNorm and a three-layer MLP map each token back to the original 4096-dimensional latent space, with a tanh activation and explicit clipping to keep the residual inside a controlled range. The output is a residual tensor of shape $(4, 4096)$ that is added to the backbone latent action tokens. At deployment, we run this policy branch in inference mode and feed the refined latent tokens to the frozen VLA head without any further gradient updates.

For actor–critic training we pair the policy with a critic that shares the same inputs but has a simpler structure. The critic uses MAP blocks with a single latent per view to obtain global visual summaries for the current and retrieved states, and uses mean pooling over the four latent tokens followed by a linear projection to obtain compact token summaries. The four resulting vectors are concatenated into a single feature of size 4×1024, normalized with a float32 layer normalization for numerical stability under mixed precision, and passed through a multi-layer perceptron that outputs a scalar value. The critic is used only during training to provide value estimates for the Soft Actor–Critic objective; at test time only the policy branch is active.

To quantify the overhead introduced by EFN itself, we benchmark several variants of the residual module under the same latent interface. Table B.1 reports per-step latency for the full actor–critic configuration and for lighter policy-only variants. Removing the critic already reduces latency by more than 40%, and a one-layer encoder achieves the lowest cost while preserving the interface, which confirms that EFN adds only a modest runtime overhead on top of the frozen backbone.

B.4. Simulation Setup and Baseline Configurations

We now detail the simulation benchmarks, backbone policies, evaluation protocol, and baseline implementations used in our experiments.

Benchmarks. We primarily evaluate on the LIBERO long-horizon manipulation benchmark [8], which provides language-conditioned tabletop tasks in a simulated kitchen. Following the official protocol, we use the four standard suites: *LIBERO-Spatial*, *LIBERO-Object*, *LIBERO-Goal*, and *LIBERO-Long*. Each suite groups tasks that emphasize spatial reasoning, object-centric manipulation, goal-directed rearrangement, and extended multi-step interactions, respectively. As reported in Table 1 in the main paper, our EFN variants consistently achieve the highest average success rate across all four suites under the same backbone and interaction budget.

Backbone policies. Across all experiments, EFN wraps a frozen vision–language–action backbone. For the LIBERO simulations, we use the officially released checkpoints and configurations of **OpenVLA** [6] and **UniVLA** [2] without architectural modifications. For the real-world experiments, we employ the publicly released **AgiBot GO-1** [1] controller as our backbone and keep its perception and low-level control stack unchanged. Its performance on LIBERO is reproduced by strictly following the official README instructions (training script, hyperparameters, and evaluation protocol); we do not perform additional tuning specific to our method, and all baselines share the same backbone initialization.

Evaluation protocol. Unless otherwise stated, all reported numbers are averaged over **50 evaluation episodes** per task and **3 random seeds** (i.e., 150 rollouts per cell in Table 1 in the main paper). During evaluation, policies act deterministically by using the mean action of the underlying stochastic policy (SAC) and the deterministic retrieval rule of each method. An episode is marked as *Success* if the environment-specific success flag is triggered before the task horizon; otherwise it is counted as a failure. The *Step* metric

counts the number of control steps from the first *non-blank* action until success or timeout, where we ignore the initial blank warm-up steps (e.g., simulator loading and robot settling) that occur before the policy starts issuing meaningful actions.

Having fixed the benchmarks, backbones, and evaluation protocol, we now describe the baseline families that we compare EFN against. All baselines share the same frozen VLA backbone, experience bank (when applicable), and interaction budget as EFN.

B.4.1. kNN-RAG (nearest-neighbor retrieval).

We implement a non-parametric kNN-RAG baseline that controls the agent purely by retrieving and reusing past actions from the experience bank, without learning any residual policy. At each non-blank step, we encode the current observation and language instruction with the same frozen VLA backbone used by all other methods and obtain a joint visual–language key. This key is compared to all step-level keys stored in the experience bank (main-paper Sec. 3.2) using cosine similarity, and we select the single nearest neighbor in this embedding space. The action stored in that retrieved transition is then directly executed as the agent's action at the current step, i.e., the controller simply "copies what worked before" for the most similar past state.

In contrast to residual methods, kNN-RAG does not introduce any additional trainable parameters and does not update the backbone or learn an adaptation head; its behavior is fully determined by the fixed VLA encoder, the similarity metric, and the contents of the experience bank. This makes kNN-RAG a strong retrieval-only baseline: it benefits from semantic retrieval over rich past trajectories, but lacks the capacity to adjust or interpolate actions when the retrieved state is only approximately similar to the current situation, exposing the limitations of naive action copying from memory.

B.4.2. ResAct-style residual RL baseline.

We further implement a ResAct-style residual RL baseline that adapts the core ideas of ResAct [9], namely observation-difference features and action residuals, to our VLA setting. To keep the comparison fair, this baseline shares the same frozen vision–language–action backbone and residual RL infrastructure as our other methods. At each time step t, we obtain a visual embedding $f_{\text{vis}}(o_t)$ from the VLA's visual tokens (e.g., pooled over patches), along with proprioceptive features $f_{\text{prop}}(x_t)$ and an instruction or goal embedding $e(\ell)$. In addition, we cache the previous-step visual embedding $f_{\text{vis}}(o_{t-1})$ and executed action a_{t-1}, and form an observation-difference feature

$$d_t = f_{\text{vis}}(o_t) - f_{\text{vis}}(o_{t-1}), \tag{B.19}$$

optionally passed through a small MLP and LayerNorm for dimensionality reduction and stabilization. The resulting ResAct state is the concatenation

$$s_t^{\text{ResAct}} = [\, f_{\text{vis}}(o_t),\ d_t,\ a_{t-1},\ f_{\text{prop}}(x_t),\ e(\ell)\,], \tag{B.20}$$

which exposes both the absolute scene content and its recent change, together with the previous control signal, to the residual policy.

