PEARL: PROCESS-EXPERIENCE AWARE REINFORCEMENT LEARNING FOR MULTI-TURN AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Reinforcement learning for multi-turn agents often relies on sparse outcome signals that indicate whether a trajectory succeeds, but provide limited guidance about which intermediate decisions should be corrected. This becomes especially problematic for failed trajectories, which may still contain many valid actions and therefore should not be treated as uniformly negative experience. We propose **PEARL** (Process-Experience Aware Reinforcement Learning), a framework that learns from recurring failure patterns across past interactions. Rather than penalizing an entire failed trajectory, PEARL identifies failure patterns that repeatedly appear in unsuccessful behavior and uses successful experience to remove patterns that are not uniquely associated with failure. The remaining failure-specific patterns are then used to provide more targeted negative supervision during reinforcement learning. On AppWorld, PEARL consistently improves over ARPO on both Normal and Challenge tasks at the 4B and 8B scales. On ALFWorld, PEARL achieves 97.9% mean success at 4B, outperforming the strongest baseline by 2.7 points, while remaining competitive at 8B. Extensive ablations and process analyses further validate the effectiveness of PEARL's selective failure-memory design.

1 INTRODUCTION

Outcome-reward reinforcement learning is well suited to multi-turn language agents because task completion can often be verified automatically. However, the supervision it provides is temporally coarse: a long trajectory with many intermediate decisions may receive only a single terminal reward. This makes it difficult to determine which local decisions should be corrected. Moreover, failed trajectories often contain useful actions, so treating the entire trajectory as negative experience can introduce misleading supervision.

A natural alternative is to learn from recurring local patterns associated with failure. Doing so is not straightforward. A behavior that appears in a failed trajectory is not necessarily responsible for the failure, and the same pattern may also occur in successful trajectories. In addition, the policy evolves during training, making older experience progressively less representative of the current policy. These observations suggest that useful failure memory should be selective rather than simply accumulate all past failures.

Our key idea is to identify local failure patterns from controlled outcome contrasts and reuse them across training. Branching rollout methods naturally provide such contrasts by generating alternative continuations from the same interaction history (Dong et al., 2026; Ji et al., 2026). When two branches lead to different outcomes, their divergence provides a local signal about which decisions distinguish the better and worse continuations. The challenge is to turn these one-off contrasts into reusable supervision without over-penalizing behaviors that are also compatible with success.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

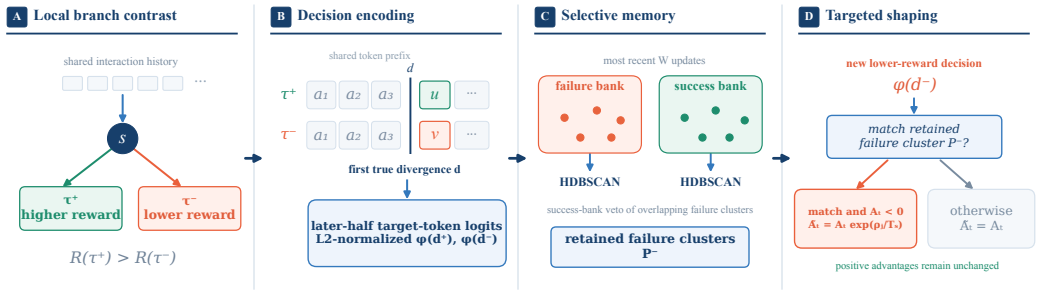

Figure 1: Overview of PEARL. Unequal-outcome branches provide local decision contrasts. Recurring decision patterns are stored in recent failure and success memories, success-side evidence filters ambiguous failure patterns, and retained failure patterns guide stronger negative updates.

We introduce **PEARL** (Process-Experience Aware Reinforcement Learning), illustrated in Figure 1. PEARL converts unequal-outcome branches into local training signals and aggregates recurring decision patterns in separate failure and success memories. Rather than treating every decision on a failed branch as wrong, it retains patterns that recur across trajectories and uses successful experience to remove patterns that are also common on the success side. Only decisions matching the remaining failure-specific patterns receive stronger negative updates, while positive updates are left unchanged. A sliding memory window further keeps the stored experience aligned with the current policy.

This design turns sparse trajectory-level outcomes into more targeted local supervision. Importantly, PEARL does not attempt to label individual actions as intrinsically correct or incorrect. Instead, it identifies recurrent decision patterns that are consistently associated with worse outcomes and remain distinguishable from successful behavior.

We evaluate PEARL on AppWorld (Trivedi et al., 2024) and ALFWorld (Shridhar et al., 2021) at both 4B and 8B model scales. On AppWorld, PEARL improves over ARPO on both Normal and Challenge tasks at both scales. On ALFWorld, PEARL achieves the best reported mean at 4B and remains competitive at 8B. Controlled ablations and process diagnostics further validate the main design choices and show how selective failure memory changes the shaping signals observed during training.

Our contributions are:

- We turn unequal-outcome branches from shared interaction histories into local training signals for multi-turn agent reinforcement learning.
- We introduce a selective failure-memory mechanism that reuses recurring failure patterns while filtering patterns also supported by successful behavior.
- We evaluate PEARL on AppWorld and ALFWorld at two model scales, with controlled ablations and process diagnostics that connect memory behavior to final task performance.

2 Related Work

Group-relative and branching agent RL. GRPO replaces a learned critic with within-group reward normalization and has become a common foundation for reasoning RL (Shao et al., 2024; Guo et al., 2025). Agent-oriented extensions exploit additional rollout structure: GiGPO combines within- and across-group comparisons (Feng et al., 2025); Tree-GRPO estimates intra-tree and inter-tree advantages from shared-prefix rollouts; and ARPO allocates branches using uncertainty changes along an interaction. Sequence-level normalization and strategic trajectory abstractions provide related ways to restructure the optimization signal (Zheng et al., 2025; Xue et al., 2026). These methods determine which trajectories to sample or how their current rewards become advantages. In contrast, PEARL uses the resulting branches to build a cross-update memory of recurrent decisions and remains an add-on to the underlying branching optimizer.

Credit assignment from sparse outcome rewards. Long-horizon agent training must infer useful local updates from trajectory-level outcomes. Recent approaches improve this inference through future-sensitive objectives, selective eligibility traces, information gain, entropy balancing, or learned process abstractions (Ma et al., 2026; Mou et al., 2026; Wang et al., 2025; Dong et al., 2025). They reduce uniform sequence-level credit, but an individual trajectory still provides limited evidence that a local decision pattern recurs across lower-reward outcomes and is absent from higher-reward ones. PEARL addresses a different axis: it compares continuations from a shared branch point, aggregates matching decisions across updates, and applies extra weight only after recurrence and success-disambiguation tests.

