Alignment Flywheel-Guided Suppression Patching for Interactionless Inverse Reinforcement

Elias Malomgré [1], **Pieter Simoens** [1]
elias.malomgre@ugent.be, pieter.simoens@ugent.be

[1]**Department of Information Technology Science, Ghent University of**
[1]**IDLab, Ghent University - imec)**

Abstract

Reward-free task specification methods replace hand-designed rewards with signals derived from demonstrations, implicit feedback, safety constraints, social norms, or other non-scalar sources. Once a learned reward model is optimized by a downstream RL policy, however, the policy can exploit regions where the reward overgeneralizes beyond the support of the signal that induced it. We introduce *flywheel patching*, a mechanism for correcting such failures without retraining the reward model. An Alignment Flywheel identifies where reward should be removed, using reward-free verifier or human feedback, and compiles accepted corrections into localized suppression patches in the deployed reward mapping. We instantiate the mechanism for Interactionless Inverse Reinforcement with MoE-GUIDE, placing patches in the model's bottleneck space and using local patch retrieval during online reward evaluation. We evaluate the patched reward in MuJoCo under a stricter downstream protocol than prior MoE-GUIDE work, removing the expert-state reset curriculum and including HalfCheetah, a previous failure case attributed to reward-model overgeneralization. Main and appendix experiments characterize downstream performance, OOB-margin sensitivity, patch-budget and tightness robustness, bottleneck locality, scalable lookup, and constraint-targeted variants. The results position flywheel patching as a maintenance layer for reward-free task specifications: correction signals need only identify where reward should change, not provide a complete scalar replacement reward.

1 Introduction

Reinforcement learning Sutton & