When LLM Reward Design Fails: Diagnostic-Driven Refinement for Sparse Structured RL

Anonymous authors
Paper under double-blind review

Abstract

For sparse, structured reinforcement-learning tasks with semantic reward-function interfaces, LLM-generated reward shaping is better framed as a debugging problem than as pure one-shot generation. We study PPO-trained agents, using MiniGrid as the core sparse-structured evaluation and MuJoCo reaching/locomotion as boundary stress tests. Our audit finds two dominant one-shot failure modes—reward flooding and semantic/API misunderstanding—plus a rarer, less reliably labeled weak-shaping case. We propose **diagnostic-driven iterative refinement**, where training diagnostics and a failure-mode taxonomy guide targeted reward-function revision. On sparse structured tasks with diagnosable reward failures, refinement improves DoorKey-8×8 from 2.3% success without shaping to 97.6%, and KeyCorridor from 31.2% one-shot to 86.7% with high seed-to-seed variance. Controls indicate that these gains are not attributable to generic retrying or extra training alone: metrics-only re-prompting produces large drops (DoorKey-8×8: 97.6% → 68.6%; KeyCorridor: 86.7% → 11.5%), while a static-vocabulary control recovers much of the gap (DoorKey-8×8: 87.6%; KeyCorridor: 70.7%), showing that the taxonomy prompt itself is a major mechanism and that dynamic trigger labels provide only partially isolated incremental evidence. Budget-matched and Best-of-3 comparisons help separate refinement from selection and training-time effects. Component-removal stress tests, sensitivity analyses, and an audit against author labels provide converging evidence for the debugging interpretation while revealing calibration limits. Continuous-control results expose the boundary: success-based diagnostics can misfire in dense-reward locomotion, and return-trend feedback removes one false-positive mechanism without producing robust locomotion gains. The low-call protocol is a protocol-cost contrast with population-based reward search, not a shared-benchmark performance comparison. In the four environments where we run a fully crossed variance design, variance point estimates are consistent with larger gains from reward correction when LLM reward-function variance dominates, but bootstrap intervals are wide and temper exact share claims. The method's scope is deliberately bounded to sparse structured tasks with reliable structured interfaces under PPO training; richer semantic fields such as `event_text` can help, hurt, or be neutral depending on alignment with the task structure.

1 Introduction

Reinforcement learning in environments with sparse rewards remains a fundamental challenge. An agent exploring randomly in a multi-step task—such as finding a key, unlocking a door, and reaching a goal—may never encounter the terminal reward signal within feasible training budgets. Reward shaping (Ng et al., 1999) addresses this by providing intermediate feedback, but designing effective shaping functions demands substantial domain expertise and iterative manual tuning.

Recent work has explored LLMs as reward function generators: Eureka performs evolutionary optimization over reward code with GPU-parallel simulation, while Text2Reward generates dense reward code from language and compact environment representations (Ma et al., 2024; Xie et al., 2024). These systems achieve impressive results, but they provide limited insight into *why* LLM-generated rewards fail—treating reward

generation primarily as a performance-driven search or refinement problem rather than analyzing the structure of failures.

We argue that LLM reward design for sparse, structured tasks is better understood as a *debugging* problem than a generation problem. One-shot LLM generation fails in recurring, identifiable ways; lightweight diagnostics can often detect and repair these failures cheaply in the studied sparse structured setting; and the approach breaks predictably when diagnostics are misaligned with the task's evaluation signal. This reframing leads to a practical method—diagnostic-driven iterative refinement—that improves over one-shot generation and non-diagnostic re-prompting using 1–4 LLM calls per reward-design run, repeated across seeds for evaluation, and CPU training, without population-based search. This is a protocol-level efficiency comparison, not a direct performance comparison with evolutionary reward-search systems on shared benchmarks.

Our **main contribution** is a diagnostic-driven framework for identifying and repairing systematic failure modes in LLM-generated reward shaping, supported by targeted experiments that test the framework's core assumptions. We provide three **supporting findings**:

1. **A failure taxonomy for structured reward interfaces**: prompt controls show that taxonomy-rich feedback is functionally important, but also that static taxonomy vocabulary explains a substantial part of the improvement, so automated trigger labels should not be read as the sole causal mechanism. The taxonomy itself is grounded in an audit where reward flooding and semantic/API misunderstanding are the two dominant failure patterns, while weak or negligible shaping appears as a rarer and less reliably labeled case. On the dominant-mode subset, an LLM auto-labeler agrees with author labels with 94% precision (32/34 predicted dominant labels; 71.4% overall accuracy across all categories).
2. **Variance-based evidence** for when refinement helps: LLM generation variance has the clearest dominant point estimate in representative sparse tasks (DoorKey-8×8: LLM std 47.5% vs. RL std 0.3%; crossed LLM share 96% with a wide $[66, 100]$% bootstrap interval), making diagnostic refinement most useful in this regime; residual and RL-training variance have larger point estimates in representative continuous-control decompositions, limiting the benefit of reward-function correction.
3. **A domain-boundary stress test**: success-based diagnostics produce false positives on dense-reward locomotion, systematically degrading the feedback loop. We provide root-cause analysis and evaluate a return-trend diagnostic adaptation, showing both where the framework breaks and how one specific failure mechanism can be removed.

The method is strongest for sparse, structured tasks with reliable semantic state fields and exposed subgoal structure, and all experiments use PPO-trained policies. Richer natural-language event descriptions are not uniformly beneficial: an interface ablation shows that removing `event_text` is neutral on DoorKey-8×8 and improves a KeyCorridor rerun. We explicitly characterize this scope: it is not yet a claim about raw-pixel reward design, arbitrary robotics tasks, or algorithm-agnostic reward-function validity, and dense-reward settings require different diagnostics (Section 11).

Roadmap and evidence status. Table 2 provides the headline 10-seed MiniGrid and reaching results; Section 8 tests the diagnostic story through lower-seed stress tests and prompt controls; Sections 9.4–9.7 report boundary, budget, variance, and selection analyses. The central positive claim is about sparse structured PPO tasks with diagnosable reward-function failures. MuJoCo locomotion, fixed-coefficient RND, model-sensitivity, and protocol-cost comparisons are included as context and boundary evidence rather than as broad superiority claims.

2 Background

2.1 Reward Shaping

Potential-based reward shaping (Ng et al., 1999) provides a principled framework for adding intermediate rewards without altering the optimal policy. In practice, most reward shaping uses heuristic bonuses tied

to task-specific subgoals (e.g., +0.2 for picking up a key). While effective, these heuristics require domain knowledge and manual tuning.

2.2 LLMs for Code Generation in RL

Eureka (Ma et al., 2024) demonstrated that LLMs can generate reward functions for continuous-control tasks, using evolutionary search over hundreds of candidates evaluated in GPU-parallel simulation (IsaacGym). Text2Reward (Xie et al., 2024) generates dense executable reward code from language task descriptions and compact environment representations. Our work takes a complementary, diagnostic-driven approach: rather than searching broadly, we analyze *why* generated rewards fail and use targeted feedback to repair specific failure modes with minimal iteration. We evaluate primarily on discrete MiniGrid tasks and include continuous MuJoCo tasks as stress tests for the diagnostic assumptions.

2.3 MiniGrid Environments

We use the MiniGrid framework (Chevalier-Boisvert et al., 2023), which provides a suite of grid-world environments with partial observability. The agent receives a $7 \times 7 \times 3$ symbolic observation encoding object type, color, and state for each visible cell. Actions are discrete: turn left, turn right, move forward, pick up, drop, toggle, and done.

2.4 MuJoCo Continuous Control

To probe the boundary beyond discrete grid worlds, we also evaluate on MuJoCo (Todorov et al., 2012) continuous-control tasks via Gymnasium (Towers et al., 2024). For reaching tasks: **Reacher-v4** (2-DOF planar arm, 11-dim observation, 2-dim action) and **FetchReach-v4** (7-DOF robotic arm, 13-dim observation, 4-dim action) (Plappert et al., 2018). For the reaching diagnostics, we treat success as an end-effector distance below 0.05.

We further evaluate on standard locomotion benchmarks: **HalfCheetah-v4** (2D runner, 17-dim observation, 6-dim action) and **Hopper-v4** (single-legged hopper, 11-dim observation, 3-dim action). Unlike reaching tasks with sparse binary success, these feature dense multi-component rewards (forward velocity + alive bonus − control cost), making them a boundary test for diagnostics designed around binary success signals.

2.5 Intrinsic Motivation

Random Network Distillation (RND) (Burda et al., 2019) is a widely-used intrinsic motivation method that provides exploration bonuses based on prediction error between a fixed random network and a trainable predictor. The intrinsic reward is high for novel states and decreases as the predictor learns. RND is domain-agnostic and requires no task-specific knowledge, making it a natural comparison baseline for our LLM-guided reward design approach.

3 Method

3.1 Overview

Our pipeline has three components (Figure 1):

1. **Reward Generation**: An LLM receives a natural-language description of the environment and produces a Python function `reward_fn(obs, action, reward, terminated, truncated, info, state)` that returns a shaped reward and updated state dictionary.

2. **RL Training**: A standard PPO (Schulman et al., 2017) agent trains using the shaped reward. The environment wrapper provides structured information (agent position, carrying status, event text) in the `info` dictionary.

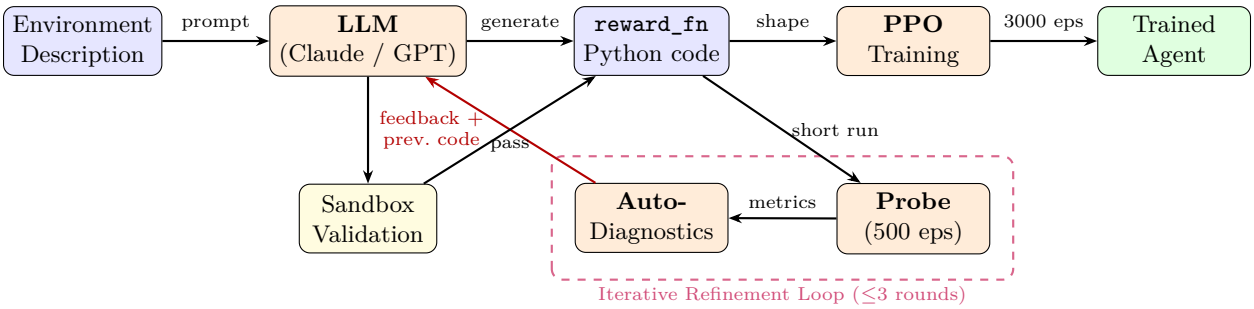

Figure 1: Overview of LLM-Guided Reward Design. The LLM generates a reward-shaping function from a natural-language environment description, validated in a sandbox before use. In iterative mode (dashed), short probe runs provide diagnostics for up to 3 refinement rounds.

3. **Iterative Refinement** (optional): After a short probe training run, automated diagnostics assess whether the reward function exhibits failure modes. The diagnosis and training metrics are fed back to the LLM, which produces a revised function.

3.2 Reward Function Interface

The generated reward function receives the full transition tuple plus a mutable `state` dictionary that persists across steps within an episode (reset between episodes). This enables one-time bonuses:

```
def reward_fn(obs, action, reward, terminated,
              truncated, info, state):
    shaped = reward
    if not state.get("key_picked_up"):
        if "picked up" in info["event_text"].lower():
            shaped += 0.2
            state["key_picked_up"] = True
    return shaped, state
```

The `info` dictionary exposes: `agent_pos`, `carrying`, `event_text` (natural-language event descriptions), `step_count`, and `max_steps`.

3.3 LLM Prompt Design

The generation prompt includes: (1) environment name and goal description, (2) available observation space and action space, (3) `info` dictionary fields with examples, (4) constraints (reward magnitudes 0.01–0.5, use one-time bonuses, avoid continuous rewards). For refinement, we additionally include: (5) the previous reward function source code, (6) probe training metrics (success rate, mean reward), and (7) diagnosed failure modes.

3.4 Automated Diagnostics

After each probe training run, we compute three diagnostics (exact thresholds in Table 16; supporting evidence in Section 8):

- **Reward hacking**: If mean reward > 0.5 and success rate < 0.2, the agent is accumulating shaped rewards without solving the task. Disabled for dense-reward locomotion tasks, which lack a binary success signal and use return-trend analysis instead (Section 9.4).
- **Shaping weakness**: If both success rate < 0.1 and mean reward < 0.1, the reward bonuses are too small to guide exploration.

- **Learning plateau**: If success rate has stagnated (improvement $< 5\%$ over the probe window) after $> 1{,}000$ probe episodes, the agent may be stuck in a local optimum. This gate is inactive under the default 500-episode MiniGrid probes and is retained for longer-probe variants rather than treated as a driver of the headline 500-episode results.

These diagnostics are formatted as natural-language warnings in the refinement prompt. The control experiments (Section 8) show that taxonomy-guided feedback is useful but imperfect: reward-hacking detection can generate false positives on some environments, shaping-weakness feedback is useful on moderately difficult tasks, and the plateau rule requires longer probes before it can be interpreted causally.

3.5 Iterative Refinement Algorithm

Algorithm 1 formalizes the iterative refinement procedure.

Algorithm 1: LLM-guided iterative reward design

$f_0 \leftarrow \text{LLM}(\mathcal{D})$; validate f_0 in the sandbox.
for $k = 0, \ldots, K-1$ **do**
  $\pi_k \leftarrow \text{PPO}(f_k, N_p)$ probe training
  $m_k \leftarrow \text{Evaluate}(\pi_k)$ success rate and reward
  if m_k.success_rate $\geq \tau$ **then break**.
  $d_k \leftarrow \text{Diagnose}(m_k)$ failure-mode feedback
  $f_{k+1} \leftarrow \text{LLM}(\mathcal{D}, f_k, m_k, d_k)$; validate f_{k+1}.
$\pi^* \leftarrow \text{PPO}(f_K, N_f)$ full training

3.6 Safety and Validation

Generated reward functions are executed in a sandboxed namespace with restricted `__builtins__` (only `math`, `abs`, `min`, `max`, `len`, etc.). Functions that fail validation (syntax errors, runtime errors on dummy inputs) trigger up to 3 retries with error feedback.

3.7 Extension to Continuous Control

For MuJoCo environments, we adapt the pipeline as follows: (1) the `info` dictionary exposes position-based fields (`distance_to_target`, `end_effector_pos`, `target_pos`) and event text ("reached target", "moved closer"), (2) the LLM prompt is adapted for continuous observations (flat vectors) and continuous actions, emphasizing distance-based progress bonuses over event-driven bonuses, (3) the PPO trainer uses an MLP policy with Gaussian actor, running observation normalization, and MuJoCo-appropriate hyperparameters (rollout length 2048, 10 PPO epochs, no entropy bonus). The reward function interface remains identical, ensuring the same iterative refinement protocol applies.

For locomotion environments (HalfCheetah, Hopper), the LLM designs an alternative reward function accessing velocity, position, and control cost from the `info` dictionary. The "no shaping" baseline uses the default Gymnasium reward. Since locomotion tasks have no binary success criterion, we evaluate using mean episode return over the final 100 episodes.

4 Interpretation

We connect our empirical approach to the reward shaping literature as an interpretive lens for the observed effectiveness and failure modes.

4.1 LLM Bonuses as Approximate Potential-Based Shaping

Ng et al. (Ng et al., 1999) proved that potential-based shaping $F(s, s') = \gamma\Phi(s') - \Phi(s)$ preserves the optimal policy. The one-time bonuses generated by our LLM (e.g., +0.2 when the key is first picked up) can

be interpreted as approximations to potential-based shaping over an augmented state space that includes event flags.

Let $\mathcal{S}^+ = \mathcal{S} \times \{0,1\}^k$ be the state space augmented with k binary milestone indicators (e.g., key picked up, door opened). Define a potential function:

$$\Phi(s, \mathbf{m}) = \sum_{i=1}^{k} b_i \cdot m_i \tag{1}$$

where b_i is the bonus for milestone i and $m_i \in \{0,1\}$ indicates whether milestone i has been achieved. The one-time bonus $+b_i$ when milestone i is first triggered corresponds to $\Phi(s', \mathbf{m}') - \Phi(s, \mathbf{m})$ at the transition where m_i changes from 0 to 1 (neglecting the discount factor). This connection is approximate because: (1) the LLM does not explicitly construct Φ, (2) the discount correction $\gamma\Phi(s') - \Phi(s)$ is not enforced, and (3) some LLM-generated functions include non-potential components (e.g., stagnation penalties). The near-potential structure therefore serves as intuition for why well-designed one-time bonuses can provide useful intermediate signal without obviously changing the task objective, not as a policy-invariance guarantee.

The failure mode of *reward flooding*—continuous per-step bonuses—is not potential-based over the original state representation. In finite-horizon or terminating tasks, a per-step bonus changes the relative return of trajectories with different lengths and can favor delaying termination; dynamic-potential variants illustrate why time-varying shaping terms require explicit structure rather than arbitrary repeated bonuses (Ng et al., 1999; Devlin & Kudenko, 2012). This can alter the optimal policy by favoring longer trajectories, exactly the behavior we observe when reward flooding masks the task signal.

4.2 Iterative Refinement Through a Variance-Reduction Lens

Consider the space of reward functions $\mathcal{F}$ that an LLM can generate. One-shot generation samples $f \sim P_{\text{LLM}}(\mathcal{F}|\mathcal{D})$ conditioned on the environment description $\mathcal{D}$. The resulting policy performance $J(f) = \mathbb{E}_{\pi_f}[R]$ is a random variable with variance:

$$\text{Var}[J] = \text{Var}_f[\mathbb{E}_\pi[R|f]] + \mathbb{E}_f[\text{Var}_\pi[R|f]] \tag{2}$$

The first term is LLM generation variance; the second is RL training variance. This equation describes the ideal marginal decomposition. Our anchored estimates in Section 9.6 use single-seed conditioning as a directional approximation, while the fully crossed 5×5 design in Table 14 realizes the random-effects partition directly on a representative subset.

Iterative refinement can be viewed as a conditional sampling procedure: rather than drawing independent samples from $P_{\text{LLM}}(\mathcal{F}|\mathcal{D})$, each refinement step conditions on the previous function and its training diagnostics, sampling from $P_{\text{LLM}}(\mathcal{F}|\mathcal{D}, f_k, d_k)$. This can be interpreted as reducing effective variability in the sampled reward functions by shifting generation toward functions that avoid diagnosed failure modes. The approach should be most useful when LLM generation variance dominates, which is consistent with the pattern we observe for discrete MiniGrid tasks, and should provide limited benefit when RL variance dominates, as in the continuous MuJoCo stress tests.

5 Experimental Setup

5.1 Environments

We evaluate on nine MiniGrid environments and four MuJoCo environments (Table 1):

Environment	Type	Key Challenge	Sub.
LavaGap-S5	Grid	Navigation + avoidance	2
DoorKey-5×5	Grid	Key → door → goal	3
DoorKey-8×8	Grid	Larger search space	3
UnlockPickup	Grid	Key → door → box	4
KeyCorridor-S3R1	Grid	Corridor navigation	3
BlockedUnlockPickup	Grid	Move ball, key, door, box	4
FourRooms	Grid	Multi-room navigation	1
MultiRoom-N6	Grid	Long-horizon (6 rooms)	1
ObstructedMaze-2Dlh	Grid	Hidden key, obstruction	4
Reacher-v4	Cont.	2-DOF arm → target	1
FetchReach-v4	Cont.	7-DOF gripper → goal	1
HalfCheetah-v4	Cont.	Forward locomotion	1
Hopper-v4	Cont.	Balance + locomotion	1

Table 1: Evaluation environments. Grid = MiniGrid discrete, Cont. = MuJoCo continuous. Sub. = sequential subgoals. The three new environments (FourRooms, MultiRoom-N6, ObstructedMaze) test longer planning horizons and harder exploration. Locomotion tasks feature dense default rewards and no binary success criterion.

5.2 Conditions

We compare five conditions:

1. **No Shaping**: Baseline PPO with only the sparse environment reward.
2. **LLM One-Shot**: A single LLM call generates the reward function.
3. **LLM Iterative**: Up to 3 refinement iterations with probe runs (500 episodes for MiniGrid, 200 for MuJoCo). Dense-reward locomotion uses the return-aware diagnostic adaptation in the main results; the original success-diagnostic variant is reported separately as a broken baseline.
4. **Hand-Crafted**: Human-designed reward functions with tuned bonus magnitudes.
5. **RND**: Random Network Distillation (Burda et al., 2019) intrinsic motivation as a domain-agnostic exploration baseline. The main table uses a fixed, untuned coefficient of 0.1 across environments; predictor architecture: 3-layer MLP (256–256–128). We separately run a post-hoc coefficient sweep on the two DoorKey tasks where reviewers identified this fixed coefficient as a likely confound.

5.3 Training Details

All conditions use PPO (Schulman et al., 2017). The main one-shot, no-shaping, hand-crafted, RND, and MiniGrid iterative conditions are evaluated across **10 random seeds** (42, 123, 456, 789, 1024, 2048, 3141, 4096, 5555, 7777). Exceptions are stated where they occur: return-aware locomotion refinement uses 6 completed seeds, and some secondary controls have fewer successful runs due to reward-function validation failures.

MiniGrid hyperparameters: learning rate 3×10^{-4}, $\gamma = 0.99$, GAE $\lambda = 0.95$, clip ratio 0.2, entropy coefficient 0.1, 2 PPO epochs, batch size 64, rollout length 512. Observations are normalized by dividing by 10 (MiniGrid symbolic values range 0–10). Training budget: 3,000 episodes.

MuJoCo hyperparameters: learning rate 3×10^{-4}, $\gamma = 0.99$, GAE $\lambda = 0.95$, clip ratio 0.2, entropy coefficient 0.0, 10 PPO epochs, batch size 64, rollout length 2,048. Observations use running mean/std normalization. MLP policy with 256-unit hidden layers and Gaussian actor (learnable log-std) with tanh squashing. Training budgets: 1,000 episodes for Reacher, HalfCheetah, and Hopper; 3,000 episodes for FetchReach (to account for its 7-DOF complexity).

The LLM used for reward generation is Claude Haiku (`claude-haiku-4-5-20251001`) with temperature 0.4.

Episode budget. The MiniGrid iterative condition uses up to $K \times N_p + N_f = 3 \times 500 + 3{,}000 = 4{,}500$ total training episodes (50% overhead). The MuJoCo iterative condition uses up to $3 \times 200 + N_f$ episodes, where N_f is the environment-specific full training budget. Probe runs are short and the additional compute cost is modest relative to total training.

6 Results

6.1 Main Results

Table 2 presents final performance for each condition, computed over the last 100 training episodes and averaged across random seeds.

Environment	Metric	No Shaping	LLM One-Shot	LLM Iterative	Hand-Crafted	RND
MiniGrid (discrete, 3,000 episodes)						
LavaGap-S5	SR	$64.4 \pm 23.7\%$	$64.3 \pm 32.6\%$	$\mathbf{88.8 \pm 16.7}\%$	$63.6 \pm 15.2\%$	$85.5 \pm 15.1\%$
DoorKey-5×5	SR	$99.8 \pm 0.4\%$	$85.3 \pm 31.4\%$	$99.7 \pm 0.5\%$	$\mathbf{99.9 \pm 0.3}\%$	$13.2 \pm 6.6\%$
DoorKey-8×8	SR	$2.3 \pm 1.1\%$	$58.3 \pm 47.3\%$	$97.6 \pm 3.0\%$	$\mathbf{100.0 \pm 0.0}\%$	$0.1 \pm 0.3\%$
UnlockPickup	SR	$0.1 \pm 0.3\%$	$15.8 \pm 7.8\%$	$\mathbf{24.3 \pm 10.3}\%$	$18.9 \pm 5.2\%$	$0.0 \pm 0.0\%$
KeyCorridor	SR	$11.7 \pm 5.3\%$	$31.2 \pm 37.1\%$	$\mathbf{86.7 \pm 28.5}\%$	$29.2 \pm 17.8\%$	$10.4 \pm 3.6\%$
BlockedUnlockPickup	SR	$0.0 \pm 0.0\%$	$0.7 \pm 0.8\%$	$0.6 \pm 1.0\%$	$\mathbf{0.8 \pm 0.8}\%$	$0.0 \pm 0.0\%$
FourRooms	SR	$\mathbf{27.2 \pm 4.2}\%$	$19.9 \pm 6.5\%$	$23.3 \pm 7.2\%$	$17.7 \pm 2.9\%$	$6.1 \pm 1.2\%$
MultiRoom-N6	SR	$0.0 \pm 0.0\%$	$\mathbf{0.5 \pm 1.0}\%$	$0.3 \pm 0.7\%$	$0.0 \pm 0.0\%$	$0.0 \pm 0.0\%$
ObstructedMaze	SR	$0.5 \pm 0.7\%$	$\mathbf{7.4 \pm 5.0}\%$	$6.6 \pm 2.3\%$	$6.5 \pm 3.1\%$	$0.0 \pm 0.0\%$
MuJoCo Reaching (continuous, 1,000 / 3,000 episodes)						
Reacher-v4	SR	$65.0 \pm 7.6\%$	$\mathbf{77.0 \pm 4.1}\%$	$73.7 \pm 6.5\%$	$75.4 \pm 6.6\%$	$67.5 \pm 8.6\%$
FetchReach-v4	SR	$6.5 \pm 2.4\%$	$7.5 \pm 2.6\%$	$\mathbf{12.8 \pm 8.0}\%$	$8.0 \pm 3.4\%$	$6.5 \pm 2.2\%$
MuJoCo Locomotion (continuous, 1,000 episodes; iterative = return-aware, $n = 6$)						
HalfCheetah-v4	Ret	1864 ± 749	2190 ± 405	2215 ± 485	2287 ± 497	1580 ± 641
Hopper-v4	Ret	$453 \pm 104^{\dagger}$	485 ± 101	472 ± 112	462 ± 163	549 ± 178

Table 2: Main results across 13 environments (last 100 episodes, mean ± sample std). SR = success rate; Ret = mean episode return. **Bold** indicates best performance per environment for MiniGrid and reaching tasks; no locomotion cell is bolded because those rows are boundary diagnostics with no robust corrected improvement. We report 12 primary comparisons in Appendix B; 9/12 are significant after per-family Holm–Bonferroni correction. The iterative column reports standard diagnostic refinement for MiniGrid and reaching tasks, and return-aware refinement for dense-reward locomotion. Return-aware locomotion iterative results use the first 6 completed canonical seeds; other main results use 10 seeds. FetchReach trained for 3,000 episodes; all other MuJoCo environments for 1,000. † The Hopper no-shaping value is the 1,000-episode main-budget result; the budget-matched 1,600-episode no-shaping result is 882 in Section 9.5. RND uses a fixed, untuned coefficient $c = 0.1$ in this table; Table 3 reports a coefficient sweep for the two DoorKey rows.

We do not interpret these results as showing that LLM rewards generally dominate hand-crafted rewards, intrinsic motivation, or continuous-control baselines. The central claim is narrower: in sparse structured PPO settings where one-shot LLM rewards fail for diagnosable reasons, taxonomy-guided diagnostic feedback improves over one-shot generation and non-diagnostic retrying. The hand-crafted, RND, and MuJoCo results are included to characterize context and boundaries, not to claim broad reward-design superiority.

Key findings:

- **One-shot LLM design is unreliable**: While it improves over no shaping on DoorKey-8×8 (2.3% → 58.3%), it shows high variance (sample std up to 47.3%) and performs comparably to no shaping on LavaGap (64.3% vs. 64.4%). On DoorKey-5×5, one-shot drops to 85.3% while all other conditions reach ∼100%.
- **Iterative refinement provides its clearest gains on sparse structured tasks**: KeyCorridor improves from 31.2% one-shot to 86.7% iterative, and DoorKey-8×8 reaches 97.6% (vs. 58.3% one-

shot). Hand-crafted baselines provide useful context, but the primary comparison is against one-shot generation and non-diagnostic controls.

- **Fixed-coefficient RND is mixed and coefficient-sensitive**: With the untuned coefficient used in Table 2, RND intrinsic motivation *hurts* performance on DoorKey-5×5 (13.2% vs. 99.8% no shaping) and DoorKey-8×8 (0.1% vs. 2.3%). A reviewer-requested sweep shows this was partly a tuning artifact on DoorKey-5×5: $c = 0.01$ reaches 100.0%, while $c = 0.1$ and $c = 0.5$ remain much worse. On DoorKey-8×8, however, all swept RND coefficients remain near zero. We therefore treat RND as contextual intrinsic-motivation evidence, not as a tuned baseline claim.

- **The strongest positive results occur where one-shot generation is unreliable but subgoals are semantically exposed**: DoorKey-8×8 reaches 97.6%, LavaGap reaches 88.8% under this PPO/protocol implementation, and KeyCorridor reaches 86.7% iterative success. The KeyCorridor mean is high but unstable (sample std 28.5%): the median seed reaches 100%, but one seed reaches only 19% and another 50%. Later stress tests show that diagnostic thresholds and `event_text` signals can substantially change outcomes, so the KeyCorridor result is best read as high-variance evidence that taxonomy-guided refinement can improve over one-shot generation, not as isolated proof that dynamic diagnostic labels are the sole repair mechanism. On simpler environments (DoorKey-5×5) all methods converge. On BlockedUnlockPickup, all conditions achieve $\leq 0.8\%$, indicating a task beyond the reach of reward shaping alone. UnlockPickup improves modestly but remains challenging (24.3% iterative success).