Memory-based reward shaping and internal representations. The closest representation-level precursor is MEDS, which clusters layer-wise target-token logits and uses their recurrence for dynamic reward shaping (Liu et al., 2026). PEARL retains the useful idea that a token's logit trajectory across layers is a compact signature of a decision, but changes both supervision and memory semantics. Specifically, outcome-side labels come from discordant parent–branch pairs rather than whole-trajectory outcomes; regions start at the first true divergence; failure and success representations occupy separate banks; and success acts as an asymmetric filter before negative-only shaping. These labels identify candidate update locations, not ground-truth local errors. HDBSCAN (McInnes et al., 2017) permits an unknown number of patterns and leaves low-density observations as noise, while a sliding window limits the lifetime of representations collected under older policies. Thus, PEARL is a training-time memory for selective credit assignment rather than an inference-time store of demonstrations or reflections.

3 METHOD

3.1 OVERVIEW

For a prompt x, an agent policy π_θ generates a multi-turn trajectory $\tau = (a_1, o_1, \ldots, a_T, o_T)$ and receives a verifiable terminal reward $R(\tau)$. Our base optimizer is ARPO, which samples multiple trajectories for the same prompt and may branch from an intermediate state to explore alternative continuations.

PEARL augments ARPO with a cross-update memory that turns recurring local failure patterns into additional training signals. As illustrated in Figure 1, the method has four stages. First, branches with different outcomes provide local failure–success comparisons. Second, each divergent decision is represented by a compact decision-aligned feature. Third, recent failure and success features are clustered separately, and success-side evidence is used to filter ambiguous failure patterns. Finally, negative advantages are strengthened only for decisions matching the retained failure patterns.

3.2 LOCAL SUPERVISION FROM DISCORDANT BRANCHES

A terminal failure reward does not reveal which local decisions should receive stronger negative updates. Branches from the same saved state provide a more focused comparison because they share the same preceding interaction history.

We therefore use only parent–branch pairs whose terminal rewards differ. For a parent trajectory τ^p and its branch τ^b, we define the higher-reward continuation as successful and the lower-reward continuation as failed:

$$(\tau^+, \tau^-) = \begin{cases} (\tau^b, \tau^p), & R(\tau^b) > R(\tau^p), \\ (\tau^p, \tau^b), & R(\tau^p) > R(\tau^b). \end{cases} \tag{1}$$

Pairs with equal outcomes contribute no memory entries.

Because the two continuations may still share tokens after branching, we locate their first actual token difference d and use only the remainder of that action as the local comparison region. Pairs without a valid divergence are skipped. This removes the shared prefix from supervision and focuses the comparison on where the two continuations begin to differ.

3.3 DECISION-ALIGNED LAYER-LOGIT REPRESENTATION

We next encode the decision at the first divergence. Instead of clustering full hidden states, which contain substantial contextual information unrelated to the selected action, we represent how strongly different layers support the token that is actually chosen.

Let y_d denote the token at divergent position d, $h_{d-1}^{(\ell)}$ the hidden state used to predict it at layer ℓ, and W_{y_d} the corresponding output-head row. We compute

$$z_d^{(\ell)} = \left\langle h_{d-1}^{(\ell)}, W_{y_d} \right\rangle, \tag{2}$$

and concatenate the target-token logits from the latter half of an L-layer model:

$$\phi(d) = \mathrm{L2Norm}\left([z_d^{(L/2)}, \ldots, z_d^{(L-1)}] \right). \tag{3}$$

For Qwen3-4B-Instruct-2507, $L = 36$, yielding an 18-dimensional representation compared with the 2,560-dimensional final hidden state. The resulting feature captures how support for the selected token develops across later layers while remaining compact for density-based clustering. Section 4.3 compares this representation with several alternatives.

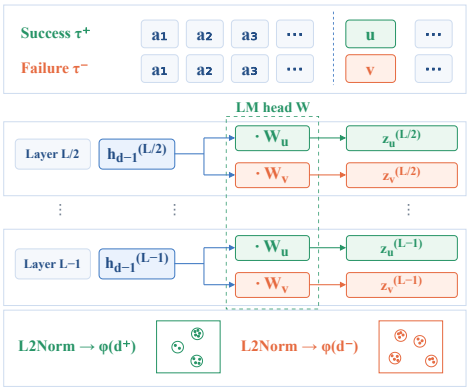

Figure 2: Decision representation at the first divergence. Each branch scores its selected token across the latter half of the network, producing a compact layer-logit feature for clustering.

3.4 CLUSTERING AND FILTERING RECENT DECISION PATTERNS

We maintain separate failure and success banks to identify recurring decision patterns. At update s, each discordant pair contributes a feature $\phi(d^-)$ from the lower-reward branch and $\phi(d^+)$ from the higher-reward branch:

$$\phi(d^-) \to \mathcal{B}_s^-, \qquad \phi(d^+) \to \mathcal{B}_s^+.$$

Because the policy changes during training, both banks retain only recent observations. Entries older than W updates are removed. We periodically fit HDBSCAN to the L2-normalized features in each bank, allowing recurrent patterns to form clusters while isolated observations remain noise. We use $W = 25$ and refit every $K = 10$ updates by default. The default clustering configuration is MCS10/MS5.

Recurring failure patterns are not automatically suitable for shaping, since the same behavior may also appear in successful trajectories. We therefore use the success bank to filter the failure clusters. Let $\mathcal{C}_s^- = \{C_j^-\}$ denote the failure clusters and μ_j^- the centroid of cluster j. Under the default hard filter, a failure cluster is retained only if its centroid is not assigned to a success cluster:

$$\mathcal{P}_s^- = \left\{ C_j^- \in \mathcal{C}_s^- : \mathrm{predict}_{\mathcal{B}_s^+}(\mu_j^-) = -1 \right\}. \tag{4}$$

This filtering is asymmetric: successful behavior is used to reject ambiguous failure patterns, while failure patterns do not remove success clusters.

We also consider a soft alternative that avoids removing an entire failure cluster. For a query feature ϕ, let $p_-(\phi)$ and $p_+(\phi)$ denote its HDBSCAN membership strengths in the failure and success banks. We define

$$q(\phi) = \max\{p_-(\phi) - p_+(\phi), 0\}. \tag{5}$$

This score reduces the shaping strength when a decision is also strongly supported by successful behavior. The default method uses hard filtering; Section 4.3 compares the two strategies.

3.5 SELECTIVE NEGATIVE-ADVANTAGE SHAPING

The retained failure clusters provide additional supervision for recurring lower-reward decision patterns. We use them only to strengthen negative updates, while leaving positive advantages unchanged.

For a current lower-reward decision $\phi(d^-)$ that matches retained failure cluster $C_j^- \in \mathcal{P}_s^-$, we define its relative cluster support as

$$\rho_j = \frac{|C_j^-|}{\sum_{C_k^- \in \mathcal{P}_s^-} |C_k^-|}. \tag{6}$$

Let A_t denote the original token advantage and T_s the shaping temperature, which is set to 1 by default. For tokens from the localized divergence d^- to the end of the corresponding action, we use

$$\widetilde{A}_t = \begin{cases} A_t \exp(\rho_j / T_s), & A_t < 0 \text{ and } \phi(d^-) \in C_j^-, \\ A_t, & \text{otherwise.} \end{cases} \tag{7}$$

Tokens before the divergence and all positive advantages remain unchanged.

Under soft filtering, the exponent is additionally scaled by $q(\phi(d^-))$ from Equation 5.