On top of this state representation, we reuse exactly the same residual RL architecture as in our main baseline. The actor receives s_t^{ResAct} and outputs a residual action $a_t^{\text{res}} \sim \pi_\phi(a \mid s_t^{\text{ResAct}})$, while the critic estimates $Q_\theta(s_t^{\text{ResAct}}, a_t^{\text{res}})$. The final control command is still defined as an additive refinement to the frozen backbone action,

$$a_t = a_t^{\text{base}} + a_t^{\text{res}}, \tag{B.21}$$

so that we modify only the state input to the residual branch rather than changing the action parameterization. Training uses the same Soft Actor–Critic recipe and hyperparameters as our residual baseline (discount factor, learning rates, batch size, target update schedule, replay buffer size, etc.), and relies solely on the environment reward r_t^{env} without any retrieval or similarity-based shaping. During both training and evaluation, we maintain the cached pair $(f_{\text{vis}}(o_{t-1}), a_{t-1})$ online (with $f_{\text{vis}}(o_{-1})$ and a_{-1} initialized from the first step), ensuring that ResAct operates under the same backbone, data, and interaction budget as our other baselines while explicitly leveraging observation differences and previous actions in its residual policy.

B.4.3. R2A-style retrieval-augmented baseline

We also implement a retrieval-augmented RL baseline inspired by R2A [4] on top of the same frozen VLA backbone as EFN. Because the original R2A is designed for value-based, non-residual control with a specialized dual-process architecture and multiple auxiliary objectives, we do not attempt a bit-level reproduction. Instead, we construct an *R2A-style retrieval-augmented SAC* that preserves the core idea of conditioning the policy and critic on a retrieval-derived context vector, while adapting the design to our continuous-control, residual-on-VLA setting. Throughout this section we explicitly refer to this baseline as an *adapted implementation* and keep all SAC hyper-parameters and backbone interfaces identical to those of EFN for a fair comparison.

State representations and experience storage follow our EFN setup. At each step t, we form a state embedding

$$s_t = \big[f_{\text{vis}}(o_t),\ f_{\text{prop}}(x_t),\ e(\ell)\big], \tag{B.22}$$

where f_{vis} is the pooled visual embedding from the frozen VLA, f_{prop} encodes proprioceptive signals (joint angles, end-effector pose, etc.), and $e(\ell)$ is the language embedding of the task instruction. We reuse the same step-level experience bank as EFN, but for R2A-style retrieval we additionally precompute keys $k_i = f_{\text{key}}(s_i)$ for stored states s_i

using a small MLP. At decision time we compute a query $q_t = f_{\text{query}}(s_t)$ (sharing parameters with f_{key} in our implementation), perform k-nearest-neighbor search over $\{k_i\}$ using cosine similarity, and obtain indices $\{i_1, \ldots, i_K\}$. For each neighbor we build a retrieval feature $\tilde{s}_{i_j}$ by concatenating its state, executed action, and scalar reward, and aggregate them with a single-head attention module:

$$\alpha_j = \text{softmax}_j\left(\frac{q_t^\top W_a \tilde{s}_{i_j}}{\sqrt{d}}\right), \qquad u_t = \sum_{j=1}^{K} \alpha_j \, W_v \tilde{s}_{i_j}, \tag{B.23}$$

which yields the retrieval context u_t that summarizes task-relevant past experience.

On top of the frozen VLA backbone, we train a residual SAC agent that receives both s_t and u_t. The backbone produces a base action $a_t^{\text{base}} = \pi_{\text{VLA}}(o_t, \ell)$, while the residual actor outputs $a_t^{\text{res}} \sim \pi_\phi(a \mid s_t, u_t)$, and the executed action is $a_t = a_t^{\text{base}} + a_t^{\text{res}}$. The critic is similarly conditioned on the retrieval context, $Q_\theta(s_t, u_t, a_t)$, and is trained with the standard SAC objective using only the environment reward r_t^{env}; we do not introduce any similarity-based reward terms or additional auxiliary losses. At training time, every sampled transition from the replay buffer recomputes u_t and u_{t+1} via the same retrieval procedure, and at evaluation time we freeze all parameters and keep retrieval active, so that adaptation arises purely from the parametric policy and value functions conditioned on u_t. This design captures the central principle of R2A—using retrieved experience as an additional context for decision making—while remaining compatible with our residual continuous-control setting and avoiding the heavy slot-based memory, bidirectional RNNs, and information-bottleneck losses of the original algorithm.

B.4.4. GC-TTT-style test-time training baseline

We additionally implement a goal-conditioned test-time training (GC-TTT) baseline adapted to our VLA+residual setting. Concretely, we treat our residual SAC policy (a residual head on top of a frozen VLA backbone) as a goal-conditioned policy and pair it with an offline replay buffer $\mathcal{D}$ collected during pre-training. Each transition in $\mathcal{D}$ stores $(s_t, a_t, r_t, s_{t+1}, g)$ together with the state embedding $z_t = f_{\text{enc}}(s_t)$ and goal embedding $e(g)$ used by our main method. For every episode in $\mathcal{D}$ we also precompute a discounted trajectory return $R_{\text{traj}} = \sum_t \gamma^t r_t$ and attach this scalar to all transitions from that episode, which serves as a simple measure of trajectory quality.

At evaluation time, given a new goal g^* we perform a single GC-TTT update before running the episode. From the initial observation o_0 we obtain the state embedding $z_0 = f_{\text{enc}}(s_0)$ and compute a relevance score for each stored transition i as

$$\text{sim}_i = \cos(z_0, z_i) + \lambda \cos\big(e(g^*), e(g_i)\big), \tag{B.24}$$

where λ balances state and goal similarity. We first select the top-M transitions by sim_i to form a relevant subset $\mathcal{D}_{\text{rel}}$, and then rank them by the associated trajectory return R_{traj}. Keeping only the top q-th percentile yields a high-quality subset $\mathcal{D}_{\text{good}}(s_0, g^*)$ that is both close to the current state–goal pair and drawn from successful trajectories.

We then adapt the residual policy parameters ϕ on $\mathcal{D}_{\text{good}}(s_0, g^*)$ using a small number N of gradient steps of a behavior-cloning loss,

$$\mathcal{L}_{\text{GC-TTT}}(\phi) = -\mathbb{E}_{(s_i, a_i) \sim \mathcal{D}_{\text{good}}}\big[\log \pi_\phi(a_i \mid s_i, g^*)\big], \tag{B.25}$$

while keeping the VLA backbone frozen. After these updates, the adapted policy $\pi_{\phi_{\text{adapt}}}$ is used to control the robot for the entire episode. At the end of the episode, we reset ϕ back to the pre-trained weights ϕ_0 and repeat the same procedure for the next evaluation goal. Compared to the original GC-TTT formulation, our implementation is a critic-free variant that relies on trajectory returns instead of a learned value function and only adapts the lightweight residual head. We therefore view this baseline as an "adapted GC-TTT" instantiation tailored to our VLA setting rather than an exact reproduction of the original system.

B.5. Real-World Setup and Task Specifications

Hardware and environment. All real-world experiments are conducted on an AgiBot-G1 manipulator mounted next to a fixed tabletop workspace. The arm is controlled in Cartesian space at 10 Hz using position and gripper commands issued by the frozen VLA backbone (and EFN, when enabled). A calibrated RGB(-D) camera is rigidly mounted above the workspace and looks down at the table and shelf; its images are resized to the same resolution as in simulation and fed directly into the VLA visual encoder without additional preprocessing. The shelf, box, and drawer are anchored at fixed positions in front of the robot, and all objects are restricted to a safe reachable workspace.

Task specifications. We instantiate four real-world manipulation tasks that mirror the long-horizon, language-conditioned scenarios used in simulation: **BottlePlace**, **ShelfSort**, **StockLift**, and **DrawerStore**. Each task is defined by (i) an initial object configuration, (ii) a target region, and (iii) a success condition.

BottlePlace. A plastic bottle is placed on the tabletop within a fixed start region in front of the right arm. A small open box is placed to the side of the table and is treated as the goal region. The task is considered successful if the bottle ends in the interior of the box (its 3D center lies inside the pre-defined box region) and the right gripper is fully opened at the end of the episode. A typical language command is: *"Use your right arm to pick up the bottle and place it into the box."*

ShelfSort. Several drink containers (e.g., cans and bottles with different labels) are arranged on a tabletop shelf, with at least one item of the same category already present in a designated "cluster" region. Another drink from the same category is placed on the shelf in a randomized start position. The success condition requires that the target drink is placed within the cluster region next to visually similar items (within a fixed 3D tolerance) and that the right gripper is released. An example language instruction is: *"Grasp the drink from the shelf with your right arm and place it next to the similar drinks, then open the gripper."*

StockLift. A stack of identical cans is placed on a small auxiliary tabletop, and a separate shelf contains an empty spot marked as the goal region. The top can of the stack is always designated as the object to manipulate. The task is successful if the robot lifts the top can from the stack, moves it to the shelf, and leaves it resting within the predefined goal region with the right gripper open. A representative command is: *"Pick up the top can from the small table with your right arm and place it onto the shelf, then release."*

DrawerStore. A packet of napkins is placed on the main tabletop within reach of the right arm. A drawer in front of the robot starts in the closed position and serves as the storage location. The robot must (1) use the left arm to pull the drawer open beyond a minimum displacement; (2) use the right arm to grasp the napkin packet from the tabletop; (3) transfer the packet from the right gripper to the left gripper; and (4) place the packet into the drawer interior. The episode is marked as successful if, at termination, the napkins lie fully inside the drawer volume and both grippers are open. An example language instruction is: *"Open the drawer with your left arm, pick up the napkin packet with your right arm, hand it to the left arm, and place it into the drawer."*

Per-episode protocol and experience logging. For all four tasks, each episode begins from a manually reset initial configuration with objects placed in their designated start regions and the arms in a neutral pose. At each control step, the overhead camera image and the language instruction are passed through the frozen VLA backbone to produce an action proposal; EFN (or other baselines) then optionally modifies this proposal before it is sent to the low-level controller at 10 Hz. We cap the horizon at a fixed maximum number of steps (typically 60–80, depending on the task); if the success condition is met earlier, the episode terminates immediately. On failure (timeout, dropped object, or clear deviation from the goal), the episode is terminated and the environment is manually reset.

During deployment with EFN, we maintain an online experience bank that stores real-world rollouts. For each successful episode, we record the RGB images, language instruction, latent visual embeddings, latent action tokens, and executed actions at every non-blank step, and append these transitions to the experience bank. Failed or aborted episodes are not written to the bank by default to avoid contaminating the retrieval database with suboptimal behaviors.

Safety considerations. All experiments are run with conservative joint and workspace limits, velocity and torque bounds, and soft joint limits enabled on the AgiBot-G1 controller. Episodes are immediately stopped and reset if any potential collision or unsafe configuration is detected. These safety measures slightly reduce the usable workspace but do not otherwise affect the task definitions or evaluation protocol described above.

C. Rationale behind Design

C.1. Design Objectives and Constraints

Our starting point is a practical limitation of current vision–language–action (VLA) policies: once trained and deployed, the backbone is typically frozen and cannot *learn from new experience* collected in the target environment. However, real-world deployment quickly exposes distribution shift, long-horizon credit assignment issues, and subtle failure modes that are hard to anticipate in advance.

Our primary objective is therefore to endow a *frozen* VLA backbone with a mechanism that can still "become smarter" over time at deployment, improving success rate and reducing action steps by exploiting past executions, without ever modifying backbone weights. This objective is shaped by several hard constraints arising from realistic deployment scenarios:

- **No backbone finetuning.** The core VLA policy is treated as a fixed, pre-trained component. We do not update its parameters during deployment; all adaptation must occur through an auxiliary module that interfaces with its latent representations.
- **Limited compute at deployment.** Real robots and embedded platforms cannot afford frequent, large-scale finetuning or replay-based training. Any learning or adaptation mechanism must be lightweight, incremental, and compatible with strict latency and memory budgets.
- **Safety and reproducibility.** Deployment-time changes must be controlled and auditable. Residual corrections should stay in a safe regime (e.g., not arbitrarily overriding the backbone), and the overall procedure should remain stable enough to reproduce results across runs and hardware setups.
- **Backbone-agnostic interface.** The adaptation mechanism should generalize across different VLA architectures (e.g., OpenVLA, UniVLA, AgiBot-G1 controllers) without requiring backbone-specific surgery. This motivates operating purely on the shared latent interface (visual tokens, language tokens, and latent action tokens) exposed

by the frozen policy.

- **Minimal code and system changes.** To ease integration into existing stacks, we aim to attach a thin "add-on" module on top of the VLA's latent action interface, rather than redesigning perception, language processing, or low-level control.

All subsequent design choices in our framework, including experience storage, retrieval, residual policy architecture, and training objectives, are made to satisfy these constraints while serving the central goal of *deployment-time improvement with a frozen backbone*. The proposed Experience Feedback Network (EFN) should therefore be viewed as a minimal, safe, and portable layer that augments a unified VLA policy with post-deployment learning capabilities, rather than a replacement for the underlying model.

C.2. Why a Frozen VLA with a Residual Head?

Given the objectives and constraints in Sec. C.1, we deliberately start from a *frozen* vision–language–action (VLA) backbone and attach a light residual head, instead of (i) directly finetuning the large model at deployment, or (ii) retraining a new policy from pixels to actions.

Continuing to finetune the backbone online is conceptually simple but practically fragile. It requires nontrivial compute and memory on the robot, frequent gradient updates, and careful replay management to avoid catastrophic forgetting. More importantly, aggressive finetuning risks damaging the strong cross-task generalization that large VLAs have already acquired on benchmarks such as LIBERO and our real-world AgiBot-G1 tasks. Once the backbone drifts, it is difficult to guarantee reproducibility across deployments and hardware.

On the other hand, discarding the VLA and retraining a full pixel-to-action policy would forfeit the benefits of large-scale pre-training altogether. Such a design would be data-inefficient, slow to adapt in deployment, and tightly coupled to a specific robot and environment, contradicting our goal of a backbone-agnostic, reusable mechanism.

A residual head offers a middle ground. We keep the base policy's action a_t^{base} as the default behavior and learn only a small correction Δa_t:

$$a_t = a_t^{\text{base}} + \Delta a_t. \tag{C.1}$$

This preserves the backbone's strong generalization and task coverage, while allowing EFN to refine local decisions where the base policy systematically underperforms. Because the residual only needs to model *deviations* from a competent policy, it is more sample-efficient and typically more stable to train, consistent with the intuition behind residual policy learning and ResAct-style methods [9].

Our ablations further support this design. When we replace the residual head with a standalone action predictor that ignores the base action ("no residual / direct action prediction"), performance drops in both success rate and step efficiency (see Appendix E.2). In other words, throwing away the base action and asking the small head to solve the entire control problem again is strictly worse than learning corrections on top of it.

Compared to prior residual RL baselines such as ResAct, EFN's residual is *retrieval-driven* rather than blind: the correction Δa_t is conditioned on the current latent state and on retrieved experience that encodes how similar situations were handled in the past. This retrieval-guided residual design is central to EFN: it lets us keep the backbone frozen, leverage its existing generalization, and still obtain deployment-time improvements through small, data-efficient adjustments.

C.3. Why an Episodic Experience Bank?

To support deployment-time improvement with a frozen backbone, we organize past executions into an *episodic* experience bank. Instead of storing isolated transitions, we keep complete rollouts $\tau = \{(o_t, a_t, \dots)\}_{t=1}^{T}$ together with their language instruction and outcome. At the *trajectory* level, we encode the instruction into a fixed-dimensional embedding ℓ_τ, which is used for instruction-filtered retrieval during deployment. At the *step* level, we store compact visual features, retrieval keys, and the corresponding base actions. This two-level structure lets us first select a small set of candidate episodes that match the current instruction, and then perform fine-grained, step-wise retrieval of locally similar states within those episodes.

The retrieval key for each step combines mean and max pooled features. Concretely, given a set of visual tokens $\{v_i\}_{i=1}^{N}$, we compute

$$\mu = \frac{1}{N}\sum_i v_i, \tag{C.2}$$

$$m = \max_i v_i \quad \text{(element-wise)}, \tag{C.3}$$

$$k = \text{LN}([\mu; m]), \tag{C.4}$$

and L2-normalize k before computing cosine similarity. Intuitively, the mean component captures the global scene configuration, while the max component emphasizes salient objects and local cues that may be critical for manipulation. L2-normalization followed by cosine similarity yields a scale-insensitive, numerically stable similarity measure across different backbones and environments. In practice, we found this mean–max fusion more robust than simple global average pooling alone, which tended to produce noisier neighbors and more sensitivity to background clutter.

We further impose two simple but important priors in retrieval: *instruction filtering* and an *efficiency prior*. Instruction filtering uses ℓ_τ to restrict retrieval to episodes

whose language embedding is close to the current instruction, which reduces harmful cross-task transfer on multi-task benchmarks such as LIBERO and prevents the agent from imitating "good actions for the wrong task." The efficiency prior biases the bank towards shorter successful trajectories: during offline construction and online retrieval, we prefer episodes that reach success in fewer steps. This aligns with deployment-time goals (reusing concise, high-quality behaviors) and empirically improves both success and step efficiency; removing instruction filtering or the length prior ("w/o instruction embed") degrades performance in our ablations (see Appendix Table D.1).

Finally, we emphasize that the experience bank and its retrieval interface are shared across all retrieval-based baselines. kNN-RAG, R2A-style, GC-TTT-style methods, and EFN all operate on the *same* episodic bank, use the *same* key/query construction and cosine similarity, and differ only in how they *consume* the retrieved neighbors (direct action copying, test-time gradient updates, or retrieval-conditioned residuals). This shared interface ensures that EFN does not benefit from any hidden advantage at the retrieval level; improvements arise solely from how retrieved experience is integrated into action selection.

C.4. Why Residual Policy Optimization Instead of Direct Action Prediction?

From an optimization perspective, directly predicting actions with a new lightweight head is equivalent to discarding the base policy altogether. The new actor must re-learn the full mapping from latent state to action, effectively reconstructing the entire action manifold of the task. For a small network trained from sparse, delayed rewards at deployment, this is a hard problem: the search space is large, gradients are noisy, and the sample complexity is high. In practice, such a head tends to overfit to a small set of trajectories and fails to reliably generalize across the diverse scenes and tasks that the frozen VLA backbone already handles well.

Residual policy optimization takes a different view. Instead of learning a fresh action a_t^{new}, EFN learns a correction Δa_t around the base action a_t^{base}:

$$a_t \;=\; a_t^{\text{base}} + \Delta a_t. \tag{C.5}$$

This constrains the RL search space to a local neighborhood of a_t^{base}, which is already a strong, pre-trained solution. The residual head only needs to model how to adjust actions in cases where the backbone is systematically biased or underspecified, rather than solving the whole control problem from scratch. This local parameterization smooths the optimization landscape, reduces the effective dimensionality of the problem, and makes gradient-based updates more stable and data-efficient. Moreover, when retrieval is unreliable or EFN is uncertain, the learned residual naturally shrinks towards zero, and the agent safely falls back to the base action a_t^{base}.