- **Richer semantic interfaces are not always better**: The headline KeyCorridor result (86.7%, $n = 10$) is essentially matched by a 5-seed rerun that removes `event_text` from both the prompt and runtime interface (86.8%), while the paired with-events rerun reaches only 55.2%. The five seed IDs are a subset of the headline canonical seeds, but the ablation uses independent LLM generations, so the near-equality should be read as a scope warning rather than a paired replication. The method needs structured semantic state, but not every semantic field helps.

- **Harder environments reveal limits**: On FourRooms, no shaping (27.2%) outperforms all reward-shaped conditions, suggesting the LLM's intermediate bonuses interfere with the agent's exploration in open multi-room layouts. On ObstructedMaze, LLM one-shot (7.4%) and iterative (6.6%) substantially improve over no shaping (0.5%), demonstrating that LLM-designed rewards help on tasks with hidden subgoal structure. MultiRoom-N6 remains near 0% across all conditions, indicating a horizon beyond current reward shaping capabilities.

Environment	**RND** $c = 0.01$	**RND** $c = 0.1$	**RND** $c = 0.5$
DoorKey-5×5	$\mathbf{100.0 \pm 0.0}\%$	$14.8 \pm 5.6\%$	$4.5 \pm 4.5\%$
DoorKey-8×8	$\mathbf{0.9 \pm 0.9}\%$	$0.0 \pm 0.0\%$	$0.4 \pm 0.8\%$

Table 3: Reviewer-requested RND coefficient sweep on the two DoorKey tasks (10 seeds, 3,000 episodes, final-100 success rate). The fixed $c = 0.1$ baseline in Table 2 was badly mistuned on DoorKey-5×5, where $c = 0.01$ solves the task, but tuning over this small grid does not rescue RND on DoorKey-8×8.

6.2 Learning Curves

Figure 2 shows learning curves across the core sparse tasks. The iterative condition learns faster and reaches higher final performance than one-shot on the environments where one-shot reward generation is most unreliable. On DoorKey-8×8, the no-shaping baseline never exceeds 2.3% success, while iterative refinement reaches 97.6%. On LavaGap, iterative refinement achieves 88.8%—more than 25% above the hand-crafted baseline (63.6%).

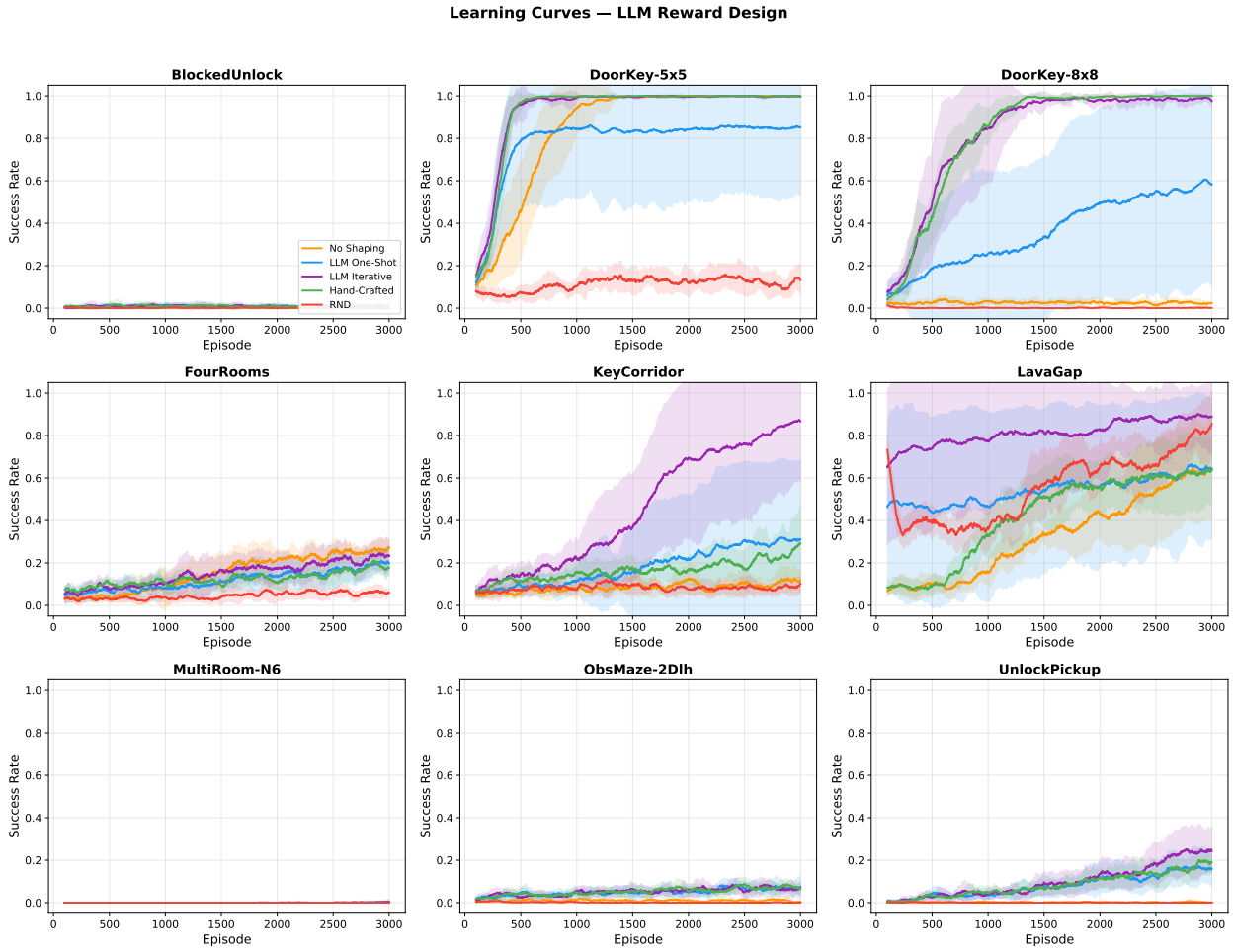

Figure 2: Learning curves (success rate vs. episode, smoothed over 100 episodes). Shaded regions show ±1 sample std across seeds.

6.3 Post-Refinement Training Episodes

Figure 3 compares the number of full-training episodes required to reach 80% success rate after the reward function has been selected or refined. Probe episodes used during reward design are excluded from this figure; total environment-interaction controls are reported separately in Section 9.5. Under this post-refinement training view, the iterative condition reaches the threshold faster than the hand-crafted baselines on DoorKey-5×5 and DoorKey-8×8.

7 Failure Taxonomy

The main results show that one-shot LLM reward generation is useful but unreliable. To make the failure structure actionable, we classify generated reward functions into two dominant failure modes and one rarer weak-shaping case (Table 4).

Reward flooding is one of the most damaging observed failure modes. In DoorKey-8×8, one generated function assigned +0.02 per forward step, leading to high accumulated reward but only ∼5% success rate—the agent learned to walk in circles collecting step rewards rather than solving the task. The iterative diagnostic detects this pattern (high reward, low success) and explicitly warns the LLM in the refinement prompt.

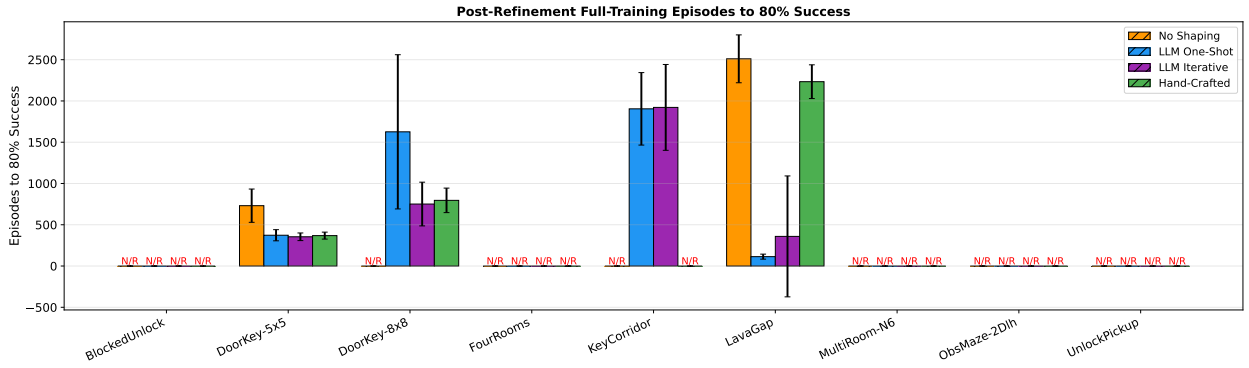

Figure 3: Full-training episodes to reach 80% success after reward selection/refinement. Probe episodes are excluded here; total-budget controls are reported in Section 9.5. Hatched bars (N/R) indicate the threshold was never reached within 3,000 full-training episodes.

Failure Mode	**Description**
Reward Flooding	Continuous per-step rewards (e.g., +0.02 for every forward action) dominate the shaped reward, masking task-completion signal.
Semantic/API Misunderstanding	LLM misuses environment semantics or APIs (e.g., maps MiniGrid action indices incorrectly, or hallucinates invalid `info` fields).
Weak / Negligible Shaping	Bonus magnitudes too small relative to the sparse reward, providing insufficient gradient signal; rare in our audit.

Table 4: Failure modes of LLM one-shot reward generation. Reward flooding and semantic/API misunderstanding are the dominant modes in our audit; weak or negligible shaping is observed more rarely and is less reliably labeled.

This taxonomy is not only descriptive: it defines the feedback vocabulary used in iterative refinement. The next section tests whether those diagnostic signals are useful enough to serve as the method's core mechanism.

8 Diagnostics Evidence and Controls

A strong objection to our approach is that the diagnostics *are* the method, yet aggregate performance alone does not establish that the diagnostic signals are reliable. We therefore treat the diagnostics as empirically useful but imperfect heuristics and evaluate them through targeted controls: a component-removal stress test (which detector families are risky?), prompt controls against metrics-only and static-vocabulary baselines (does taxonomy-rich feedback add value, and how much is due to the static taxonomy vocabulary?), a taxonomy-labeling audit against author labels (are dominant failure-mode labels recoverable?), and sensitivity checks for threshold and probe design. Table 5 summarizes what each control supports and what it does not establish. Appendix C lists all trigger values used in the experiments.

Unless stated otherwise, control experiments in this section are independent lower-seed reruns, not exact reproductions of the 10-seed means in Table 2; captions report the seed counts and protocol differences.

Control	Question answered	Main result	Limitation
Metrics-only re-prompting	Does taxonomy-rich feedback add value beyond retrying with raw metrics?	Removing diagnostic tags and static failure-mode vocabulary causes large hard-task drops (DoorKey-8×8: 97.6% → 68.6%; KeyCorridor: 86.7%→11.5%).	Prompt-level control; intentionally removes both static and dynamic taxonomy information.
Static-vocabulary control	Are dynamic per-iteration labels isolated as the mechanism?	Keeping static failure-mode vocabulary but removing dynamic labels recovers much of the gap (DoorKey-8×8: 87.6%; KeyCorridor: 70.7%).	Full diagnostics still trend higher, but full-vs-static gaps are not statistically resolved at $n = 10$.
Component-removal stress test	Which detector families create sensitivity?	Shaping-weakness removal hurts DoorKey; reward-hacking removal exposes false positives on KeyCorridor.	Low-seed independent reruns; the plateau gate is inactive under 500-episode probes.
Threshold sensitivity	Are fixed trigger values brittle?	Reward-hacking and shaping thresholds reveal calibration sensitivity.	No universal threshold; plateau rows are inactive under default probe length.
Probe-length sensitivity	How much probe budget is needed?	Diagnostic quality depends on the observation window; longer probes reduce failures on hard tasks.	Weakens any claim that very short probes always suffice.
Taxonomy audit	Do dominant failure labels match author labels?	Reward flooding and semantic/API misunderstanding are recoverable against author labels for dominant-mode predictions (94% precision).	Author-label audit, not a human inter-annotator agreement study; does not validate online trigger heuristics or rare classes.
Best-of-3 selection	Is improvement just selection among multiple samples?	Selection helps strongly; iterative has a higher KeyCorridor mean but the $n = 10$ gap is not statistically resolved.	MuJoCo Best-of-3 has incomplete seeds due to validation failures.
Episode-budget control	Is improvement just extra training time?	MiniGrid gains persist beyond extra training time; Hopper exposes a training-duration confound.	Does not remove all optimizer/reward interaction effects.

Table 5: Evidence matrix for the diagnostic mechanism. Environment, seed, and budget details are reported in the corresponding subsections; this table summarizes the reviewer-facing interpretation. The controls support the debugging interpretation through converging evidence, while also identifying calibration limits and boundary cases.

8.1 Component-Removal Stress Test

We remove each diagnostic component one at a time—reward-hacking detection (RH), shaping-weakness detection (SW), and learning-plateau detection (LP)—and compare against the full system on four representative environments. These are independent 3-seed reruns and should be read as a stress test for detector sensitivity rather than as an isolated causal decomposition of the 10-seed headline results.

Environment	Full	−RH	−SW	−LP[†]
DoorKey-5×5	1.00	1.00	0.99	1.00
DoorKey-8×8	0.93	0.95	**0.64**	0.97
KeyCorridor	0.71	1.00	1.00	0.18
UnlockPickup	0.21	0.25	0.23	0.23

Table 6: Component-removal stress test: mean success rate from independent 3-seed reruns with one detector removed at a time. **Bold** marks substantial degradation (>15 points) outside the LP column. [†]Because the plateau gate requires > 1,000 probe episodes, the −LP column reports an observed rerun drop rather than causal evidence about plateau warnings under the default 500-episode probes.

The stress test reveals an asymmetric sensitivity pattern. **Shaping-weakness detection matters on moderately difficult tasks**: removing it drops DoorKey-8×8 from 93% to 64% ($\Delta = -29$ points), where the reward function must provide sufficient exploration guidance. The −LP KeyCorridor drop (71% → 18%) is not interpreted as a clean plateau-detector effect: local-log inspection shows that the default 500-episode

probes do not satisfy the > 1,000-episode plateau eligibility gate, so plateau warnings do not fire in these runs. We therefore treat this row as evidence of lower-seed rerun sensitivity, not as proof that the plateau detector explains the KeyCorridor gains.