The base clipped policy-gradient objective is then applied using $\widetilde{A}_t$:

$$\mathcal{L}_{\text{PG}} = -\mathbb{E}_t \Big[\min \Big(r_t \widetilde{A}_t, \text{clip}(r_t, 1-\epsilon, 1+\epsilon) \widetilde{A}_t \Big) \Big], \tag{8}$$

where $r_t = \pi_\theta(y_t \mid s_t) / \pi_{\theta_{\text{old}}}(y_t \mid s_t)$.

Thus, PEARL leaves the underlying policy optimization procedure unchanged and modifies only the magnitude of selected negative updates.

4 EXPERIMENTS

We evaluate PEARL on AppWorld and ALFWorld, followed by controlled ablations and process diagnostics.

4.1 EXPERIMENTAL SETUP

Benchmarks and models. We evaluate Qwen3-4B-Instruct-2507 and Qwen3-8B (Yang et al., 2025) on two multi-turn agent benchmarks. AppWorld evaluates interactive coding agents on Normal and more compositional Challenge tasks over simulated applications, while ALFWorld evaluates text agents on Seen and Unseen embodied household tasks. Success rate is the primary metric for both benchmarks. The agent prompts for both environments are provided in Appendix C.

Baselines. We compare with GRPO, GiGPO, Tree-GRPO, and ARPO under the same model and environment infrastructure. PEARL is built on ARPO because its branching rollouts provide the matched local comparisons required by Section 3.2. GRPO, GiGPO, ARPO, and PEARL use 16 rollouts per prompt; Tree-GRPO follows its tree-expansion setting and produces 24 physical leaves per prompt group.

Training and evaluation protocol. We evaluate AppWorld after 400 optimizer updates and ALFWorld after 300 updates. All main experiments use three training seeds, and we report the mean and standard deviation across seeds. The default PEARL configuration uses MCS10/MS5, $W = 25$, $K = 10$, $T_s = 1$, hard success-bank filtering, and failure-only advantage shaping. Unless otherwise specified, ablations are conducted on AppWorld with Qwen3-4B-Instruct-2507 using a single controlled run and evaluated at step 400.

4.2 MAIN RESULTS

Table 1 summarizes the main results. On AppWorld, PEARL consistently improves over ARPO at both model scales: by 3.67/1.16 points on Normal/Challenge at 4B and by 4.56/1.40 points at 8B.

Table 1: Final task success rates (%) after 400 AppWorld or 300 ALFWorld updates. Each "Mean" equally weights the two splits. Trained-method values are mean ± standard deviation over three runs; Base Model is the untrained model. Bold and underline mark the best and second-best distinct values within each model scale, with ties formatted alike.

	AppWorld			ALFWorld		
Method	Normal	Challenge	Mean	Seen	Unseen	Mean
			Qwen3-4B-Instruct-2507			
Base Model	8.93	5.60	7.26	38.1	33.1	35.6
GRPO	52.38±0.79	24.38±0.16	38.38±0.48	94.9±2.8	94.3±5.7	94.6±4.2
ARPO	<u>53.37</u>±2.18	<u>29.02</u>±2.56	<u>41.19</u>±2.37	95.1±0.4	92.3±0.8	93.7±0.2
GiGPO	44.84±4.76	24.02±0.52	34.43±2.64	<u>96.0</u>±2.0	<u>94.4</u>±3.4	<u>95.2</u>±2.7
Tree-GRPO	36.41±1.49	18.98±0.04	27.70±0.72	85.6±10.1	79.9±14.2	82.7±12.1
PEARL	**57.04**±0.69	**30.18**±0.84	**43.61**±0.77	**98.0**±0.6	**97.9**±0.4	**97.9**±0.5
			Qwen3-8B			
Base Model	11.31	4.48	7.89	57.6	55.5	56.5
GRPO	43.95±0.10	20.26±2.84	32.11±1.47	94.9±1.8	<u>96.4</u>±1.1	95.6±1.5
ARPO	<u>50.60</u>±1.98	<u>23.70</u>±3.96	<u>37.15</u>±2.97	<u>95.5</u>±0.3	95.4±0.6	95.4±0.2
GiGPO	45.24±4.37	17.91±0.08	31.57±2.14	**97.7**±0.9	**98.0**±0.0	**97.9**±0.4
Tree-GRPO	42.16±2.48	16.67±0.76	29.41±1.62	95.0±0.2	93.8±0.3	94.4±0.2
PEARL	**55.16**±1.98	**25.10**±1.76	**40.13**±1.87	**97.7**±1.2	<u>96.4</u>±1.9	<u>97.1</u>±1.6

On ALFWorld, PEARL achieves the highest 4B mean at 97.9%, 2.7 points above the strongest baseline, and reaches 97.1% at 8B, within 0.8 points of the best result. Figure 3 shows the corresponding 4B evaluation trajectories throughout training.

4.3 Ablations

We next examine the main design choices of PEARL, including shaping direction, shaping location, success-bank filtering, memory window, clustering granularity, and clustering features.

Effect of shaping direction. Figure 4(a) varies the shaping direction without cross-bank filtering. Failure-only shaping achieves 52.58%/27.98% on Normal/Challenge, compared with 44.64%/25.42% for success-only shaping and 47.02%/24.78% for bidirectional shaping. Adding success shaping to failure shaping lowers performance by 5.56/3.20 points. These results support PEARL's negative-only shaping design, as successful trajectories already receive positive reward-based updates.

Success evidence helps disambiguate failure patterns. Patterns observed on failed branches may also occur in successful behavior and are therefore poor targets for additional negative shaping. PEARL uses the success bank to filter these overlapping patterns. As shown in Figure 4(b), enabling this filter improves Normal/Challenge performance by 5.16/3.04 points over no filtering. This result supports using successful experience to determine which failure patterns should be retained for shaping. Section 4.4.1 examines a concrete example of this filtering.

Pattern-guided shaping outperforms random shaping. To test whether the selected shaping locations matter, we replace PEARL's pattern-matched locations with random locations while keeping the shaping budget and multipliers unchanged. At step 400, pattern-guided shaping achieves 57.74%/31.02% on Normal/Challenge, compared with 49.17%/25.37% for random shaping. The 8.57/5.65-point gains show that PEARL benefits from where negative shaping is applied, rather than from additional negative weight alone.