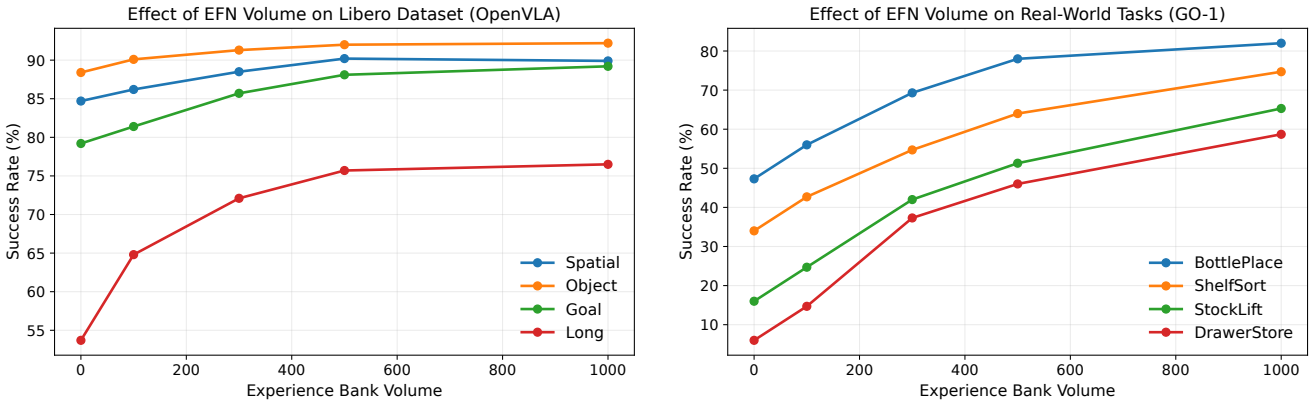
Our ablations corroborate this reasoning: a variant that replaces the residual optimization with direct action prediction (ignoring a_t^{base}) consistently underperforms EFN in both success and step metrics (see Sec. E and the "direct action prediction" rows in the supplementary tables). These results suggest that, under a frozen VLA backbone, optimizing a retrieval-conditioned *residual* is a strictly more favorable formulation than learning a standalone action head.

C.5. Why Similarity-shaped Dense Rewards and the Anti-idling Design?

The reward decomposition in main-paper Fig. 4 is designed to turn retrieval into a dense and behaviorally aligned learning signal, rather than relying solely on sparse task success. A naive starting point is to reward the agent by how well its next observation matches the retrieved next frame, e.g., a cosine similarity term $a_t \in [0, 1]$ between the actual next visual embedding and the retrieved target. This already provides a dense signal at every step, but it also introduces a failure mode: the agent can exploit the reward by finding a visually similar pose and then "vibrating in place," maintaining high similarity without making real progress toward task completion.

To avoid this, we explicitly split the similarity-shaped reward into components that distinguish *progress* from *stagnation*. The progress term $[p_t]_+$ measures how much closer the agent moves towards the retrieved next frame compared to the previous step, and only positive progress is rewarded. This encourages trajectories that steadily approach the retrieved outcome, rather than merely staying close once a high-similarity configuration is found. In addition, we introduce a motion term m_t and an anti-idling penalty weighted by w_{lazy}: steps that change the scene or end-effector state in a meaningful way are rewarded, whereas near-static steps (high similarity but negligible motion) are penalized. Finally, a temporal coefficient λ_{time} discourages unnecessarily long trajectories, biasing the policy towards concise, decisive behaviors that reach success in fewer steps.

From this perspective, the reward is not a black-box signal delegated entirely to SAC to "figure out" credit assignment; instead, it directly encodes our behavioral preferences: (i) match good retrieved futures, (ii) *move toward* them rather than hovering idly nearby, and (iii) reach them quickly. The ablations in Table D.1 (also summarized in main-paper Fig. 4) support this design. Removing the similarity-shaped dense terms ("w/o dense rewards") and relying only on sparse task success leads to lower average success and less consistent improvements over OpenVLA, indicating that the dense retrieval-based components are crucial for effective post-deployment learning. Dropping the anti-idle terms ("w/o anti-idle") yields a policy that attains simi-



(a) LIBERO tasks

(b) Real-world tasks

Figure C.1. Performance consistently improves as the experience bank grows in both simulated LIBERO tasks with OpenVLA and real-world manipulation tasks with GO-1. Here, the bank volume refers to the number of stored trajectories (episodes) in the experience bank.

lar success but with noticeably longer episodes, confirming that, without explicit penalties on idleness, the agent tends to favor visually safe but inefficient behaviors. In contrast, the full EFN reward decomposes the similarity signal into progress, motion, anti-idle, and time components, producing the best overall trade-off between success and step efficiency on LIBERO.

C.6. Why Reinforcement Learning for EFN?

A natural question is why EFN is trained with reinforcement learning rather than supervised learning. The core reason is that our training signal is defined *through* the environment dynamics and therefore does not admit straightforward backpropagation. As described in Sec. 3.3, after executing $\mathbf{a}_t$ the environment advances to a new state s_{t+1} with vision features $\mathbf{F}_{t+1}$. We then define a dense semantic similarity reward

$$r_t^{\text{sem}} = \cos\big(\mathbf{u}(\mathbf{F}_{t+1}), \mathbf{u}(\hat{\mathbf{F}}^{+})\big), \tag{C.6}$$

where $\hat{\mathbf{F}}^{+}$ is the retrieved next-frame feature and $\mathbf{u}(\cdot)$ is a projection into the similarity space. Both s_{t+1} and $\mathbf{F}_{t+1}$ are only revealed after interacting with the simulator or real robot. Since the transition $s_t \xrightarrow{\mathbf{a}_t} s_{t+1}$ is non-differentiable and we do not have a tractable model of the environment, we cannot propagate gradients from r_t^{sem} back through the dynamics to EFN parameters as in standard supervised learning. Instead, we are precisely in the canonical RL setting: EFN chooses actions, observes next states and rewards from a black-box environment, and must improve its policy from these interaction signals.

Within RL, we adopt Soft Actor–Critic (SAC) for training EFN. First, our control space is continuous (especially on the real AgiBot-G1 platform), making SAC a natural choice as a widely used off-policy algorithm for continuous actions. Second, the off-policy nature of SAC lets us aggressively reuse transitions from the experience bank and the ongoing rollouts, which aligns with our central theme of "learning from experience" rather than discarding past data. Third, SAC's entropy regularization is particularly helpful in our residual setting: it encourages exploration instead of collapsing too early to the trivial solution $\Delta\mathbf{a}_t \approx \mathbf{0}$, and stabilizes learning when the dense similarity-shaped rewards and anti-idle penalties are combined.