An unexpected finding is that **removing the reward-hacking detector improves performance** on KeyCorridor (71% → 100%, $\Delta = +28$) and does not hurt DoorKey-8×8 (93% → 95%). This suggests the reward-hacking detector generates false positives on these environments, triggering unnecessary refinements that degrade the reward function. We retain the detector in the headline pipeline because it provides a modest UnlockPickup safeguard and catches pathological flooding cases, but practitioners should disable it or raise its cutoff when probe traces show useful progress despite low early success. These results suggest that shaping-weakness and reward-hacking diagnostics are useful but require calibration, while plateau detection should only be analyzed in protocols where its eligibility gate can actually fire.

8.2 Taxonomy Prompt Controls

A natural question is whether the gains come from taxonomy-rich diagnostic feedback or merely from giving the LLM another attempt. We first compare the full diagnostic pipeline against a *metrics-only baseline* that provides the LLM with identical numerical metrics (success rate, mean reward) but strips all dynamic diagnostic tags, failure-mode names, and static "common failure modes to avoid" guidance. The LLM receives only: "the results are suboptimal; write an improved reward function." This is a strong prompt-level ablation of taxonomy information.

Environment	Full Diagnostic	Metrics-Only
DoorKey-5×5	99.7%	$92.8 \pm 21.4\%$
DoorKey-8×8	97.6%	$68.6 \pm 38.9\%$
UnlockPickup	24.3%	$6.8 \pm 7.7\%$
KeyCorridor	86.7%	$11.5 \pm 10.1\%$

Table 7: Full diagnostic-guided refinement vs. metrics-only re-prompting (mean ± sample std; metrics-only uses 9–10 successful seeds, with UnlockPickup at $n = 9$). The full diagnostic column reuses the 10-seed Table 2 means. The taxonomy-rich feedback condition provides 29–75 percentage point improvements on the hardest tasks, but this prompt-level ablation removes both dynamic diagnostic labels and static failure-mode vocabulary.

Removing taxonomy-rich feedback causes large performance drops on the hardest tasks: DoorKey-8×8 drops from 97.6% to 68.6% ($\Delta = 29.0$ points; Welch $p = 0.043$; bootstrap 95% CI $[7.9, 53.5]$ points), KeyCorridor from 86.7% to 11.5% ($\Delta = 75.2$ points; $p = 6.7\times10^{-6}$; CI $[56.0, 90.7]$), and UnlockPickup from 24.3% to 6.8% ($\Delta = 17.5$ points; $p = 6.1\times10^{-4}$; CI $[9.4, 24.7]$). DoorKey-5×5 is saturated and the smaller gap is not statistically resolved ($p = 0.33$). The metrics-only baseline is comparable to one-shot generation on the hard sparse tasks, providing evidence that **taxonomy-rich feedback, not mere re-prompting, is important to iterative refinement's effectiveness**.

To separate static taxonomy vocabulary from dynamic trigger labels, we add a stricter reviewer-requested control on DoorKey-8×8 and KeyCorridor. This condition keeps the static `COMMON FAILURE MODES TO AVOID` block and the numerical probe metrics, but removes per-iteration trigger strings such as `REWARD HACKING DETECTED`, `SHAPING TOO WEAK`, and `PLATEAU DETECTED`. Iteration-0 reward functions are shared with the full diagnostic condition when available, so the manipulation begins at refinement.

The static-vocabulary control changes the mechanism interpretation. On DoorKey-8×8, full diagnostics exceed static-vocabulary prompting by 10.0 points, but the interval includes small or negative gaps (bootstrap CI $[-1.8, 30.3]$; Welch $p = 0.33$; $d = 0.46$). Static vocabulary exceeds metrics-only by 19.0 points, but that comparison is also not resolved at $n = 10$ (CI $[-10.6, 47.9]$; $p = 0.24$; $d = 0.54$). On KeyCorridor, static vocabulary captures most of the improvement over metrics-only (70.7% vs. 11.5%; $\Delta = 59.2$ points; CI $[32.1, 83.5]$; $p = 0.0017$; $d = 1.90$), while full diagnostics trend higher than static vocabulary by 16.0 points but remain statistically unresolved (CI $[-13.6, 46.7]$; $p = 0.34$; $d = 0.44$). We therefore interpret the prompt controls as evidence that the failure taxonomy is a major active ingredient; the online diagnostic labels may

Environment	Full Diagnostic	Static Vocab Only	Metrics-Only
DoorKey-8×8	$97.6 \pm 3.0\%$	$87.6 \pm 30.6\%$	$68.6 \pm 38.9\%$
KeyCorridor	$86.7 \pm 28.5\%$	$70.7 \pm 42.9\%$	$11.5 \pm 10.1\%$

Table 8: Static-vocabulary prompt control (10 seeds). Static-vocab-only keeps the failure-mode vocabulary and raw metrics while removing dynamic per-iteration diagnostic labels. It recovers much of the metrics-only gap, especially on KeyCorridor, so the mechanism should be framed as taxonomy-guided prompting plus diagnostics rather than dynamic trigger labels alone.

add value, but these experiments do not isolate them as the sole driver. A seed-level inspection illustrates why we avoid dismissing dynamic labels entirely: on DoorKey-8×8 seed 789, static-vocab-only reaches only 1% final-100 success, whereas the full diagnostic run reaches 100%. This single catastrophic static-vocab seed explains much of the DoorKey static-vocab variance and is suggestive, not conclusive, evidence that dynamic labels can rescue hard cases.

8.3 Taxonomy Labeling Audit

We evaluate whether failure-mode labels agree with author labels. From the iterative refinement logs across all environments, we sample 49 reward functions that underwent refinement and manually label each with its primary failure mode: reward flooding, semantic/API misunderstanding, weak/negligible shaping, or none/mixed. We then compare these labels against an independent LLM-based auto-labeler (Claude Sonnet 4.6, temperature 0). This audit supports the taxonomy labeling used in our analysis for the two dominant modes; it does not validate the online trigger heuristics directly, which are separately evaluated through controls and sensitivity checks.

Failure Mode	Precision	Recall	Support
Reward flooding	0.94	0.65	23
Semantic/API misunderstanding	0.94	1.00	17
Other / rare / mixed	—	—	9
Overall accuracy	71.4% (35/49)		

Table 9: Taxonomy labeling audit: LLM auto-labeler vs. author labels on 49 reward functions. The two dominant failure modes (flooding, semantic/API misunderstanding) achieve 94% precision on predicted dominant labels (32/34). Overall accuracy across all categories is 71.4% (35/49). Rare or mixed cases are grouped because weak/negligible shaping has support = 1 and should not be interpreted as a stable class-level statistic.

The two dominant failure modes—reward flooding and semantic/API misunderstanding, which together account for 40/49 author-labeled samples—are labeled with 94% precision when the auto-labeler predicts one of those dominant modes. In this 49-function audit, the auto-labeler does not miss any author-labeled semantic/API misunderstanding cases. Reward flooding recall is lower (0.65); the 8 missed flooding cases fall into rare or mixed categories, indicating a conservative boundary between continuous over-rewarding, too-small shaping, and ambiguous cases rather than confusion with semantic API errors. We do not interpret rare or mixed classes as stable categories because weak/negligible shaping has only one author-labeled example and none/mixed cases are inherently ambiguous. A domain split emerges in the labeled data: semantic/API misunderstanding dominates MiniGrid failures, while reward flooding dominates continuous-control failures—consistent with the structural differences between discrete event-driven and dense continuous reward landscapes. Because the reference labels are author labels rather than independently adjudicated human labels, this audit should be read as evidence that the dominant taxonomy categories are recoverable by an independent LLM labeler, not as a measurement of human inter-annotator reliability.

8.4 Threshold Sensitivity

We vary each diagnostic threshold independently while keeping all other components fixed, evaluating on three representative environments (DoorKey-8×8, KeyCorridor, LavaGap) with up to 5 successful seeds per configuration. Table 10 reports mean success rates.

Sweep	Env	Low	Default	High
RH cutoff (0.3 / 0.5 / 1.0)				
	DoorKey-8×8	80.0 ± 44.7	94.2 ± 6.8	98.6 ± 2.6
	KeyCorridor	74.0 ± 39.7	53.2 ± 44.8	48.2 ± 47.5
	LavaGap	87.6 ± 16.1	81.8 ± 21.0	86.8 ± 14.9
Plateau cutoff (0.02 / 0.05 / 0.1)				
	DoorKey-8×8	80.4 ± 43.3	94.2 ± 6.8	66.3 ± 56.6
	KeyCorridor	$\mathbf{100.0 \pm 0.0}$	53.2 ± 44.8	68.8 ± 45.5
	LavaGap	89.8 ± 18.0	81.8 ± 21.0	84.8 ± 17.0
SW cutoff (0.05 / 0.1 / 0.3)				
	DoorKey-8×8	98.2 ± 2.4	94.2 ± 6.8	98.6 ± 2.1
	KeyCorridor	71.0 ± 39.3	53.2 ± 44.8	66.0 ± 46.7
	LavaGap	78.6 ± 16.3	81.8 ± 21.0	82.0 ± 27.5

Table 10: Threshold sensitivity: mean success rate (%) ± sample std from independent sensitivity runs (up to 5 successful seeds per cell) when varying one diagnostic threshold while holding others at default. Default cells are sweep reruns rather than Table 2 means. The plateau cutoff is inactive under 500-episode probes because the plateau gate requires > 1,000 probe episodes, so those rows should be read as rerun variability rather than a causal threshold effect. **Bold** marks the best cell per sweep.

Two actionable patterns emerge from Table 10. **Reward-hacking threshold**: increasing the cutoff (less aggressive detection) improves DoorKey (80% → 99%) but slightly degrades KeyCorridor (74% → 48%); together with the component-removal stress test, this indicates that reward-hacking detection is highly environment-sensitive rather than monotonic in its threshold. **Shaping-weakness threshold**: shows non-monotonic, environment-dependent effects; no single direction is universally optimal. The plateau rows are included for transparency, but because the > 1,000-episode gate is inactive in these 500-episode probes, their variation should be interpreted as lower-seed rerun and LLM stochasticity rather than evidence about the plateau cutoff itself.

These low-seed descriptive sweeps show that the method is **sensitive but diagnostically interpretable**: threshold choices materially affect performance, especially on KeyCorridor, and the reward-hacking detector in particular needs task-specific calibration. Bootstrap CIs on threshold differences are wide: for example, raising the reward-hacking cutoff from 0.5 to 1.0 changes KeyCorridor by -5.0 points with CI $[-55.2, 47.4]$, while lowering it to 0.3 changes KeyCorridor by $+20.8$ points with CI $[-26.0, 64.4]$. The only threshold-difference CI that excludes zero is KeyCorridor's plateau-low row ($+46.8$ points, CI $[11.2, 82.4]$), but the plateau gate is inactive under these probes, so we treat this as rerun sensitivity rather than a causal plateau-threshold result. We therefore use the sweeps as calibration evidence rather than as definitive threshold-selection tests.

8.5 Probe-Length Sensitivity

The probe run is the diagnostic observation window: too short, and the diagnostics may lack signal; too long, and compute is wasted. We evaluate probe lengths of 300, 500, and 800 episodes on three representative environments (3 seeds each).

Environment	300 eps	500 eps	800 eps
DoorKey-8×8	0.67 ± 0.58	1.00 ± 0.01	0.98 ± 0.03
KeyCorridor	0.49 ± 0.43	0.40 ± 0.52	$\mathbf{1.00 \pm 0.00}$
LavaGap	0.72 ± 0.23	0.93 ± 0.06	0.90 ± 0.10
Overall	0.62 ± 0.39	0.78 ± 0.39	$\mathbf{0.96 \pm 0.07}$

Table 11: Probe-length sensitivity: mean success rate ± sample std from independent 3-seed reruns. Longer probes provide more reliable diagnostic observations, especially on hard environments.

Contrary to the intuition that shorter probes suffice for quick diagnosis, **longer probes improve the aggregate mean and substantially stabilize KeyCorridor**, though 800 episodes is not uniformly best on every environment. The overall mean increases monotonically from 62% (300 eps) to 96% (800 eps), with variance dropping from ±39% to ±7%. The effect is most pronounced on KeyCorridor, where probe=300 achieves only 49% while probe=800 reaches 100% across all seeds—the longer observation window gives the LLM less noisy refinement metrics and reduces premature or misleading updates.

On DoorKey-8×8, the 300-episode probe produces a catastrophic failure on one seed (SR=0%) while the other two seeds succeed, suggesting that 300 episodes is below the threshold needed for reliable reward-function evaluation on this task. The default of 500 episodes provides a reasonable balance, but practitioners working on hard tasks may benefit from longer probes.

These results reinforce the paper's central thesis: the diagnostic system's effectiveness depends on the quality of its observations, not just the quality of its heuristics.

8.6 Interface Ablation

The method depends on a curated reward-function interface (Section 11). To test how much the specific `event_text` field contributes, we strip it from both the LLM prompt and the runtime `info` dictionary, forcing the LLM to rely on other semantic fields (`carrying`, `agent_pos`, `step_count`). We compare the standard pipeline ("with events") against this degraded interface ("without events") on DoorKey-8×8 and KeyCorridor (5 seeds each, iterative refinement with 3 iterations). The two ablation conditions use the same five canonical seed IDs and independent LLM generations; those seed IDs are a subset of the 10-seed headline run.

Env	With evt.	No evt.	No − With
DoorKey-8×8	$96.6 \pm 7.1\%$	$94.4 \pm 6.9\%$	−2.2
KeyCorridor	$55.2 \pm 41.0\%$	$86.8 \pm 29.5\%$	+31.6

Table 12: Interface ablation: success rate (%) ± sample std (5 seeds) when `event_text` is removed from the LLM prompt and runtime info dict. On KeyCorridor, removing `event_text` *improves* performance.

The results are nuanced. On DoorKey-8×8, removing `event_text` costs only 2.2 percentage points—within noise—because the LLM can infer subgoal completion from `carrying` transitions (e.g., `"nothing"` → `"yellow key"` implies key pickup). On KeyCorridor, removing `event_text` *improves* performance by 31.6 points (55.2% → 86.8%), essentially matching the 10-seed headline mean of 86.7%. Because the ablation uses independent LLM generations, this near-equality is not a paired replication of the headline result; it is nevertheless a strong warning against framing the KeyCorridor result as specifically event-text-driven. Inspection of the generated reward functions suggests that `event_text` strings on KeyCorridor include misleading signals (e.g., partial-progress events for irrelevant objects) that cause the LLM to write over-specific reward functions; without these signals, the LLM falls back on more robust position- and carrying-based heuristics.

This finding refines the paper's scope claim: the method's effectiveness depends on having *some* structured semantic interface (`carrying`, `agent_pos`), but does not critically depend on any single field. Rich event descriptions can help, hurt, or be neutral depending on how well they align with the task's actual subgoal structure.