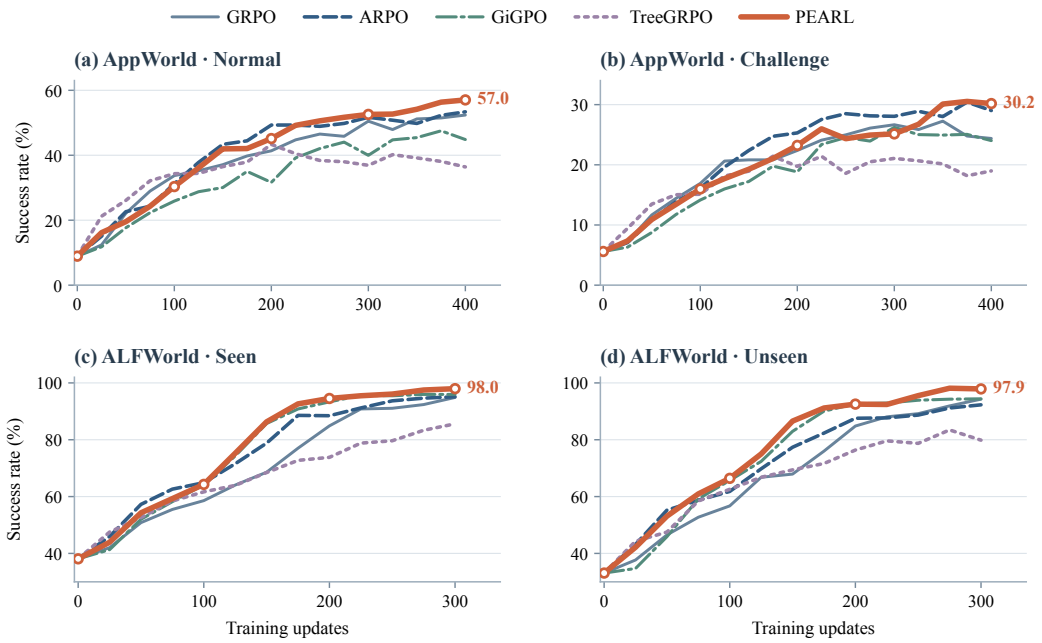

Figure 3: Qwen3-4B-Instruct-2507 evaluation success rates (%) on AppWorld (top, through 400 updates) and ALFWorld (bottom, through 300 updates). Step 0 is the shared pre-training evaluation; subsequent points, spaced every 25 updates, are averages across the available training runs. Orange endpoint labels show PEARL's final scores reported in Table 1.

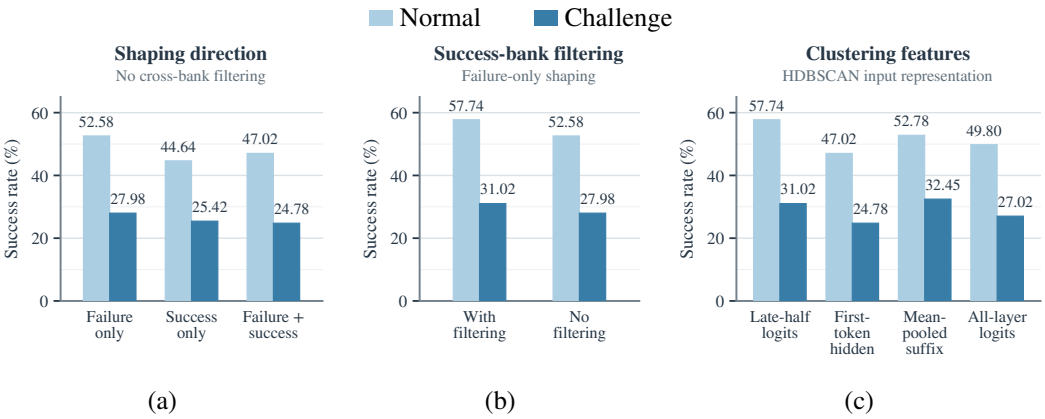

Figure 4: Single-run AppWorld success rates (%) at step 400 with Qwen3-4B-Instruct-2507. (a) Failure-only, success-only, and bidirectional shaping without cross-bank filtering. (b) With versus without success-bank filtering, holding failure-only shaping and MCS10/MS5 fixed. (c) Four HDBSCAN input representations, holding all other settings fixed. Light and dark bars denote Normal and Challenge, respectively.

Recent memory outperforms unlimited history. Extending the hard-filter memory window from 25 updates to unlimited history ($W = \infty$) reduces Normal performance from 57.74% to 50.00% and Challenge performance from 31.02% to 26.38%. Thus, retaining more historical experience does not necessarily provide a better shaping signal. Section 4.4 examines how the memory window affects shaping activity during training.

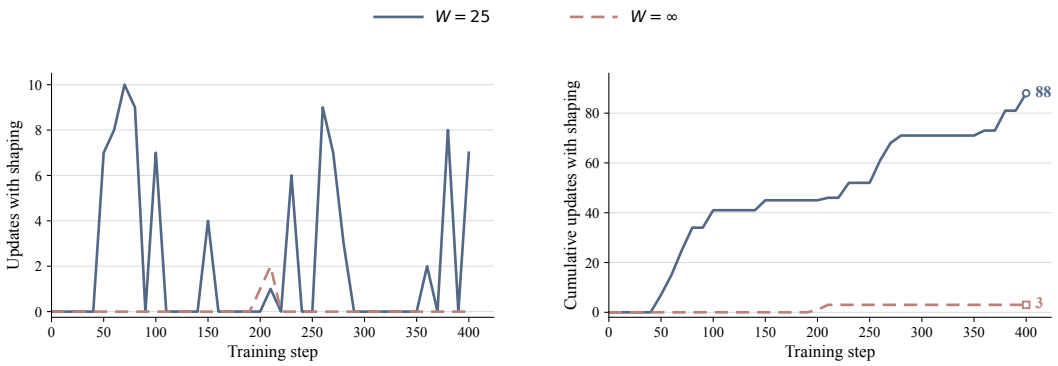

Figure 5: Failure-shaping activity with a 25-update window and unlimited history ($W = \infty$) on AppWorld (Qwen3-4B-Instruct-2507, hard filtering, MCS10/MS5; one run per setting). Left: number of updates with active failure shaping in each consecutive block of 10 updates. Right: cumulative number of updates with active failure shaping, reaching 88 for $W = 25$ and 3 for unlimited history.

Cluster granularity interacts with filtering. Table 2 compares MCS10/MS5 and MCS5/MS3 under hard and soft filtering. With hard filtering, moving to the finer MCS5/MS3 setting decreases Normal/Challenge performance by 10.52/10.80 points. With soft filtering, the same change instead improves performance by 2.38/8.63 points. The effect of finer clustering therefore depends strongly on how the resulting patterns are filtered. Section 4.4 examines the corresponding shaping activity.

Table 2: AppWorld success rate (%) by clustering granularity and filtering for Qwen3-4B-Instruct-2507 (step 400; one run per setting).

Granularity	Normal	Challenge
Hard filter		
MCS10/MS5	**57.74**	31.02
MCS5/MS3	47.22	20.22
Soft filter		
MCS10/MS5	51.19	25.98
MCS5/MS3	53.57	**34.61**

Ablation on clustering features. We compare four representations for HDBSCAN while keeping all other settings fixed: late-half target logits, first-token hidden states, mean-pooled suffix hidden states, and all-layer target logits. All variants use MCS10/MS5 and Euclidean distance on L2-normalized features.

As shown in Figure 4(c), late-half target logits achieve 57.74% on Normal and 31.02% on Challenge. They give the best Normal result while remaining within 1.43 points of the best Challenge result. Compared with first-token hidden states, late-half logits improve Normal/Challenge by 10.72/6.24 points, and compared with all-layer logits by 7.94/4.00 points. Mean-pooled hidden states obtain the highest Challenge score, but their Normal performance drops to 52.78%. Overall, late-half target logits provide the strongest balance across the two splits and are therefore used as the default clustering feature.