Our ablations support these design decisions. When we remove SAC and rely only on a value-style critic without the entropy-regularized actor update ("w/o SAC'' in Table D.1), EFN's improvements over the backbone shrink in both success and step metrics. In contrast, the full SAC-based optimization leverages the dense similarity rewards and the retrieval structure to deliver consistent gains on top of a frozen VLA policy. Overall, reinforcement learning—and SAC in particular—fits our design philosophy: a lightweight optimization layer that operates on the VLA's latent interface, reuses interaction experience, and learns directly from deployment-time feedback without modifying any backbone weights.

C.7. Why Live Experience Growth at Deployment?

Our deployment-time design explicitly separates *what can change* (the experience bank) from *what remains fixed* (the VLA backbone and EFN weights), so that the system can adapt in the field without sacrificing stability.

Why not update network parameters at deployment? Updating parameters online turns deployment into test-time training (TTT), which easily leads to instability and hard-to-debug performance collapse; our GC-TTT baseline empirically illustrates this risk. Moreover, backpropagating through a large VLA in the wild is compute- and engineering-heavy, and makes it difficult to guarantee la-

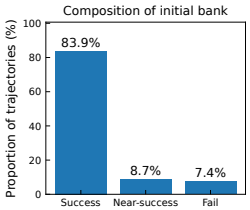


Figure C.2. Composition of successful, near-successful, and failed trajectories on LIBERO.

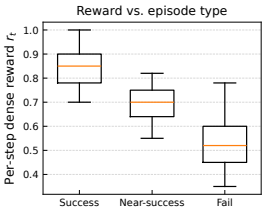


Figure C.3. Per-step reward distribution for successful, near-successful, and failed episodes.

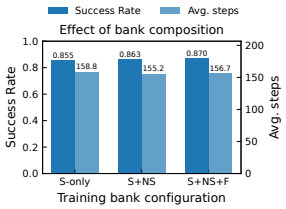


Figure C.4. Training performance with different experience bank compositions on LIBERO.

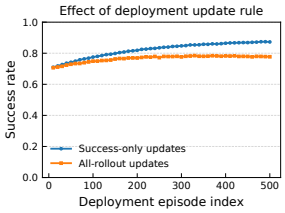


Figure C.5. Comparison of rollout update strategies during deployment: Success-only vs All.

tency and reliability. Keeping the backbone and EFN frozen also simplifies reproducibility: a fixed set of weights yields consistent behavior, whereas a continuously updated model is a moving target whose behavior depends on its entire online training history.

Why still grow the experience bank? Freezing weights does not preclude "learning from experience": EFN adapts by changing *which* trajectories it can retrieve, not *how* it processes them. As shown in Figure C.1, enlarging the experience bank consistently improves success rates on LIBERO and real-world GO-1 tasks, because a larger bank provides more relevant and diverse execution snippets for residual guidance on top of the frozen VLA. By appending only validated (successful) rollouts, this live growth remains a safe form of adaptation: the agent gains from accumulating high-quality experiences while preserving the stability, compute efficiency, and auditability of a frozen model.

D. Discussion, Limitations, and Future

D.1. Success and Failure Experience

A key design choice in EFN is to treat success, near-success, and failure differently at *training* and *deployment* time. During training, the offline bank intentionally contains a mixture of successful rollouts, near-successful rollouts that complete only a subset of sub-goals, and even clearly failed trajectories (Figure C.2). This is feasible because our reward r_t is defined purely from semantic feature similarity rather than a binary success flag: any transition that moves the state closer to a goal-consistent embedding can provide useful signal, regardless of whether the episode ultimately succeeds.

Figures C.3 and C.4 support this design. The per-step reward distributions show that failed and near-successful episodes still contain many high-similarity steps, i.e., meaningful partial progress. In the ablation on bank composition, moving from `S-only` to `S+NS` consistently improves performance, and even `S+NS+F` does not significantly degrade results, indicating that EFN is robust to a moderate amount of failure noise when the reward is similarity-based.

In contrast, during deployment we only append rollouts classified as successful to the live bank. Success detection is fully automatic: instead of human labels, we reuse the same feature-matching criterion as Eq. (11) in the main paper and mark an episode as successful if its final state embedding is sufficiently similar to the goal embedding. Figure C.5 shows that appending all rollouts (`All-rollouts`) gradually harms performance, whereas a success-only update strategy maintains a cleaner bank and yields higher, more stable success rates. A remaining limitation is that our current detector is a simple similarity threshold; future work could explore learned, task-aware success estimators or soft weighting of borderline near-successes to further refine which experiences are admitted into the live bank.

D.2. Experience-centric post-deployment learning

Most prior approaches to post-deployment adaptation in embodied agents are implicitly *parameter-centric*: they treat the agent's weights as the primary locus of change, and focus on finetuning, test-time training, or meta-RL updates to continually reshape the backbone policy. In contrast, EFN adopts an explicitly *experience-centric* view. Once deployed, the weights of both the VLA backbone and the EFN residual controller are frozen; the only component that continues to evolve is the experience bank. Under this perspective, learning after deployment is expressed not as further gradient descent in parameter space, but as the incremental accumulation, organization, and retrieval of interaction histories that remain external to the backbone.

Our empirical comparisons support this reframing. Methods such as R2A-style retrieval-augmented RL and GC-TTT represent the "keep updating the backbone" route: they can improve performance, but at the cost of substantial online optimization, sensitivity to hyperparameters, and potential instability when applied to long-horizon manipulation on LIBERO or to real-world AgiBot-G1 deployment. By contrast, EFN uses a lightweight residual head together with retrieval over a growing bank and already achieves comparable or better gains, while leaving the underlying VLA unchanged. This suggests that much of the benefit commonly attributed to continual finetuning can instead be

Table D.1. Ablation studies on the LIBERO benchmark

Method	Spatial		Object		Goal		Long		Average	
	Succ.	Step	Succ.	Step	Succ.	Step	Succ.	Step	Succ.	Step
OpenVLA	84.7	119.5	88.4	163.7	79.2	121.5	53.7	275.9	76.5	160.2
w/o SAC	80.4	124.2	81.5	170.9	76.7	122.3	42.6	282.3	70.3	161.2
w/o dense rewards	85.6	117.0	90.1	161.4	84.3	119.4	66.8	270.9	81.7	161.3
w/o instruction embed	88.1	114.2	90.7	162.1	86.9	116.8	74.4	**262.1**	85.0	160.0
w/o anti-idle	89.9	120.9	91.3	162.9	**88.2**	123.1	75.4	280.2	86.2	167.4
OpenVLA+EFN(full)	**90.2**	**111.8**	**92.0**	**158.3**	88.1	**114.5**	**75.7**	264.3	**86.5**	**158.2**

harvested by enriching and exploiting a structured memory of past executions, without repeatedly pushing the base policy into unexplored regions of parameter space.