9 Analysis

9.1 Refinement Trajectory

Figure 4 shows how probe success rate evolves across refinement iterations. Key observations:

- **DoorKey-5×5**: Often solved within 1–2 iterations; all methods converge to ∼100% given 3,000 episodes, making this primarily a sample-efficiency benchmark.
- **DoorKey-8×8**: Clear improvement trajectory across iterations. Full training with the refined reward function reaches 97.6%, up from 58.3% with one-shot. Hand-crafted achieves 100%.
- **KeyCorridor**: The largest mean improvement—iterative refinement reaches 86.7% vs. 31.2% one-shot and 29.2% hand-crafted—but with high seed-to-seed variance. In light of the static-vocabulary and `event_text` controls, this result is best interpreted as evidence that taxonomy-guided refinement can improve over one-shot generation in a high-variance sparse structured setting, not as isolated proof that dynamic diagnostic labels are the sole repair mechanism.
- **LavaGap**: Iterative refinement achieves 88.8%, above the one-shot (64.3%) and hand-crafted (63.6%) baselines under this PPO/protocol implementation.

9.2 Continuous Control Results

To probe how the same reward-design interface behaves beyond discrete grid worlds, we evaluate on four MuJoCo environments (Figure 5). The reward function interface remains identical—the LLM receives a natural-language environment description and generates a Python function—but the results are best read as boundary evidence rather than a broad claim about continuous control.

Reaching tasks. On Reacher-v4, LLM one-shot achieves the highest success rate (77.0 ± 4.1%), outperforming no shaping (65.0%, Welch's $p < 0.001$, $d = 1.97$; significant with per-family correction within MuJoCo reaching). On FetchReach-v4 (3,000 episodes), iterative refinement achieves the highest success rate (12.8 ± 8.0%), roughly 2× no shaping (6.5%), though all conditions remain far from solving this challenging sparse-reward task.

Locomotion tasks. On HalfCheetah-v4 and Hopper-v4, the LLM designs alternative reward functions accessing velocity, position, and control cost. Unlike reaching tasks, these environments already have dense default rewards, making them a stress test for diagnostics designed around sparse binary success. In Table 2, the locomotion iterative column reports the return-aware diagnostic adaptation from Section 9.4, while the original success-diagnostic protocol is retained in Table 13 as the broken iterative baseline. The main finding is negative: iterative refinement does not robustly outperform one-shot on locomotion tasks. On HalfCheetah, return-aware iterative refinement (2,215 ± 485) is statistically indistinguishable from LLM one-shot (2,190 ± 405, corrected $p = 0.9795$). On Hopper, no condition significantly outperforms the baseline, and the budget-matched no-shaping control in Section 9.5 is much stronger than return-aware refinement. These results suggest that the feedback-driven diagnostic approach is most effective for sparse-reward environments where the LLM can identify specific failure modes.

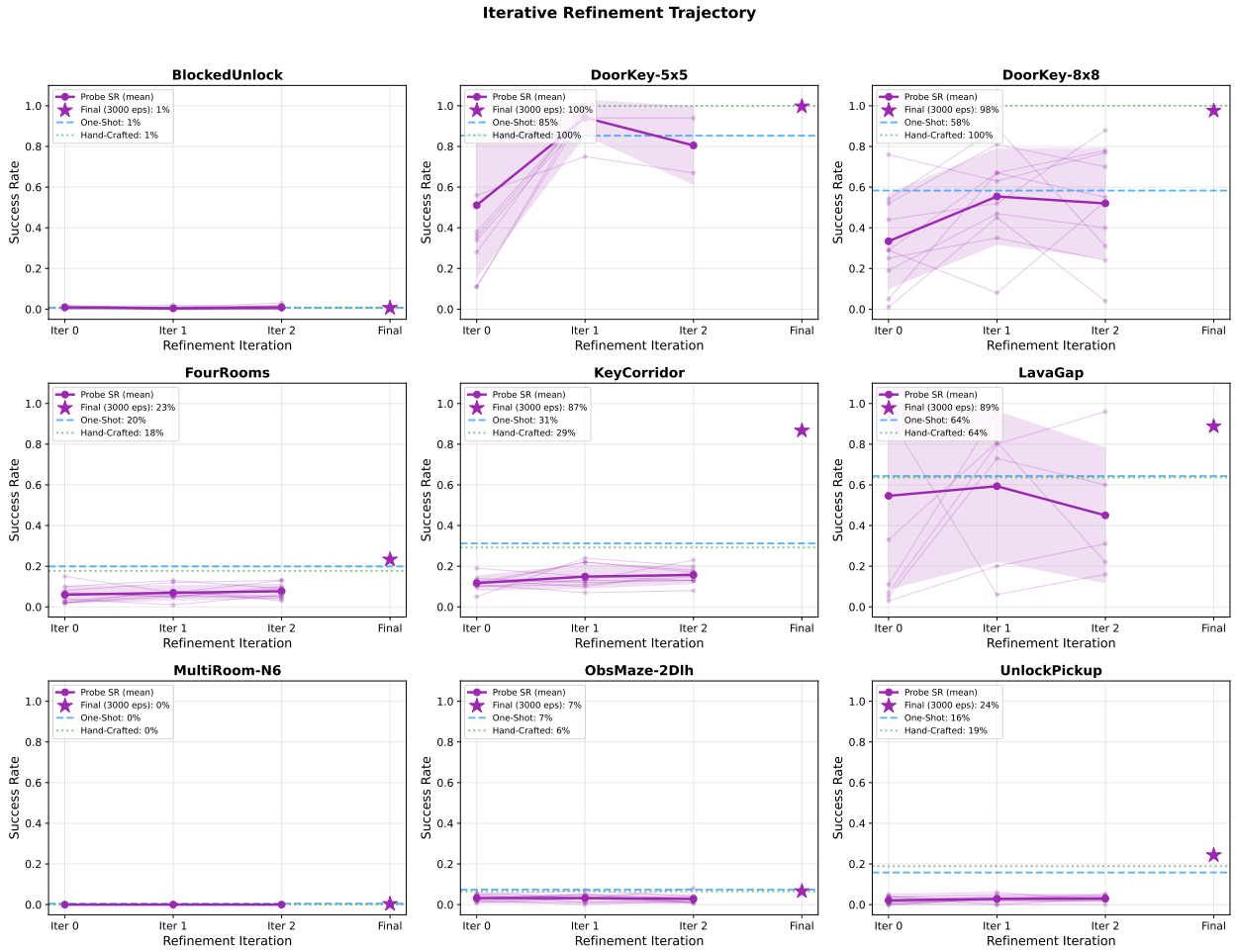

Figure 4: Refinement trajectory: probe success rate at each refinement round. Individual seed traces shown in light purple; mean in bold. Stars indicate final full-training results. Dashed lines show one-shot and hand-crafted baselines.

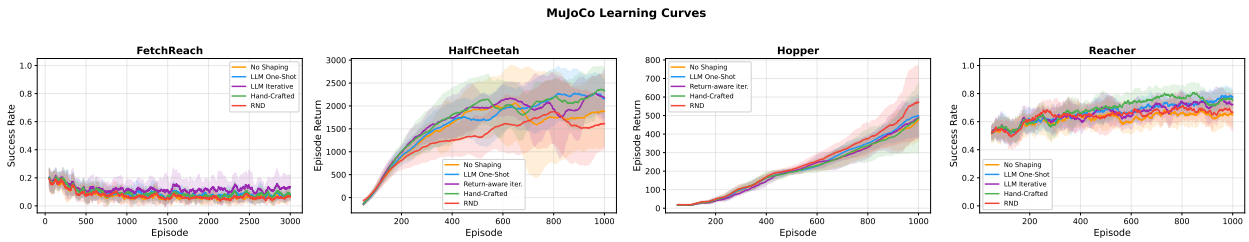

Figure 5: MuJoCo learning curves (smoothed over 50 episodes). Top: reaching tasks (success rate). Bottom: locomotion tasks (episode return). Shaded regions show ±1 sample std across seeds; return-aware locomotion iterative curves use 6 seeds and other conditions use 10 seeds.

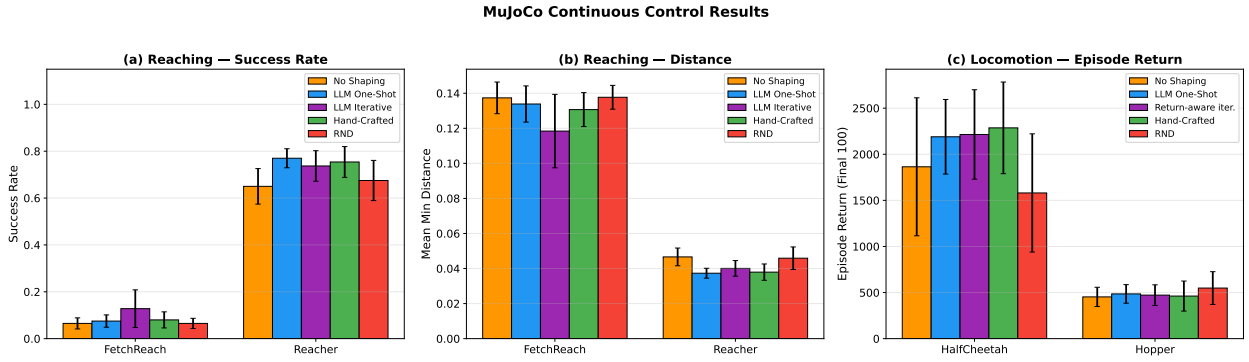

Figure 6: MuJoCo final performance: success rate for reaching tasks, episode return for locomotion tasks. Error bars show ±1 sample std across seeds; return-aware locomotion iterative bars use 6 seeds and other conditions use 10 seeds.

9.3 Secondary Comparisons

We keep the main analysis focused on the diagnostic mechanism. Additional comparisons with RND, a naive LLM+RND combination, exploratory model sensitivity, qualitative reward-function patterns, and protocol-level cost are summarized in Appendix A. These analyses provide useful context, but they are not the central evidence for the debugging thesis.

9.4 Dense-Reward Boundary Case

The dense-reward locomotion result is primarily a boundary finding. The sharpest counterexample is Hopper: with the same total episode budget (1,600), no-shaping extended reaches 882 while the original success-diagnostic iterative protocol reaches only 525, and the return-aware adaptation remains lower at 472 with a shorter 1,000-episode final training run. Thus the refinement loop provides no advantage when the primary bottleneck is training duration rather than reward-function quality. HalfCheetah is less negative but still not a robust gain: success-diagnostic iterative refinement achieves 2,040 return compared to 2,190 for one-shot, and return-aware refinement is statistically indistinguishable from one-shot.

Root cause. Investigation reveals a systematic false-positive failure in the diagnostic system. The reward-hacking detector fires when mean reward is high but success rate is low ($\text{mr} > 0.5 \wedge \text{sr} < 0.2$). On locomotion environments, `is_success` is always `False` (there is no binary success criterion), so $\text{sr} = 0$ while $\text{mr} \sim 2{,}000$. This triggers a "REWARD HACKING DETECTED" warning *every iteration*, causing the LLM to systematically strip effective shaping components from its reward function.

Mechanism-removal diagnostic adaptation. We disable success-based diagnostics for dense-reward environments and replace them with *return-trend analysis*: the system compares mean return in the first vs. second half of the probe run, flagging "RETURN DECLINING" (second half $< 0.9\times$ first half, indicating destabilizing shaping) or "RETURN STAGNATED" ($|\Delta| < 0.05\times$ first half, indicating insufficient signal). A dedicated locomotion refinement prompt provides domain-specific guidance: velocity milestone bonuses, stability bonuses, and magnitude constraints (<10% of typical episode return). This prompt omits success rate (always zero, confusing to the LLM) and instead frames diagnostics in terms of return trajectory.

We evaluate the adaptation on the first 6 completed canonical seeds per environment (Table 13). On HalfCheetah, the return-aware iterative method achieves $2{,}215 \pm 485$ mean return, numerically above the broken iterative baseline ($2{,}040 \pm 685$) and no shaping ($1{,}864 \pm 749$) but not distinguishable from one-shot. On Hopper, the adapted method (472 ± 112) is comparable to no shaping (453 ± 104) but below both the broken iterative baseline (525 ± 161) and the budget-matched no-shaping control, consistent with Hopper benefiting primarily from training duration rather than reward function quality (Section 9.5). Critically, the return-trend diagnostics **eliminate one specific systematic false-positive failure**: no run triggers

the spurious "REWARD HACKING" warning that previously caused the LLM to strip effective shaping components.

Env	No Shape	One-Shot	Broken Iter.	Return-Aware
HalfCheetah	1864 ± 749	2190 ± 405	2040 ± 685	2215 ± 485
Hopper	453 ± 104	485 ± 101	525 ± 161	472 ± 112

Table 13: Dense-reward locomotion adaptation (mean ± sample std). Return-aware diagnostics remove the systematic false-positive reward-hacking warning, but do not establish robust locomotion gains. Return-aware results use the first 6 completed canonical seeds; other columns use 10 seeds.

This boundary case illustrates a central lesson: **iterative refinement diagnostics must be aligned with the task's evaluation signal**. In the sparse structured tasks studied here, where the diagnostics align with binary success signals, the framework is most effective. When they are misaligned (dense reward, no success criterion), the feedback loop becomes counterproductive. Return-trend analysis removes the specific false-positive reward-hacking mechanism in locomotion, but the Hopper result shows that removing one diagnostic failure mechanism is not enough to produce metric gains in dense continuous control.

9.5 Episode Budget Control

The iterative protocol uses more total training episodes than one-shot (3×500 probe + 3,000 full = 4,500 for MiniGrid; 3×200 + full for MuJoCo). To control for this confound, we run extended-budget experiments where no-shaping and LLM one-shot conditions train for the same total episodes as iterative (Figure 7). On the hardest MiniGrid environments, iterative refinement's advantage persists strongly: DoorKey-8×8 iterative (97.6%) far exceeds both no-shaping extended (4.7%; raw Welch $p < 10^{-16}$, $d = 21.3$) and LLM extended (57.9%). KeyCorridor iterative (86.7%) similarly outperforms no-shaping extended (12.9%; raw Welch $p = 1.4 \times 10^{-5}$, $d = 3.6$) and LLM extended (54.8%). These two extended no-shaping comparisons are part of the manuscript-defined primary family in Appendix B. UnlockPickup iterative (24.3%) also exceeds no-shaping extended (0.3%; raw Welch $p < 10^{-4}$, $d = 3.3$), but we treat this budget-control comparison as descriptive rather than part of Table 15. These results support the interpretation that the sparse-task gains come from reward function quality rather than additional training time. However, on Hopper, the budget-matched no-shaping control (882) far exceeds both standard no-shaping (453) and the original success-diagnostic iterative protocol (525). Even the return-aware adaptation remains below this matched-budget control (472 at 1,000 episodes vs. 882 at 1,600 episodes), indicating that Hopper's bottleneck is training duration rather than reward design.