4.4 Process diagnostics: how does PEARL use its memory?

The ablations above establish how PEARL's design choices affect final performance. We next examine how memory retention and filtering translate into shaping activity, and why the resulting update is negative-only.

Finite history preserves more usable failure signals. Figure 5 shows that $W = 25$ activates failure shaping in 88 of 400 updates, compared with only 3 under unlimited history. Unlimited history stores failures, but accumulated successful behavior vetoes many clusters; even retained clusters shape only when a new decision matches. Thus, more stored entries need not provide more usable shaping signals.

Soft filtering makes fine-grained clusters more usable. Table 5 shows similar failure evidence under both filters, but retained clusters are available in 6% versus 90% of late-training updates, and shaping activates in 2% versus 46% (hard versus soft). Soft filtering preserves more fine-grained patterns while attenuating success-overlapping matches, consistent with its stronger Challenge result in Table 2.

Failure shaping provides more targeted supervision. PEARL applies extra negative weight only to decisions that match recurrent failure-side patterns, leaving positive advantages unchanged. Figure 4(a) supports this asymmetry: the success bank is more useful for vetoing ambiguous failure patterns than for supplying extra positive weight.

4.4.1 Case study: success evidence vetoes shared behavior

A failed trajectory can contain useful shared actions. Table 3 contrasts successful and failed AppWorld payment trajectories to show how success memory vetoes a lookup shared by both outcomes.

Table 3: AppWorld, Qwen3-4B-Instruct-2507, training step 400. Two behaviors from each cluster exhibit the same API-inspection pattern; success cluster S_1 filters failure cluster F_1.

Cluster	Trajectory behavior and outcome
Success S_1	**Behavior 1: friends and roommates.** Inspect the contact-relationship API specification; receive a valid schema. Later, collect the matching requests before denying them, and reject all 14. The trajectory succeeds.
	Behavior 2: siblings. Inspect the same API specification; receive a valid schema. Later, check both transaction senders and receivers and like all 8 matching transactions. The trajectory succeeds.
Failure F_1	**Behavior 1: friends and roommates.** Inspect the same specification; receive a valid schema. **Later failure:** Deny requests while paging through the pending list; reject only 13 of 14, missing request 1844. Modifying the list during traversal is consistent with a pagination skip.
	Behavior 2: siblings. Inspect the same specification; receive a valid schema. **Later failure:** Check only whether the sender is a sibling, ignoring sibling receivers. Like only 2 of the 8 matching transactions.
Shared pattern. Inspect the contact-relationship API specification before identifying people by relationship. **Memory decision.** S_1 **filters** F_1**.** The same specification-inspection behavior occurs in successful trajectories. The corresponding failure cluster is excluded from additional negative shaping.	

The failed paths diverge only after this shared lookup. Success memory therefore vetoes its extra negative shaping; Appendix Table 6 gives an ALFWorld counterpart.

5 Conclusion

We introduced PEARL to turn sparse outcome feedback into selective local supervision for multi-turn agents. Rather than assigning extra blame to every decision in a failed trajectory, PEARL stores recurring failure-side patterns, uses successful experience to veto patterns shared across outcomes, and strengthens negative updates only where a retained pattern matches. Across AppWorld and ALFWorld, experiments at two model scales show improvements over the underlying optimizer, while ablations and bank diagnostics support the roles of targeted shaping, success disambiguation, and recent memory. These findings suggest that outcome-level feedback becomes more informative when past decisions are selected for reuse, without requiring a locally verified error label for every action.

AI USE STATEMENT

Generative AI assistants, including OpenAI Codex and Claude, were used to assist with manuscript drafting and editing, figure and table preparation, research scripting and code inspection, and exploratory analysis of experimental diagnostics. AI-generated content was not treated as independent scientific evidence. The authors reviewed and verified all AI-assisted work and take full responsibility for the methods, experimental results, data provenance, citations, and claims presented in the paper.

ETHICS STATEMENT

This work studies reinforcement learning for agents in simulated software and household environments and does not involve human subjects or private user data. More capable tool-using agents may nevertheless introduce security and misuse risks when deployed in real-world systems. Our experiments are restricted to benchmark environments; practical deployment should incorporate appropriate permission boundaries, action auditing, and human oversight.

REPRODUCIBILITY STATEMENT

Section 3 specifies the branch-pair construction, decision representation, memory filtering, and advantage-shaping rules. Section 4 reports the model scales, training and evaluation protocol, evaluation checkpoints, and default hyperparameters, while Appendix A provides additional implementation details and aggregate results. Together, these descriptions are intended to make the experimental setup, training procedure, and reported analyses sufficiently clear for reproduction and comparison.

REFERENCES

Guanting Dong, Licheng Bao, Zhongyuan Wang, Kangzhi Zhao, Xiaoxi Li, Jiajie Jin, Jinghan Yang, Hangyu Mao, Fuzheng Zhang, Kun Gai, et al. Agentic entropy-balanced policy optimization. *arXiv preprint arXiv:2510.14545*, 2025.

Guanting Dong, Hangyu Mao, Kai Ma, Licheng Bao, Yifei Chen, Zhongyuan Wang, Zhongxia Chen, Jiazhen Du, Huiyang Wang, Fuzheng Zhang, Guorui Zhou, Yutao Zhu, Ji-Rong Wen, and Zhicheng Dou. Agentic reinforced policy optimization. In *International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=TX4k7BF6aO.

Lang Feng, Zhenghai Xue, Tingcong Liu, and Bo An. Group-in-group policy optimization for llm agent training. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

Yuxiang Ji, Ziyu Ma, Yong Wang, Guanhua Chen, Xiangxiang Chu, and Liaoni Wu. Tree search for LLM agent reinforcement learning. In *International Conference on Learning Representations*, 2026.

Yang Liu, Enxi Wang, Yufei Gao, Weixin Zhang, Bo Wang, Zhiyuan Zeng, Yikai Zhang, Yining Zheng, and Xipeng Qiu. The past is not past: Memory-enhanced dynamic reward shaping. *arXiv preprint arXiv:2604.11297*, 2026.

Chiyu Ma, Shuo Yang, Kexin Huang, Jinda Lu, Haoming Meng, Shangshang Wang, Bolin Ding, Soroush Vosoughi, Guoyin Wang, and Jingren Zhou. Fipo: Eliciting deep reasoning with future-kl influenced policy optimization. *arXiv preprint arXiv:2603.19835*, 2026.

Leland McInnes, John Healy, Steve Astels, et al. hdbscan: Hierarchical density based clustering. *J. Open Source Softw.*, 2(11):205, 2017.

Chaoli Mou, Zhan Zhuang, Xinning Chen, and Yu Zhang. Beyond uniform credit assignment: Selective eligibility traces for rlvr. *arXiv preprint arXiv:2605.05965*, 2026.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Cote, Yonatan Bisk, Adam Trischler, and Matthew Hausknecht. Alfworld: Aligning text and embodied environments for interactive learning. In *International Conference on Learning Representations*, 2021.