Viewing post-deployment adaptation as experience growth rather than perpetual finetuning has broader implications. It makes behavior easier to reproduce and audit, since changes over time are driven by the content of an explicit memory store rather than opaque weight drift, and it aligns better with realistic deployment constraints where onboard compute and engineering budgets limit extensive online training. At the same time, it shifts the core challenges toward curating, scaling, and querying experience banks, for example by addressing long-term storage, retrieval efficiency, and coverage of rare but safety-critical events. A more detailed rationale for each architectural choice underlying this experience-centric design is given in Appendix C; here we emphasize the conceptual shift from parameter-centric to experience-centric post-deployment learning.

D.3. When does EFN help, and why?

Empirically, EFN helps most once the experience bank covers a modest set of *typical* situations rather than an extremely large corpus. Increasing the bank volume from 0 to 300 episodes already yields a major jump in performance, while further expanding to 1000 provides only diminishing but consistent gains (Figure C.1). This suggests that EFN is mainly sensitive to whether common layouts, viewpoints, and language instructions are represented, rather than to exhaustively storing every trajectory. In benchmarks like LIBERO, where tasks share similar manipulation structure, even a medium-sized bank is enough for retrieval to regularly surface useful precedents.

Ablations also clarify what kind of signal is needed. A retrieval-only kNN-RAG controller that directly replays the stored action of the nearest state often hurts performance, indicating that "copying the past" is too brittle under small pose or occlusion shifts. A residual-only variant (ResAct) that ignores episodic retrieval but learns a global residual head yields modest but stable gains, consistent with correcting systematic biases of the base VLA. EFN outperforms both by combining episodic guidance with local adaptation: retrieved trajectories provide a high-level prior, while the residual adjusts to the current observation. On the real AgiBot-G1 tasks, the gap between 300 and 1000 episodes is even smaller, hinting that in real deployment the task distribution is more repetitive and sensor noise higher, so memory *quality* matters more than sheer size. We also observe clear failure modes: if the scene is structurally reconfigured (e.g., the tabletop is completely rearranged), old experiences can become misleading, and when the base VLA is very weak, EFN can only provide small local corrections rather than "reviving" the policy from scratch.

D.4. Limitations

The first limitation of EFN is its reliance on the backbone representation quality. EFN operates entirely in the latent space exposed by the frozen VLA; if the backbone fails catastrophically for certain objects, viewpoints, or language instructions, both retrieval and residual correction will fail jointly. Moreover, the current design needs a non-trivial number of successful trajectories before it can boost performance: in extremely cold-start regimes with virtually no successes, EFN has little useful signal to retrieve. In practice, we observe that once there are even a handful of successful rollouts, EFN can already yield substantial gains. For example, on the challenging *DrawerStore* task in the real-world AgiBot-G1 experiments, whose base success rate is only **5.3%**, EFN increases performance to **37.3%** with a bank volume of 300 and further to **58.7%** with volume 1000 (Table 2 in the main paper; visualizations in Section F). Nevertheless, if the backbone truly never succeeds (e.g., close to 0% success), our method provides limited leverage, and the more appropriate remedy is to retrain or replace the underlying VLA rather than rely on post-deployment experience alone.

The second limitation concerns the size and management of the experience bank. Under the scales considered in this work (LIBERO simulation and our AgiBot-G1 experiments), the memory footprint and retrieval latency remain modest, and EFN adds only a small overhead on top of the base VLA policy, as summarized in Table 3 in the main paper. However, in long-running deployments with many robots or continuously operating fleets, the bank will grow without bound unless additional mechanisms are intro-

duced. In such settings, it will be necessary to design more sophisticated caching, compression, and replacement strategies (e.g., task-aware coreset selection, time- or novelty-based forgetting, or hierarchical multi-bank structures) to keep both storage and retrieval cost under control while preserving the benefits of rich episodic experience.

D.5. Future directions

A natural next step is to strengthen the *experience interface* itself. In our current design, EFN compares the latent encodings of the present observation against stored experiences, and conditions residual actions on the retrieved matches. An intriguing extension is to couple EFN with world models or predictive representations, so that retrieval can be performed in the space of *future* trajectories rather than only current views. Concretely, one could predict a short-horizon rollout under the base policy, and retrieve experiences whose *successor segments* best align with this predicted future, allowing EFN to correct course before errors manifest. Such predictive retrieval opens the door to uncertainty-aware guidance and more robust long-horizon adaptation.

Another promising direction is to move beyond a per-backbone, per-domain experience bank toward *shared* experience repositories across tasks and robots. In the present work, each VLA backbone maintains its own experience bank, tailored to a single domain or embodiment; this simplifies training but prevents reusing experience across, for example, OpenVLA and UniVLA or between different manipulation platforms. Future work could investigate unified latent representation spaces or lightweight alignment layers that enable cross-robot retrieval and reuse of experience, while guarding against negative transfer. This would turn EFN-style mechanisms into a more general infrastructure for pooling and distilling experience across heterogeneous agents and environments, rather than a separate add-on for each backbone.

E. Supplementary Experimental Results

E.1. Runtime and Memory Footprint of EFN

E.1.1. Standalone Cost

We first measure the computational footprint of EFN in isolation, excluding the VLA backbone and the critic. We instantiate the EFN policy using the same configuration as in our main experiments (1024-dimensional embedding, 2-layer Transformer encoder), and feed synthetic latent and visual features that match deployment-time shapes: $(1, 4, 4096)$ latent tokens and $(1, 256, 4096)$ visual embeddings.

We benchmark 200 forward passes after 50 warm-up iterations using CUDA event timing. As shown in Table E.1, EFN adds only ~ 1.94 ms per control step and incurs minimal activation memory (~ 2.6 MB). The parameter footprint is modest (208 MB), especially compared to the frozen VLA backbone. These results indicate that EFN is lightweight enough for real-time control and can be cleanly integrated into existing policies.

Table E.1. Standalone runtime and memory footprint.