9.6 Variance Decomposition

To understand when iterative refinement is most useful, we decompose observed performance variability into two components: variability from LLM reward-function generation and variability from RL training stochasticity. We decompose these sources by fixing one and varying the other: (1) fixing the LLM-generated reward function from seed 42 and training with all 10 RL seeds to approximate RL variance, and (2) fixing RL seed 42 and training with each of the 10 LLM-generated reward functions to approximate LLM-generation variance. This should be interpreted as a *point-estimate decomposition* rather than a formal variance partition: because each component is anchored to a single fixed seed, the two terms are not orthogonal and need not sum to 100%.

In the sparse, structured MiniGrid environments, LLM-function variance is typically much larger than RL variance. LLM variance point estimates range from 59–118% of total variance, while RL variance is generally near zero (Figure 8). DoorKey-8×8 shows RL std of just 0.3% vs. LLM std of 47.5%; DoorKey-5×5 shows zero RL variance vs. 31.7% LLM std. This suggests that performance is more sensitive to *which* reward function the LLM produces than to RL training stochasticity, providing empirical support for why diagnostic refinement—improving a poor reward function—can materially change outcomes in these tasks.

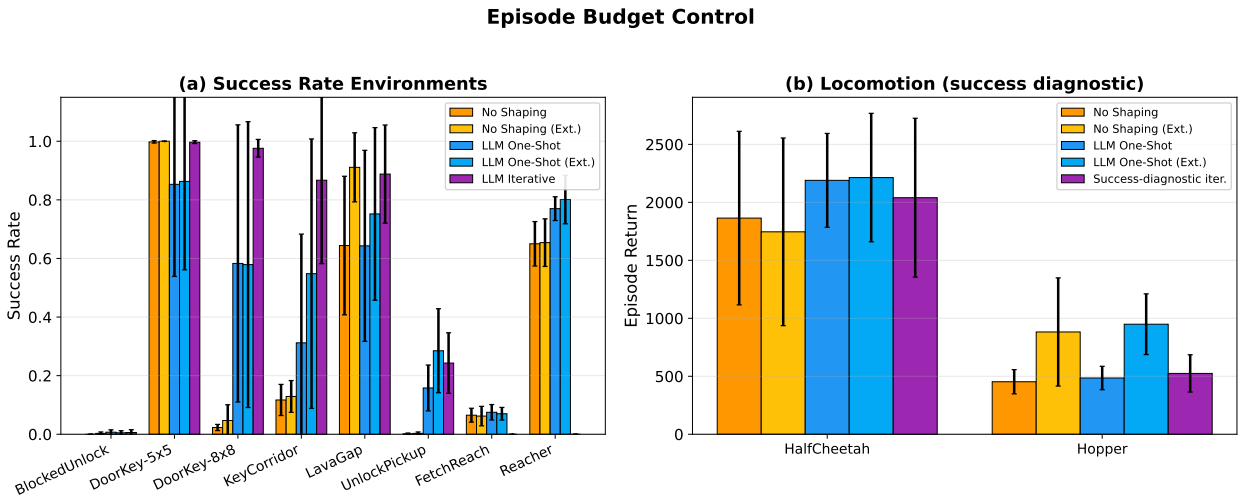

Figure 7: Episode budget control: extended-budget conditions (4,500 episodes for MiniGrid) compared with standard and iterative conditions. MiniGrid extended-budget runs use 4,500 episodes; MuJoCo extended-budget runs use full budget + 600 probe-equivalent episodes. Seed counts match the corresponding main/budget-control runs unless otherwise noted. For locomotion, the iterative bar uses the original success-diagnostic protocol, not the main Table 2 return-aware adaptation; it is retained here as the broken diagnostic baseline.

In the continuous-control environments, the pattern shifts. RL variance point estimates are larger (124–440% of total variance), whereas LLM variance is more moderate (44–153%). Although these ratios should not be interpreted literally as variance shares, they suggest that RL training stochasticity plays a larger role in continuous-control settings, consistent with the weaker gains from iterative refinement observed in the locomotion domain.

Limitations of this decomposition. Because each marginal variance is estimated using a single anchor seed rather than a fully crossed design, the components are not statistically independent, and their sum can exceed 100% as observed in several environments. We report bootstrap 95% confidence intervals (2,000 resamples, reproducible seed) alongside point estimates; these intervals are wide in several settings, especially continuous-control environments (e.g., HalfCheetah LLM std point estimate 499, 95% CI $[77, 722]$), indicating substantial instability in the exact ratio estimates. Accordingly, the anchored decomposition is best read as directional evidence about which source of variability appears more important in each regime, rather than as a precise estimate of variance explained.

Fully crossed decomposition. To complement the anchored analysis, we run a fully crossed 5×5 design (5 LLM reward functions $\times$ 5 RL seeds, all 25 combinations trained) on four representative environments spanning the variance spectrum. This supports a two-way random-effects variance decomposition with the standard estimator $\hat{\sigma}^2_{\text{LLM}} = (\text{MS}_{\text{LLM}} - \text{MS}_{\text{err}})/R$ (and symmetrically for RL), clamping negative components to zero. Because there is one observation per LLM–RL cell, the residual term includes interaction effects and unmodeled stochasticity. Table 14 reports point-estimate variance shares and row/column-bootstrap intervals over the LLM and RL seed axes.

Table 14 suggests a domain split in the crossed design that the anchored decomposition could only approximate, but the bootstrap intervals are wide and should temper any fine-grained ranking. On the sparse discrete tasks, **LLM-function variance has the largest point estimate**: DoorKey-8×8 attributes 96% of variance to the LLM reward function identity (CI $[66, 100]$%), indicating that the identity of the LLM-generated reward function accounts for most observed performance variation in this setting. KeyCorridor shows a more balanced and uncertain split (50% LLM, CI $[0, 90]$%; 50% residual, CI $[6, 94]$%), suggesting that LLM–RL interaction effects may play a larger role on this harder task. On the continuous-control tasks, the residual share has the largest point estimate (57–63%), but the LLM and RL component intervals are

Environment	n	LLM share	RL share	Residual
DoorKey-8×8	25	**96.0**% [66.4, 100.0]	0.0% [0.0, 13.4]	4.0% [0.0, 25.6]
KeyCorridor	25	**50.2**% [0.0, 89.8]	0.0% [0.0, 51.1]	49.8% [6.3, 94.4]
Reacher-v4	25	19.3% [0.0, 69.8]	23.4% [0.0, 100.0]	**57.3**% [0.0, 85.5]
Hopper-v4	25	7.3% [0.0, 62.9]	30.1% [0.0, 80.7]	**62.6**% [12.8, 93.2]

Table 14: Crossed 5×5 variance decomposition via two-way random-effects ANOVA (25 cells per env, one observation per cell). Brackets show row/column-bootstrap 95% intervals for variance shares, using the same negative-clamped estimator as the point estimates. The residual includes LLM–RL interaction effects and unmodeled stochasticity. **Bold** marks the dominant variance component. Metric: success rate for MiniGrid, final_100_reward for MuJoCo.

broad. The anchored design's 275.6% and 439.5% RL shares on these envs (Figure 8) were inflated by its non-orthogonal single-seed anchoring; the crossed design provides cleaner descriptive point estimates, not precise variance-share claims.

This decomposition supports the method's scoped interpretation most clearly for DoorKey-8×8: iterative refinement—which improves the LLM reward function—should be most useful where LLM variance appears dominant. For KeyCorridor and the continuous-control tasks, the crossed estimates are better read as directional diagnostics than as settled component rankings.

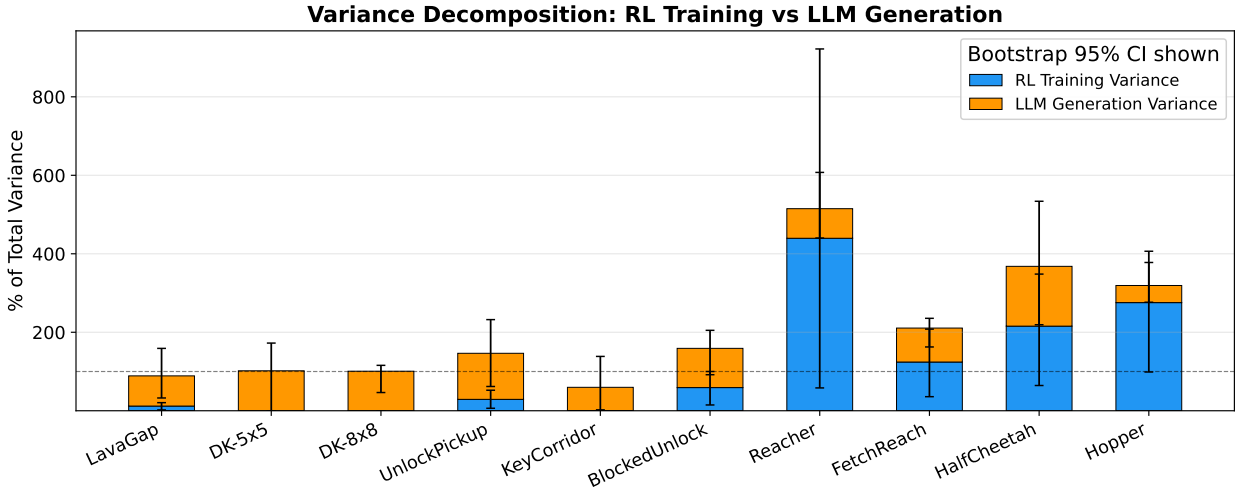

Figure 8: Anchored variance ratios for RL training vs. LLM generation per environment (single-seed anchored design). Ratios can exceed 100% and are not variance shares because the components are non-orthogonal; see Table 14 for the fully crossed four-environment design.

9.7 Selection vs. Refinement

An alternative to iterative refinement is *selection*: generate N independent reward functions, evaluate each via a short probe, and train with the best. We implement best-of-3 selection with the same total episode budget as iterative (3×500 probe + 3,000 full for MiniGrid). This approach captures some benefit of variance reduction by selecting the best from multiple independent samples.

Figure 9 compares best-of-3 selection with one-shot and iterative refinement. On DoorKey-8×8, best-of-3 (92.3%) substantially improves over one-shot (58.3%) by filtering out poor candidates, though iterative (97.6%) still achieves the highest mean performance. On KeyCorridor, best-of-3 (60.1%) roughly doubles one-shot (31.2%) but remains below iterative (86.7%). On UnlockPickup, best-of-3 (24.3%) matches iterative (24.3%), both improving over one-shot (15.8%). Seed-level tests do not resolve the iterative-vs-selection differences at $n = 10$: DoorKey-8×8 $\Delta = 5.3$ points (Welch $p = 0.51$, bootstrap CI $[-3.8, 21.4]$), KeyCorridor

$\Delta = 26.6$ points ($p = 0.106$, CI $[-2.6, 54.8]$), and UnlockPickup $\Delta = 0.0$ points ($p = 1.0$, CI $[-7.9, 6.9]$). We therefore interpret Best-of-3 as a strong selection baseline: it captures most of the DoorKey-8×8 gain and matches UnlockPickup, while the KeyCorridor mean remains descriptively lower because independent candidates often share systematic flaws. Note that best-of-3 MuJoCo locomotion results are based on fewer seeds (HalfCheetah: 5/10, Hopper: 4/10) due to reward function validation failures, limiting their statistical reliability.

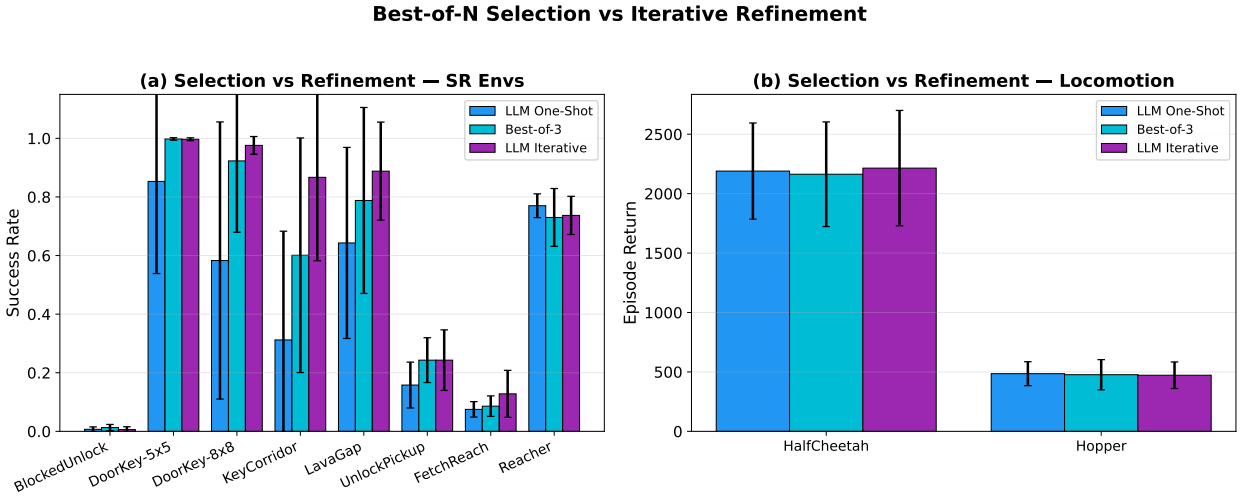

Figure 9: Best-of-3 selection vs. LLM one-shot vs. iterative refinement. MiniGrid Best-of-3 uses 10 seeds; MuJoCo locomotion Best-of-3 has incomplete validation-successful seeds (HalfCheetah: 5/10, Hopper: 4/10).

10 Related Work

LLMs as reward designers. Eureka (Ma et al., 2024) uses GPT-4 to generate reward functions for continuous-control tasks, employing evolutionary search over hundreds of candidates evaluated in GPU-parallel simulation (IsaacGym). Text2Reward (Xie et al., 2024) generates dense executable reward code from language task descriptions and compact environment representations. ELLM (Du et al., 2023) uses LLMs to suggest exploration goals rather than reward functions. Language to Rewards (Yu et al., 2023) translates task specifications into reward parameters for manipulation. More recent work extends this paradigm: Auto MC-Reward (Li et al., 2024) applies automated dense reward design to Minecraft, CARD (Sun et al., 2025) refines LLM-generated reward code using dynamic feedback and trajectory preference evaluation, and RL-VLM-F (Wang et al., 2024) uses vision-language models for reward feedback. Kwon et al. (Kwon et al., 2023) study prompting LLMs as proxy reward functions from zero- or few-shot task descriptions. Our work complements these methods by providing a systematic failure-mode taxonomy and a low-call diagnostic refinement protocol; the comparison is protocol-level, and direct performance comparison with population-based methods remains a shared-benchmark limitation.