Harsh Trivedi, Tushar Khot, Mareike Hartmann, Ruskin Manku, Vinty Dong, Edward Li, Shashank Gupta, Ashish Sabharwal, and Niranjan Balasubramanian. Appworld: A controllable world of apps and people for benchmarking interactive coding agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 16022–16076, 2024.

Guoqing Wang, Sunhao Dai, Guangze Ye, Zeyu Gan, Wei Yao, Yong Deng, Xiaofeng Wu, and Zhenzhe Ying. Information gain-based policy optimization: A simple and effective approach for multi-turn llm agents. *arXiv preprint arXiv:2510.14967*, 2025.

Xiangyuan Xue, Yifan Zhou, Zidong Wang, Shengji Tang, Philip Torr, Wanli Ouyang, Lei Bai, and Zhenfei Yin. Strata: Incentivizing agentic reinforcement learning with strategic trajectory abstraction. *arXiv preprint arXiv:2605.06642*, 2026.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Chujie Zheng, Shixuan Liu, Mingze Li, Xiong-Hui Chen, Bowen Yu, Chang Gao, Kai Dang, Yuqiong Liu, Rui Men, An Yang, et al. Group sequence policy optimization. *arXiv preprint arXiv:2507.18071*, 2025.

A Additional Experimental Details

A.1 Default training configuration

We train PEARL with Qwen3-4B-Instruct-2507 (4B) and Qwen3-8B (8B) for 300 updates on ALFWorld and 400 updates on AppWorld. Each update uses eight task prompts and 16 trajectories per prompt, for a global batch of 128 trajectories. ARPO starts with eight trajectories per prompt and allocates the remaining rollout budget through branching, with one branch per trigger.

Rollouts allow up to 50 agent turns on ALFWorld and 40 on AppWorld, with a response limit of 4,096 tokens per turn. The context limit for the accumulated interaction is 65,536 tokens for 4B models and 32,768 tokens for 8B models in both environments. Training uses sampling temperature 1.0 and top-$p = 1.0$, without top-k truncation.

All four configurations use Adam with a constant learning rate of 10^{-6}, weight decay 0.1, and $(\beta_1, \beta_2) = (0.9, 0.98)$. They use GRPO advantages, symmetric PPO clipping at 0.2, and an entropy coefficient of zero. The branching entropy probe uses 20 tokens, top-$k = 128$, $\alpha = 0.5$, $\beta = 0.2$, and threshold 0.5.

Evaluation uses a 4,096-token response limit and three greedy samples per task (top-$k = 1$). The evaluation curves in Figure 3 include checkpoints at 25-update intervals. ALFWorld uses the Seen and Unseen validation splits, and AppWorld uses the Normal and Challenge test splits. ALFWorld assigns reward 1 for task success and disables the additional invalid-action penalty. The environment prompts and action formats are given in Appendix C.

The default PEARL banks use HDBSCAN minimum cluster size 10, minimum samples 5, a 25-update window, and a reclustering interval of 10 updates. Failure shaping is enabled, success shaping is disabled, the shaping temperature is 1.0, and hard success-overlap filtering is used. The code-level shaping-strength argument is an enable/disable gate in the current implementation; the exponential multiplier itself follows Equation 7. All four configurations use counterfactual bank labels derived from unequal-outcome parent–branch pairs. The 0.3 failure/success selection ratios retained in the ALFWorld launchers are legacy settings for the alternative turn-selection path; counterfactual labeling uses the paired regions directly.

A.2 Process diagnostics

Unlimited-window bank evolution. Table 4 shows that a growing failure bank does not consistently produce shaping-active matches under unlimited history: retained clusters vary across the snapshots, and retention alone does not guarantee a matching query.

Table 4: Failure- and success-bank snapshots from the AppWorld hard-filter MCS10/MS5 run with unlimited history ($W = \infty$). "Retained entries" counts entries in retained failure clusters, not the number of clusters.

Step	Failure entries	Success entries	Retained failure clusters	Retained entries	Filtered clusters	Success clusters
100	297	347	0	0	2	2
200	716	798	1	19	2	2
300	1106	1195	0	0	5	4
400	1368	1509	1	1200	1	15

Hard versus soft filtering with small clusters. Statistics in Table 5 aggregate training updates 301–400. The two filters add nearly the same number of failure entries, but differ in how many failure clusters remain available for shaping.

New entries are counted from bank writes; cluster and shaping statistics come from training logs. Count rows are means per update; percentage rows give the fraction of updates satisfying the stated condition.

Table 5: AppWorld bank use over updates 301–400 with MCS5/MS3 clustering (one run per filter).

Statistic	Hard	Soft
New failure entries / update	3.34	3.35
Retained failure clusters / update	0.06	2.40
Updates with retained failure clusters	6%	90%
Updates with active failure shaping	2%	46%
Clusters discarded / update	2.25	0.00

B CASE STUDIES OF SELECTIVE FAILURE MEMORY

We examine three additional memory decisions from Qwen3-4B-Instruct-2507 trained with PEARL. The AppWorld success-veto case appears in Table 3 of Section 4.4.1; here we show an ALFWorld success-veto case and retained failure-side patterns from both environments. The immediate environment responses in the retained examples provide post-hoc audit evidence, not labels available to PEARL when retaining a cluster.

B.1 ALFWORLD: OPENING A CLOSED CABINET TO SEARCH FOR THE TARGET

At training step 200, success cluster S_1 filters failure cluster F_0. Table 6 shows object-search decisions in tasks requiring a cooled cup or bread to be placed on a dining table, or a cooled pot on a stove burner. All four decisions respond to a closed cabinet with the same action, `open cabinet 1`.

Table 6: ALFWorld, Qwen3-4B-Instruct-2507, training step 200. Both clusters contain the displayed closed-cabinet opening behavior; success cluster S_1 filters failure cluster F_0.

Cluster	Trajectory behavior and outcome
Success S_1	**Behavior 1: finding a cup.** Observe that cabinet 1 is closed; open it to check for the cup. Action: `open cabinet 1`. The cabinet is empty. The trajectory succeeds.
	Behavior 2: finding bread. Observe that cabinet 1 is closed; open it to check for bread. Action: `open cabinet 1`. The cabinet contains a plate. The trajectory succeeds.
Failure F_0	**Behavior 1: finding a cup.** Open the closed cabinet to check for the cup. Action: `open cabinet 1`. **Later failure:** Cool and place a `mug` instead of the required `cup`; then repeatedly look around or check inventory until the 50-turn limit.
	Behavior 2: finding a pot. Open the closed cabinet to check for the pot. Action: `open cabinet 1`. **Later failure:** Revisit searched locations without checking stoveburner 2, where a successful trajectory finds the pot. Exhaust 50 turns without acquiring or cooling it.
Shared pattern. When a candidate cabinet is closed, open it to inspect its contents for the target object.	
Memory decision. S_1 **filters** F_0**.** The same closed-cabinet opening behavior occurs in successful trajectories. The corresponding failure cluster is excluded from additional negative shaping.	

The subsequent failures involve selecting the wrong object or repeatedly searching without visiting the target location. Opening the initial cabinet is a shared information-gathering action, distinct from these later mistakes. Success memory filters this common opening behavior, illustrating why the final outcome alone is insufficient to identify which local decisions warrant additional negative shaping.