Method	Bare EFN policy
Runtime (ms / step)	1.944
Param memory (MB)	208.17
Peak activations (MB)	2.59

E.1.2. Integration Cost

We next examine the end-to-end overhead when EFN is integrated into the full GO-1 policy, including vision encoding, LLM inference, and action decoding. This measurement in Table E.2 reflects the actual burden added during real-world deployment.

Table E.2. Inference-time efficiency comparison for real-world deployment. Change is computed relative to the base GO-1.

Metric	GO-1	GO-1 + EFN (ours)
Per-step latency (ms) ↓	**35.7**	37.2 (+4.2%)
Peak GPU memory (MB) ↓	**12,851**	13,464 (+4.77%)
Avg. steps / episode ↓	491.8	**435.1** (-11.5%)
Episode time (s) ↓	17.6	**16.2** (-7.9%)

Integrating EFN increases per-step latency by only 4.2% and peak GPU memory by 4.77%, while substantially reducing the number of steps required to complete an episode (-11.5%). This yields a net improvement in real-world episode time (-7.9%). Overall, EFN adds minimal system overhead yet delivers measurable efficiency gains in deployment.

E.2. Residual correction vs. direct action prediction

For completeness, we also consider a "direct-action" variant of EFN where the base VLA policy is disabled and EFN is trained to directly output absolute actions. Concretely, instead of

$$\mathbf{a}_t = \mathbf{a}_t^{\text{base}} + \Delta\mathbf{a}_t^{\text{EFN}}, \tag{E.1}$$

we let the EFN head produce the full action

$$\mathbf{a}_t = \mathbf{a}_t^{\text{EFN}}, \tag{E.2}$$

using the same frozen VLA features and the same lightweight MLP head as in our residual formulation (see Tab. B.1 for the architectural footprint). All other training hyperparameters (SAC configuration, reward shaping, and interaction budget) are kept identical to the residual version.

Empirically, this direct-action EFN fails to learn meaningful behavior in our environments: across both simulation

Table E.3. Residual vs. direct-action EFN on the LIBERO benchmark. The direct-action variant fails to learn and remains close to zero.

Method	Avg. Succ
OpenVLA	76.5
OpenVLA+EFN (residual)	87.0
OpenVLA+EFN (direct-action)	**0**

and the real-world platform, the success rate remains close to zero throughout training (as shown in Table E.3), and the learning curves exhibit strong oscillations rather than stable improvement. We therefore omit this variant from the main tables and report it here only as a negative result. We hypothesize two main reasons. First, by removing the base policy we effectively discard the generalization ability acquired by the pre-trained VLA, forcing a small EFN head to re-learn the entire control policy from sparse rewards and high-dimensional observations. Second, the head is deliberately designed to be extremely lightweight to meet our latency budget; while this is sufficient for modeling residual corrections around a strong backbone, it appears underparameterized for modeling the full continuous action manifold from scratch. Taken together, these findings justify our choice of treating EFN as a residual corrector rather than a standalone policy.

F. Visualization

To complement the quantitative results in Table 2 in the main paper, we provide real-world demonstration videos recorded on the AgiBot-G1 manipulation platform in the `demo/` directory. For each real-world task evaluated in Table 2 in the main paper, we include a video of the EFN-augmented policy executing the task (e.g., `StockLift_with_EFN.mp4`), illustrating how the deployed agent uses retrieved experiences to refine the base VLA policy and produce more reliable executions.

Due to space limitations in the supplementary material, we provide one paired comparison (with vs. without EFN) for the **most challenging** task, *DrawerStore*, whose baseline success rate without EFN is only 5.3% while EFN raises it to 58.7% (see Table 2 in the main paper). The video `DrawerStore_without_EFN.mp4` was obtained after nearly twenty recording attempts, reflecting the low baseline success probability; even in this successful rollout, the robot exhibits hesitant behavior, including repeated grasps and an imprecise final release that almost leads to failure. In contrast, `DrawerStore_with_EFN.mp4` was recorded easily and shows a smooth, near-perfect execution of the same task, with a decisive grasp, accurate drawer interaction, and stable object placement. This qualitative gap is consistent with the quantitative improvement brought by EFN and provides an intuitive illustration of how experience feedback stabilizes behavior on the hardest real-world task.

References

[1] Qingwen Bu, Jisong Cai, Li Chen, Xiuqi Cui, Yan Ding, Siyuan Feng, Shenyuan Gao, Xindong He, Xuan Hu, Xu Huang, et al. Agibot world colosseo: A large-scale manipulation platform for scalable and intelligent embodied systems. *arXiv preprint arXiv:2503.06669*, 2025. 4

[2] Qingwen Bu, Yanting Yang, Jisong Cai, Shenyuan Gao, Guanghui Ren, Maoqing Yao, Ping Luo, and Hongyang Li. Univla: Learning to act anywhere with task-centric latent actions. *arXiv preprint arXiv:2505.06111*, 2025. 3, 4

[3] Marco Cuturi. Sinkhorn distances: Lightspeed computation of optimal transport. *Advances in neural information processing systems*, 26, 2013. 2

[4] Anirudh Goyal, Abram Friesen, Andrea Banino, Theophane Weber, Nan Rosemary Ke, Adria Puigdomenech Badia, Arthur Guez, Mehdi Mirza, Peter C Humphreys, Ksenia Konyushova, et al. Retrieval-augmented reinforcement learning. In *International Conference on Machine Learning*, pages 7740–7765. PMLR, 2022. 5

[5] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International conference on machine learning*, pages 1861–1870. Pmlr, 2018. 2

[6] Moo Jin Kim, Karl Pertsch, Siddharth Karamcheti, Ted Xiao, Ashwin Balakrishna, Suraj Nair, Rafael Rafailov, Ethan Foster, Grace Lam, Pannag Sanketi, et al. Openvla: An open-source vision-language-action model. *arXiv preprint arXiv:2406.09246*, 2024. 4

[7] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014. 3

[8] Bo Liu, Yifeng Zhu, Chongkai Gao, Yihao Feng, Qiang Liu, Yuke Zhu, and Peter Stone. Libero: Benchmarking knowledge transfer for lifelong robot learning. *Advances in Neural Information Processing Systems*, 36:44776–44791, 2023. 4

[9] Zhenxian Liu, Peixi Peng, and Yonghong Tian. Visual reinforcement learning with residual action. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 19050–19058, 2025. 5, 8