Reward misspecification and hacking. A growing literature studies the risks of misspecified rewards. Amodei et al. (Amodei et al., 2016) identify reward hacking as a key AI safety concern. Skalse et al. (Skalse et al., 2022) provide a formal characterization of reward hacking. Pan et al. (Pan et al., 2022) map the effects of reward misspecification on learned policies. Clark and Amodei (Clark & Amodei, 2016) document faulty reward functions in deployed systems. Our failure-mode taxonomy (reward flooding, semantic/API misunderstanding, and rarer weak-shaping cases) connects to this literature by characterizing the specific ways LLM-generated rewards become misspecified, and our diagnostic protocol provides a lightweight way to flag these failure modes during refinement.

Reward shaping theory. Potential-based reward shaping (Ng et al., 1999) provides theoretical guarantees on policy invariance under additive shaping. Devlin and Kudenko (Devlin & Kudenko, 2012) extend this to

dynamic potentials. Wiewiora et al. (Wiewiora et al., 2003) establish connections between potential-based shaping and Q-value initialization. Hu et al. (Hu et al., 2020) propose learning to utilize shaping rewards adaptively. Our LLM-generated one-time bonuses can be viewed as rough approximations to potential-based shaping (Section 4), connecting the empirical LLM approach to this theoretical reference point without implying a formal invariance guarantee.

Intrinsic motivation. Count-based exploration (Bellemare et al., 2016), curiosity-driven methods (Pathak et al., 2017), and Random Network Distillation (Burda et al., 2019) provide intrinsic rewards without domain knowledge. Oudeyer et al. (Oudeyer et al., 2007) survey intrinsic motivation systems for autonomous development. Go-Explore (Ecoffet et al., 2021) uses archiving and goal-directed exploration. Taiga et al. (Taiga et al., 2021) provide a systematic evaluation of bonus-based exploration. We include fixed-coefficient RND as a contextual baseline and add a small DoorKey coefficient sweep. The sweep confirms that RND is highly coefficient-sensitive: tuning solves DoorKey-5×5 but does not rescue DoorKey-8×8 under our PPO protocol.

LLMs for code generation. Code-generating LLMs (Chen et al., 2021) have been applied broadly, with specialized models like StarCoder (Li et al., 2023) and Code Llama (Roziere et al., 2023) achieving strong performance on programming benchmarks. In RL, this capability enables generating executable reward functions (Ma et al., 2024), embodied-agent skill code (Wang et al., 2023), and policy code (Liang et al., 2023). Our iterative refinement with training diagnostics is analogous to the code-test-debug cycle in software engineering, but applied to reward function design rather than general programming.

11 Limitations and Future Work

Scope: structured interfaces required. Our method depends on a curated reward-function interface: the `info` dictionary exposes semantic fields (`carrying`, `agent_pos`) and event descriptions (`event_text`). An interface ablation (Section 8.6) shows the method does not critically depend on any single field—removing `event_text` is neutral on DoorKey-8×8 and actually *improves* KeyCorridor—but it does require *some* structured semantic interface. In environments without such interfaces, the LLM would need to interpret raw observations (e.g., pixel arrays), which is a fundamentally harder problem that our current framework does not address. This domain-specificity echoes recent LLM systems work in financial document routing and reranking (Cheng et al., 2026a;b) and hardware-aware vision–language deployment (Chen et al., 2025), where task schemas, retrieval structure, or deployment substrate materially affect system behavior.

Dense-reward settings require different diagnostics. As demonstrated in Section 9.4, success-based heuristics are counterproductive on dense-reward tasks. Our return-trend adaptation addresses the specific locomotion failure mechanism, but other dense-reward domains may require further diagnostic adaptations. The framework is extensible, but each new domain class may need its own diagnostic design.

Diagnostic thresholds. The diagnostics use fixed heuristic thresholds (Table 16). Our sensitivity analysis (Section 8.4) shows that thresholds materially affect performance but in diagnostically interpretable ways, and the stress test (Section 8.1) reveals that the reward-hacking detector can generate false positives that hurt performance on some environments. Adaptive threshold selection—e.g., based on environment characteristics or early training dynamics—is a natural direction for future work.

Environment complexity. While we evaluate across 13 environments spanning discrete grid-worlds, continuous reaching, and continuous locomotion, these use low-dimensional state observations. Extending to high-dimensional observation spaces and long-horizon manipulation tasks would test the scope of the approach beyond the structured setting studied here.

Scaling to harder tasks. BlockedUnlockPickup remains below 1% success across all methods, MultiRoom-N6 near 0%, and UnlockPickup achieves only ∼24% with iterative refinement. Tasks requiring very long planning horizons or complex reward decompositions remain challenging.

Model diversity. We test four models across three providers with 3 seeds each (Appendix A). This provides exploratory evidence that the phenomenon is not unique to one model, but the sample size is insufficient for strong model-level claims. Open-source models may exhibit different failure patterns.

Algorithm and baseline scope. All experiments use PPO, so some observed failure modes may reflect interactions between reward design and PPO optimization rather than reward functions alone. All conclusions should therefore be read as PPO-protocol conclusions, not algorithm-agnostic reward-function validity claims. Testing an off-policy algorithm such as SAC would help separate reward-intrinsic failures from algorithm-specific effects. The main RND baseline uses a fixed intrinsic-reward coefficient of 0.1; a post-hoc DoorKey sweep shows that this value is badly mistuned on DoorKey-5×5 but that the swept coefficients still fail on DoorKey-8×8. The comparison with evolutionary reward-design methods is a cost/protocol comparison rather than a shared-benchmark head-to-head evaluation.

Variance decomposition. The anchored decomposition (Section 9.6) uses a single seed per component, producing non-orthogonal point estimates. We supplement it with a fully crossed 5×5 design on four representative environments (Table 14), which provides a cleaner two-way random-effects partition while leaving interaction effects in the residual. The crossed row/column-bootstrap intervals are wide, so extending the crossed design beyond these four environments would further strengthen the analysis.

Refinement protocol design. The current protocol discards the probe-trained policy and retrains from scratch, which simplifies evaluation but wastes probe computation; warm-starting from the probe policy could improve sample efficiency. In addition, strong diagnostic warning labels such as "REWARD HACKING DETECTED" may prime the LLM toward overcorrection. Softer or more specific feedback is an important future prompt-design variable.

Temperature and per-seed LLM calls. LLM temperature is fixed at 0.4 throughout; systematic temperature tuning is left for future work. Each seed uses its own LLM-generated reward function, conflating LLM and RL variance; our decomposition analysis quantifies but does not eliminate this confound.

12 Conclusion

We have argued that LLM reward design for sparse, structured RL tasks is better understood as a debugging problem than a generation problem. One-shot generation fails in recurring, identifiable ways; lightweight diagnostics can often detect and repair these failures cheaply in the studied sparse structured setting; and the approach breaks predictably when diagnostics are misaligned with the task's evaluation signal.

The evidence supports this thesis on multiple fronts. The failure taxonomy identifies reward flooding and semantic/API misunderstanding as dominant patterns in our audit, with weak or negligible shaping appearing as a rarer and less reliably labeled case. Diagnostic-driven iterative refinement achieves 97.6% success on DoorKey-8×8 (from 2.3% without shaping) and 86.7% on KeyCorridor (from 31.2% one-shot), though KeyCorridor remains high variance across seeds and sensitivity checks. Budget-matched controls remain significant on the primary sparse tasks; Appendix B reports the corrected p-values. The diagnostics are evaluated through component-removal stress tests (which reveal reward-hacking false positives and lower-seed sensitivity), comparison against a metrics-only baseline (29–75 point gaps), and a static-vocabulary control showing that the taxonomy prompt itself recovers much of the metrics-only gap. We therefore interpret the mechanism as taxonomy-guided prompting plus diagnostic refinement, not dynamic warning labels alone. The taxonomy audit against author labels reports 94% precision (32/34 predicted labels) on the two dominant failure modes and 71.4% overall accuracy across all categories.

A fully crossed 5×5 variance decomposition provides descriptive support for when refinement is most useful: on DoorKey-8×8, LLM-function variance has a 96% point-estimate share with a wide bootstrap interval, consistent with reward-function correction being especially useful in that setting. On continuous-control tasks, residual and RL variance have larger point estimates (57–63% residual, 23–30% RL), limiting the benefit of improving any single reward function. The locomotion boundary case demonstrates both the framework's failure mode (success-based false positives on dense-reward tasks) and a diagnostic adaptation that removes that specific failure mechanism without producing robust locomotion gains.

The method is strongest for sparse, structured tasks with reliable semantic interfaces—though an interface ablation shows that richer fields such as `event_text` are not uniformly helpful—and uses far fewer LLM calls than population-based reward-search protocols. The failure-mode taxonomy, diagnostic protocol, and

variance analysis may inform future work on structured reward design beyond the environments studied here.

References

Dario Amodei, Chris Olah, Jacob Steinhardt, Paul Christiano, John Schulman, and Dan Mané. Concrete problems in AI safety. In *arXiv preprint arXiv:1606.06565*, 2016.

Marc Bellemare, Sriram Srinivasan, Georg Ostrovski, Tom Schaul, David Saxton, and Remi Munos. Unifying count-based exploration and intrinsic motivation. *Advances in Neural Information Processing Systems*, 29, 2016.

Yuri Burda, Harrison Edwards, Amos Storkey, and Oleg Klimov. Exploration by random network distillation. In *International Conference on Learning Representations*, 2019.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Wei Chen, Liangmin Wu, Yunhai Hu, Zhiyuan Li, Zhiyuan Cheng, Yicheng Qian, Lingyue Zhu, Zhipeng Hu, Luoyi Liang, Qiang Tang, Zhen Liu, and Han Yang. AutoNeural: Co-designing vision-language models for NPU inference, 2025. URL `https://arxiv.org/abs/2512.02924`.

Zhiyuan Cheng, Longying Lai, and Yue Liu. Resolving the robustness-precision trade-off in financial RAG through hybrid document-routed retrieval, 2026a. URL `https://arxiv.org/abs/2603.26815`.

Zhiyuan Cheng, Longying Lai, Yue Liu, Kai Cheng, and Xiaoxi Qi. Enhancing financial report question-answering: A retrieval-augmented generation system with reranking analysis, 2026b. URL `https://arxiv.org/abs/2603.16877`.

Maxime Chevalier-Boisvert, Bolun Dai, Mark Towers, Rodrigo de Lazcano, Lucas Willems, Salem Lahlou, Suman Pal, Pablo Samuel Castro, and Jordan Terry. Minigrid & miniworld: Modular & customizable reinforcement learning environments for goal-oriented tasks. *Advances in Neural Information Processing Systems*, 36, 2023.

Jack Clark and Dario Amodei. Faulty reward functions in the wild. *OpenAI Blog*, 2016.

Sam Devlin and Daniel Kudenko. Dynamic potential-based reward shaping. In *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems*, pp. 433–440, 2012.

Yuqing Du, Olivia Watkins, Zihan Wang, Cédric Colas, Trevor Darrell, Pieter Abbeel, Abhishek Gupta, and Jacob Andreas. Guiding pretraining in reinforcement learning with large language models. *Proceedings of the 40th International Conference on Machine Learning*, 2023.

Adrien Ecoffet, Joost Huizinga, Joel Lehman, Kenneth O Stanley, and Jeff Clune. First return, then explore. *Nature*, 590:580–586, 2021.

Yujing Hu, Weixun Wang, Hangtian Jia, Yixiang Wang, Yingfeng Chen, Jianye Hao, Feng Wu, and Changjie Fan. Learning to utilize shaping rewards: A new approach of reward shaping. In *Advances in Neural Information Processing Systems*, volume 33, 2020.

Minae Kwon, Sang Michael Xie, Kalesha Bullard, and Dorsa Sadigh. Reward design with language models. In *International Conference on Learning Representations*, 2023.

Hao Li, Xue Yang, Zhaokai Wang, Xizhou Zhu, Jie Zhou, Yu Qiao, Xiaogang Wang, Hongsheng Li, Lewei Lu, and Jifeng Dai. Auto MC-reward: Automated dense reward design with large language models for Minecraft. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 16426–16435, 2024.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. StarCoder: May the source be with you! *Transactions on Machine Learning Research*, 2023.

Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. Code as policies: Language model programs for embodied control. *IEEE International Conference on Robotics and Automation*, 2023.

Yecheng Jason Ma, William Liang, Guanzhi Wang, De-An Huang, Osbert Bastani, Dinesh Jayaraman, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Eureka: Human-level reward design via coding large language models. In *International Conference on Learning Representations*, 2024.

Andrew Y Ng, Daishi Harada, and Stuart Russell. Policy invariance under reward transformations: Theory and application to reward shaping. *Proceedings of the 16th International Conference on Machine Learning*, pp. 278–287, 1999.

Pierre-Yves Oudeyer, Frederic Kaplan, and Verena V Hafner. Intrinsic motivation systems for autonomous mental development. *IEEE Transactions on Evolutionary Computation*, 11(2):265–286, 2007.

Alexander Pan, Kush Bhatia, and Jacob Steinhardt. The effects of reward misspecification: Mapping and mitigating misaligned models. *International Conference on Learning Representations*, 2022.

Deepak Pathak, Pulkit Agrawal, Alexei A Efros, and Trevor Darrell. Curiosity-driven exploration by self-supervised prediction. In *International Conference on Machine Learning*, pp. 2778–2787, 2017.

Matthias Plappert, Marcin Andrychowicz, Alex Ray, Bob McGrew, Bowen Baker, Glenn Powell, Jonas Schneider, Josh Tobin, Maciek Chociej, Peter Welinder, Vikash Kumar, and Wojciech Zaremba. Multi-goal reinforcement learning: Challenging robotics environments and request for research, 2018.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. Code Llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Joar Skalse, Nikolaus H R Howe, Dmitrii Krasheninnikov, and David Krueger. Defining and characterizing reward hacking. *Advances in Neural Information Processing Systems*, 35, 2022.

Shengjie Sun, Runze Liu, Jiafei Lyu, Jing-Wen Yang, Liangpeng Zhang, and Xiu Li. A large language model-driven reward design framework via dynamic feedback for reinforcement learning. *Knowledge-Based Systems*, 326:114065, 2025. doi: 10.1016/j.knosys.2025.114065.

Adrien Ali Taiga, William Fedus, Marlos C Machado, Aaron Courville, and Marc G Bellemare. On bonus-based exploration methods in the arcade learning environment. In *International Conference on Learning Representations*, 2021.

Emanuel Todorov, Tom Erez, and Yuval Tassa. MuJoCo: A physics engine for model-based control. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 5026–5033, 2012.