B.2 RETAINED PATTERNS IN APPWORLD AND ALFWORLD

Table 7 shows two failure-side clusters that remain eligible for shaping: unauthenticated alarm queries in AppWorld and unavailable lamp-inspection commands in ALFWorld. The API and action-validity responses are post-hoc audit observations, not labels used by PEARL to retain either cluster.

Table 7: Retained failure-side patterns in Qwen3-4B-Instruct-2507 runs. The two rows report separate bank snapshots and two audited decisions per environment.

Environment and cluster	Audited decisions, observed responses, and bank decision
AppWorld step 200 F_1	**Delay alarm by one hour:** call `apis.phone.show_alarms()` without an access token; the API returns HTTP 401 (missing, invalid, or expired authorization). **Delay alarm by twenty minutes:** make the same unauthenticated call; the API again returns HTTP 401. **Matched pattern:** invoke a protected API without satisfying its authentication requirement. **Bank decision:** retained because the success bank does not veto F_1.
ALFWorld step 300 F_0	**After turning on the lamp:** issue `examine desklamp 1`, which is outside the admissible set; the environment replies "Invalid action." **After a rejected inspection:** following a rejected `look under desklamp 1`, issue `examine desklamp 1` and again receive "Invalid action." **Matched pattern:** use an unavailable lamp-inspection command. **Bank decision:** retained because no success cluster has formed to veto F_0.

In these sampled traces, immediate rejection feedback helps interpret the local decisions, but PEARL does not use that feedback to assign error labels. Retention only leaves later matching decisions with negative advantage eligible for stronger negative shaping; it does not establish that a pattern caused the final failure.

C AGENT PROMPTS FOR ALFWORLD AND APPWORLD

This section records the agent-facing prompt templates used for the two environments. Braced fields are filled at runtime; the text below preserves the wording of the templates. The model-specific chat template supplies the role delimiters around these messages.

C.1 ALFWORLD PROMPT

ALFWorld presents one user message per turn. The `observation` field contains the current textual observation (including the task at the initial turn), and `admissible_actions` contains the current environment-provided commands, formatted as a comma-separated list of quoted strings. The agent reasons and returns one command inside `<action>` tags. The same template is reused after every valid action with updated fields.

ALFWorld user prompt

You are an expert agent operating in the ALFRED Embodied Environment.
Your current observation is: {observation}
Your admissible actions of the current situation are: [{admissible_actions}].

Now it's your turn to take an action.
You should first reason step−by−step about the current situation, then choose an admissible action for current step and present it within <action> </action> tags.

If the response lacks a valid tagged action or names a command outside the admissible set, the next user message instead reads:

ALFWorld invalid-action feedback

Invalid action. Use the <action> </action> format and select one of the admissible actions. Your admissible actions are: [{admissible_actions}].

C.2 APPWORLD PROMPT

AppWorld uses the following full prompt template, including a worked Spotify example and the rules preceding the actual task. The Jinja fields `main_user.*`, `app_descriptions`, and `instruction` are filled from the task environment. The `USER:` and `ASSISTANT:` markers are parsed into chat messages; the final user message gives the task. After each code execution, the next user message contains the returned output in a fenced block prefixed by `Output:`.

The Spotify demonstration in the archived template contains a non-executable password lookup: its list comprehension produces Boolean values, then indexes the first Boolean as a dictionary. We reproduce the template as used in the reported runs below. The intended executable lookup would be:

```
spotify_password = [
    account_password
    for account_password in passwords
    if account_password["account_name"] == "spotify"
][0]["password"]
```

AppWorld prompt template

USER:
I am your supervisor, and you are an AI Assistant whose job is to complete my day−to−day tasks fully autonomously.

To do this, you will need to interact with app(s) (e.g., spotify, venmo etc) using their associated APIs on my behalf. For this you will undertake a ∗multi−step conversation∗ using a python REPL environment. That is, you will write the python code, the environment will execute it and show you the result, based on which, you will write python code for the next step and so on, until you've achieved the goal. This environment will let you interact with app(s) using their associated APIs on my behalf.

Here are three key APIs that you need to know to get more information

To get a list of apps that are available to you.

```python
print(apis.api_docs.show_app_descriptions())
```

To get the list of APIs under any app listed above, e.g. spotify

```python
print(apis.api_docs.show_api_descriptions(app_name='spotify'))
```

To get the specification of a particular api, e.g. spotify app's login api

AppWorld prompt template (continued)

```
```python
print(apis.api_docs.show_api_doc(app_name='spotify', api_name='login'))
```

Each code execution will produce an output that you can use in subsequent calls. Using
    these APIs, you can now generate code, that I will execute, to solve the task. Let's start
    with the task

My name is: {{ main_user.first_name }} {{ main_user.last_name }}. My personal email is
    {{ main_user.email }} and phone number is {{ main_user.phone_number }}.
Task: How many playlists do I have in my Spotify playlist library?

ASSISTANT:
Okay. Let's first find which APIs are available to use in Spotify.

```python
print(apis.api_docs.show_api_descriptions(app_name='spotify'))
```

USER:
Output:
```
[
 ...
 "login : Login to your account.",
 "logout : Logout from your account.",
 ...
]
```

ASSISTANT:
Okay. Looks like I can use the `login` api. Let's find its specifications.

```python
print(apis.api_docs.show_api_doc(app_name='spotify', api_name='login'))
```

USER:
Output:
```
{
 "app_name": "spotify",
 "api_name": "login",
 "path": "/spotify/auth/token",
 "method": "POST",
 "description": "Login to your account.",
 "parameters": [
  {
   "name": "username",
   "type": "string",
   "required": true,
   "description": "Your account email.",
   "default": null,
   "constraints": []
  },
```
```

AppWorld prompt template (continued)

````
 {
  "name": "password",
  "type": "string",
  "required": true,
  "description": "Your account password.",
  "default": null,
  "constraints": []
 }
],
"response_schemas": {
 "success": {
  "access_token": "string",
  "token_type": "string"
 },
 "failure": {
  "message": "string"
 }
}
}
```

ASSISTANT:
I need the supervisor's username and password. Let's see if any app can help me get that.

```python
print(apis.api_docs.show_app_descriptions())
```

USER:
Output:
```
{{ app_descriptions }}
```

ASSISTANT:
Looks like the supervisor app could help me with that.

```python
print(apis.api_docs.show_api_descriptions(app_name='supervisor'))
```

USER:
Output:
```
[
 ...
 "show_account_passwords : Show your supervisor's account passwords."
 ...
]
```

ASSISTANT:
I can use `show_account_passwords` to get the passwords. So I will call that and save it in a variable for future reference.
````
````

AppWorld prompt template (continued)

```
```python
passwords=apis.supervisor.show_account_passwords()
print(passwords)
```

USER:
Output:
```
[
 {
  "account_name": "spotify",
  "password": "dummy_spotify_pass"
 },
 {
  "account_name": "file_system",
  "password": "dummy_fs_pass"
 },
 ...
]
```

ASSISTANT:
So the Spotify password is an entry in the `passwords` list with the account_name=spotify. I can use it to login now with the provided email address: {{ main_user.email }}.

```python
spotify_password = [account_password["account_name"] == "spotify" for account_password in passwords][0]["password"]
login_result = apis.spotify.login(username='{{ main_user.email }}', password=spotify_password)
print(login_result)
```

USER:
Output:
```
{
 "access_token": "fake_access_token",
 "token_type": "Bearer"
}
```

ASSISTANT:
First, let's check the available APIs for the Spotify app.

```python
print(apis.api_docs.show_api_descriptions(app_name='spotify'))
```

USER:
Output:
```
[
 ...
 {
```
```

AppWorld prompt template (continued)

```
 "name": "show_playlist_library",
 "description": "Search or show a list of playlists in your playlist library."
 },
 {
 "name": "show_playlist",
 "description": "Get detailed information about a specific playlist. You can view your own
     playlists or others' playlists if they are public."
 },
 ...
]
```

ASSISTANT:
The `show_playlist_library` API seems to be the one we need. Let's check its specifications.

```python
print(apis.api_docs.show_api_doc(app_name='spotify', api_name='show_playlist_library'))
```

USER:
Output:

```
{
 "app_name": "spotify",
 "api_name": "show_playlist_library",
 "path": "/spotify/library/playlists",
 "method": "GET",
 "description": "Search or show a list of playlists in your playlist library.",
 "parameters": [
  {
   "name": "access_token",
   "type": "string",
   "required": true,
   "description": "Access token obtained from spotify app login.",
   "default": null,
   "constraints": []
  },
  {
   "name": "query",
   "type": "string",
   "required": false,
   "description": "The search query string.",
   "default": "",
   "constraints": []
  },
  {
   "name": "is_public",
   "type": "boolean",
   "required": false,
   "description": "Whether to show public playlists or private playlists.",
   "default": null,
   "constraints": []
  },
  {
   "name": "page_index",
```

AppWorld prompt template (continued)

```
    "type": "integer",
    "required": false,
    "description": "The index of the page to return.",
    "default": 0,
    "constraints": [
     "value >= 0"
    ]
   },
   {
    "name": "page_limit",
    "type": "integer",
    "required": false,
    "description": "The maximum number of results to return per page.",
    "default": 5,
    "constraints": [
     "value >= 1, <= 20"
    ]
   },
   {
    "name": "sort_by",
    "type": "string",
    "required": false,
    "description": "The attribute to sort the library playlists by prefixed with +/- to reflect
        ascending/descending. Valid attributes: created_at and title. ...",
    "default": null,
    "constraints": []
   }
  ],
  "response_schemas": {
   "success": [
    {
     "playlist_id": 1,
     "title": "string",
     "is_public": true,
     "rating": 0.0,
     "like_count": 1,
     "review_count": 1,
     "owner": {
      "name": "string",
      "email": "user@example.com"
     },
     "created_at": "2019-01-01T00:00:00",
     "song_ids": [
      1
     ]
    }
   ],
   "failure": {
    "message": "string"
   }
  }
 }
```

ASSISTANT:

AppWorld prompt template (continued)

I need to page through all the playlists to get the list of playlists and save it in `playlists`.

```python
page_index = 0
playlists = []
while page_index < 10:
  playlist_page = apis.spotify.show_playlist_library(access_token=spotify_access_token,
      page_index=page_index)
  if playlist_page:
    playlists.extend(playlist_page)
    page_index += 1
  else:
    break
num_playlists = len(playlists)
print(num_playlists)
```

USER:
Output:
```
23
```

ASSISTANT:
Now that the task is completed, I can call apis.supervisor.complete_task(). Since this task has an answer to be returned, I will pass that as an argument.

```python
apis.supervisor.complete_task(answer=num_playlists)
```

USER:
Output:
Marked the active task complete.

USER:
Key instructions:

A. General instructions:

- Act fully on your own. You must make all decisions yourself and never ask me or anyone else to confirm or clarify. Your role is to solve the task, not to bounce questions back, or provide me directions to follow.
- You have full access −− complete permission to operate across my connected accounts and services.
- Never invent or guess values. For example, if I ask you to play a song, do not assume the ID is 123. Instead, look it up properly through the right API.
- Never leave placeholders; don't output things like "your_username". Always fill in the real value by retrieving it via APIs (e.g., Supervisor app for credentials).
- When I omit details, choose any valid value. For example, if I ask you to buy something but don't specify which payment card to use, you may pick any one of my available cards.
- Avoid collateral damage. Only perform what I explicitly ask for. Example: if I ask you to buy something, do not delete emails, return the order, or perform unrelated account operations.

AppWorld prompt template (continued)

B. App−specific instructions:

- All my personal information (biographical details, credentials, addresses, cards) is stored in the Supervisor app, accessible via its APIs.
- Any reference to my friends, family or any other person or relation refers to the people in my phone's contacts list.
- Always obtain the current date or time, from Python function calls like `datetime.now()`, or from the phone app's get_current_date_and_time API, never from your internal clock.
- All requests are concerning a single, default (no) time zone.
- For temporal requests, use proper time boundaries, e.g., when asked about periods like "yesterday", use complete ranges: 00:00:00 to 23:59:59.
- References to "file system" mean the file system app, not the machine's OS. Do not use OS modules or functions.
- Paginated APIs: Always process all results, looping through the page_index. Don't stop at the first page.

C. Code−operation instructions

- Make sure to end code blocks with ``` followed by a newline(\n).
- Remember, you can use the variables in your code in subsequent code blocks.
- Remember that the email addresses, access tokens and variables (e.g. spotify_password) in the example above are not valid anymore.
- Always look at API specifications (using apis.api_docs.show_api_doc) before calling an API.
- Write small chunks of code and only one chunk of code in every step. Make sure everything is working correctly before making any irreversible changes.
- The Python environment supports the standard library. But system−level operations that may access or affect OS files, processes, etc., are not allowed and will raise an error if called.
- To interact with apps, only use the provided app APIs, and not the corresponding Python packages, e.g., do NOT use `spotipy` for Spotify.
- The provided API documentation has both the input arguments and the output JSON format. Use this information when making API calls and parsing their outputs.

D. Task−completion instructions:

You must call the `apis.supervisor.complete_task` API after completing the task.
- If an answer is needed, e.g., for "How many songs are in the Spotify queue?", call it with the appropriate answer argument value.
- If no answer is required, e.g., for "Start my Spotify music player.", omit the answer argument (or set it to None/null).
- The task is doable, but if you cannot find a way, you can call it with status="fail" to exit with failure.

When the answer is given:
- Keep answers minimal. Return only the entity, number, or direct value requested − not full sentences.
  E.g., for the song title of the current playing track, return just the title.
- Numbers must be numeric and not in words.
  E.g., for the number of songs in the queue, return "10", not "ten".

USER:
Using these APIs, now generate code to solve the actual task:

AppWorld prompt template (continued)

My name is: {{ main_user.first_name }} {{ main_user.last_name }}. My personal email is {{ main_user.email }} and phone number is {{ main_user.phone_number }}.
Task: {{ instruction }}