Mark Towers, Ariel Kwiatkowski, Jordan Terry, John U Balis, Gianluca De Cola, Tristan Deleu, Manuel Goulão, Andreas Kallinteris, Markus Kha, Nicolai Koptenko, et al. Gymnasium: A standard interface for reinforcement learning environments. *arXiv preprint arXiv:2407.17032*, 2024.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023.

Yufei Wang, Zhanyi Sun, Jesse Zhang, Zhou Xian, Erdem Biyik, David Held, and Zackory Erickson. RL-VLM-F: Reinforcement learning from vision language foundation model feedback. In *Proceedings of the 41st International Conference on Machine Learning*, 2024.

Eric Wiewiora, Garrison W Cottrell, and Charles Elkan. Principled methods for advising reinforcement learning agents. *Proceedings of the 20th International Conference on Machine Learning*, pp. 792–799, 2003.

Tianbao Xie, Siheng Zhao, Chen Henry Wu, Yitao Liu, Qian Luo, Victor Zhong, Yanchao Yang, and Tao Yu. Text2reward: Reward shaping with language models for reinforcement learning. In *International Conference on Learning Representations*, 2024.

Wenhao Yu, Nimrod Gileadi, Chuyuan Fu, Sean Kirmani, Kuang-Huei Lee, Montse Gonzalez Arenas, Hao-Tien Lewis Chiang, Tom Erez, Leonard Hasenclever, Jan Humplik, et al. Language to rewards for robotic skill synthesis. *arXiv preprint arXiv:2306.08647*, 2023.

A Secondary Analyses and Practical Takeaways

Intrinsic motivation. With the fixed coefficient used in the main table, RND behaves differently from LLM-designed shaping. It is competitive on LavaGap where broad exploration is useful, but the DoorKey sweep shows high coefficient sensitivity: $c = 0.01$ solves DoorKey-5×5, while $c = 0.1$ and $c = 0.5$ do not, and none of the three coefficients solves DoorKey-8×8. A naive LLM+RND additive combination also hurts structured MiniGrid tasks (e.g., DoorKey-8×8 drops from 58.3% with LLM alone to 0.1%), suggesting that task-specific and domain-agnostic rewards require careful balancing rather than simple addition.

Evolutionary reward design and cost. Population-based methods such as Eureka evaluate many reward candidates in GPU-parallel simulation. Our protocol instead uses 1–4 LLM calls per reward-design run, repeated across seeds for evaluation, and CPU training. This is a protocol-level efficiency comparison, not a shared-benchmark performance comparison: evolutionary search may find better functions when broad search and GPU simulation are available, while diagnostic refinement is aimed at low-call correction when failure modes are identifiable.

Qualitative reward-function patterns. Generated rewards commonly use event-driven one-time bonuses, position-based progress heuristics, and bonus magnitudes comparable to hand-crafted shaping after refinement. These qualitative patterns support the debugging interpretation but are secondary to the controlled comparisons in the main text.

Exploratory model sensitivity. A small sweep across Claude Haiku, Claude Sonnet 4.6, GPT-5.2, and Grok 4 on four environments with 3 seeds each suggests that one-shot variance is not unique to one model and that refinement can help across model instances. The sample size is too small for strong model-level claims.

Practical takeaways. One-shot generation is best suited to straightforward tasks with clear subgoals; iterative refinement is most useful for sparse structured tasks where one-shot generation is high variance; Best-of-N selection is useful when independent candidates include a good reward; dense-reward tasks require diagnostic adaptation; and the variance analysis suggests prioritizing reward-function quality when LLM variance dominates, while prioritizing RL optimization when training stochasticity dominates.

B Primary Statistical Tests

Table 15 lists the 12 manuscript-defined primary comparisons used for the "9/12 significant" statement in Table 2. This final TMLR-prep family (version 2026-05-11) focuses on the sparse structured tasks where the method is expected to help most, includes two sparse-task budget controls (DoorKey-8×8 and KeyCorridor iterative vs. extended no-shaping), and keeps MuJoCo reaching/locomotion comparisons in separate families as boundary checks. The set is not an external preregistration, and it is not meant to imply that every cell in Table 2 or every descriptive statistic in Section 9.5 is part of the primary test family. We apply Welch's

t-test within each environment and Holm–Bonferroni correction separately within the MiniGrid, MuJoCo reaching, and MuJoCo locomotion families. The table is generated from the same logs as Table 2; locomotion uses episode return and all other rows use final-100-episode success rate.

Family / Env	**Primary Comparison**	Δ	p_{corr}	**Sig.**
MiniGrid / DoorKey-8×8	Iterative vs. No Shaping	+95.3 pp	< 0.0001	yes
MiniGrid / KeyCorridor	Iterative vs. No Shaping	+75.0 pp	0.0001	yes
MiniGrid / UnlockPickup	Iterative vs. No Shaping	+24.2 pp	0.0001	yes
MiniGrid / DoorKey-8×8	Iterative vs. One-Shot	+39.3 pp	0.0275	yes
MiniGrid / KeyCorridor	Iterative vs. One-Shot	+55.5 pp	0.0032	yes
MiniGrid / DoorKey-8×8	Iterative vs. Extended No Shaping	+92.9 pp	< 0.0001	yes
MiniGrid / KeyCorridor	Iterative vs. Extended No Shaping	+73.8 pp	0.0001	yes
MuJoCo Reaching / Reacher	One-Shot vs. No Shaping	+12.0 pp	0.0012	yes
MuJoCo Reaching / FetchReach	Iterative vs. No Shaping	+6.3 pp	0.0373	yes
MuJoCo Locomotion / HalfCheetah	One-Shot vs. No Shaping	+326 return	0.7395	no
MuJoCo Locomotion / Hopper	One-Shot vs. No Shaping	+32 return	0.9795	no
MuJoCo Locomotion / HalfCheetah	Return-Aware Iterative vs. One-Shot	+25 return	0.9795	no

Table 15: Manuscript-defined primary hypothesis tests used for the "9/12 significant" statement in Table 2. Significance is assessed at $p_{\text{corr}} < 0.05$ after per-family Holm–Bonferroni correction. pp = percentage points.

C Threshold Definitions

Table 16 enumerates all trigger values used in the iterative refinement procedure. All values correspond exactly to those used in our implementation.

Category	**Component**	**Value**	**Applies To**
Refinement Control			
	Success threshold τ	0.95	Binary-success tasks
	Max iterations K	3	All environments
	Probe episodes N_p	500	MiniGrid
	Probe episodes N_p	200	MuJoCo
	Full episodes N_f	3,000	MiniGrid, FetchReach
	Full episodes N_f	1,000	Reacher, HalfCheetah, Hopper
Diagnostics (Sparse-Reward)			
	Reward hacking	$\text{mr} > 0.5 \ \wedge \ \text{sr} < 0.2$	MiniGrid, reaching
	Shaping weakness	$\text{mr} < 0.1 \ \wedge \ \text{sr} < 0.1$	MiniGrid, reaching
	Learning plateau	$0.1 < \text{sr} < 0.7 \ \wedge \ \text{eps} > 1000 \ \wedge \ \Delta\text{sr} < 0.05$	Longer MiniGrid probes
Diagnostics (Dense-Reward)			
	Reward hacking	Disabled	Locomotion
	Return declining	$\bar{r}_2 < 0.9 \times \bar{r}_1$	Locomotion
	Return stagnated	$\lvert\bar{r}_2 - \bar{r}_1\rvert < 0.05 \times \bar{r}_1$	Locomotion
Evaluation			
	Success criterion	distance < 0.05	Reacher, FetchReach
	Sample efficiency	80% success rate	Figure 3

Table 16: All trigger values and thresholds used in the iterative refinement procedure. No hidden defaults exist beyond this table.

D Prompt Templates

D.1 Generation Prompt

The following prompt template is used for MiniGrid one-shot reward function generation. The `{env_description}` placeholder is filled with a natural-language description of the specific environment. The supplementary code includes the MuJoCo reaching and locomotion prompt variants; these replace the discrete action description with continuous observation/action fields, and the locomotion variant uses return-trend diagnostics rather than binary success diagnostics.

```
You are an expert reward function designer for RL.

Given a description of an environment, write a
Python reward shaping function that helps a PPO
agent learn faster.

# ENVIRONMENT
{env_description}

# FUNCTION SIGNATURE
def reward_fn(obs, action, reward, terminated,
              truncated, info, state):
    # obs: numpy array (7,7,3)
    # action: int (0=left, 1=right, 2=forward,
    #             3=pickup, 4=drop, 5=toggle)
    # info: dict with "event_text", "carrying",
    #        "agent_pos", "step_count", "max_steps"
    # state: mutable dict, reset each episode
    # Returns: (shaped_reward, state)

# DESIGN PRINCIPLES
1. Start with the original reward.
2. Add one-time bonuses (+0.1 to +0.3) for subgoals.
3. Use state dict to ensure bonuses given only ONCE.
4. Check info["event_text"] for events.
5. Keep bonuses small vs goal reward (~1.0).
6. No imports. Self-contained function.
```

D.2 Refinement Prompt

For iterative refinement, the prompt additionally includes the previous reward function source code, training metrics, and diagnosed failure modes:

```
# CURRENT REWARD FUNCTION
{current_source}

# TRAINING RESULTS
- Episodes trained: {episodes_trained}
- Success rate: {success_rate}
- Mean reward: {mean_reward}

# DIAGNOSED ISSUES
{diagnoses}

# COMMON FAILURE MODES TO AVOID
1. Reward flooding: Do NOT add per-step bonuses.
2. Action-index confusion: MiniGrid action
   0=turn_left, 2=forward, NOT directions.
3. Too-weak shaping: +0.1 may be too small.
   Use position-based progress tracking.
```

E Iterative Refinement Example

Table 17 shows the iterative refinement process on DoorKey-8×8 (seed 42). The LLM progressively improves the reward function based on probe training diagnostics.

Iter	Probe SR	Key Changes
0	29%	Basic one-time bonuses for key pickup (+0.2) and door open (+0.25). Continuous holding-key bonus causes reward flooding.
1	67%	Removed continuous bonus. Added position-based progress: bonus for reaching new max x-coordinate. Tracks exploration state.
2	78%	Increased door bonus to +0.3. Added room-aware distance tracking. Stagnation penalty (60 steps).
Final (3000 eps)		**99%** success rate

Table 17: Iterative refinement on DoorKey-8×8 seed 42. Each iteration receives the previous function's code and probe metrics. The LLM diagnoses issues and revises the reward function, improving from 29% to 99%. See Appendix F for the actual generated code.

The progression shows the LLM addressing specific weaknesses: iteration 1 adds exploration incentives when basic milestones prove insufficient, and iteration 2 increases bonus magnitudes and adds anti-stagnation measures when the agent plateaus at 67%.

F LLM-Generated Reward Functions

We present the actual LLM-generated reward functions from the DoorKey-8×8 iterative refinement (seed 42), showing how the model evolves its reward design across iterations.

F.1 Iteration 0: Initial One-Shot Generation (Probe SR: 29%)

The initial function uses basic event-driven bonuses but includes two problematic elements: a continuous per-step reward for holding the key (`+0.01`) that can cause reward flooding, and a late-game penalty that discourages exploration.

```
def reward_fn(obs, action, reward, terminated,
              truncated, info, state):
    shaped_reward = reward
    if "key_picked_up" not in state:
        state["key_picked_up"] = False
    if "door_opened" not in state:
        state["door_opened"] = False
    if "goal_reached" not in state:
        state["goal_reached"] = False
    event_text = info.get("event_text", "")
    carrying = info.get("carrying", "")
    step_count = info.get("step_count", 0)
    max_steps = info.get("max_steps", 1)
    # One-time bonuses
    if not state["key_picked_up"] \
       and "picked up" in event_text.lower() \
       and "key" in event_text.lower():
        shaped_reward += 0.2
        state["key_picked_up"] = True
    if not state["door_opened"] \
       and "opened door" in event_text.lower():
        shaped_reward += 0.25
        state["door_opened"] = True
    # PROBLEM: continuous per-step bonus
    if state["key_picked_up"] \
       and not state["door_opened"] \
       and "key" in carrying.lower():
        shaped_reward += 0.01  # reward flooding
    # Late-game penalty
    if not state["goal_reached"] \
       and step_count > max_steps * 0.8:
        shaped_reward -= 0.02
    return shaped_reward, state
```

F.2 Iteration 3: Final Refined Function (Final SR: 99%)

After three refinement rounds with diagnostic feedback, the function removes continuous bonuses, adds room-aware progress tracking, and includes stagnation detection:

```
def reward_fn(obs, action, reward, terminated,
              truncated, info, state):
    shaped_reward = reward
    if "key_picked_up" not in state:
        state["key_picked_up"] = False
    if "door_opened" not in state:
        state["door_opened"] = False
    if "last_agent_pos" not in state:
        state["last_agent_pos"] = (0, 0)
    if "steps_without_position_change" not in state:
        state["steps_without_position_change"] = 0
    if "max_x_before_door" not in state:
        state["max_x_before_door"] = 0
    if "max_x_after_door" not in state:
        state["max_x_after_door"] = 0
    event_text = info.get("event_text", "").lower()
```

```
    agent_pos = info.get("agent_pos", (0, 0))
    current_x, current_y = agent_pos
    # ONE-TIME milestone bonuses only
    if not state["key_picked_up"] \
       and "picked up" in event_text \
       and "key" in event_text:
        shaped_reward += 0.2
        state["key_picked_up"] = True
    if not state["door_opened"] \
       and "opened door" in event_text:
        shaped_reward += 0.3
        state["door_opened"] = True
    # Stagnation detection
    if agent_pos != state["last_agent_pos"]:
        state["steps_without_position_change"] = 0
    else:
        state["steps_without_position_change"] += 1
    # Room-aware progress (no continuous reward)
    if not state["door_opened"]:
        if current_x > state["max_x_before_door"]:
            state["max_x_before_door"] = current_x
    else:
        if current_x > state["max_x_after_door"]:
            state["max_x_after_door"] = current_x
    # Stagnation penalty (only after 60 stuck steps)
    if state["steps_without_position_change"] > 60 \
       and reward < 0.5:
        shaped_reward -= 0.05
    state["last_agent_pos"] = agent_pos
    return shaped_reward, state
```

Key improvements the LLM made across iterations: (1) removed the continuous `+0.01` holding-key bonus that caused reward flooding, (2) added room-aware x-coordinate progress tracking to guide exploration in the 8×8 grid, (3) replaced the coarse late-game penalty with a position-based stagnation detector (60 steps without movement), and (4) increased the door-opening bonus from 0.25 to 0.3. These changes resemble the kinds of local refinements a reward engineer might make after inspecting training diagnostics.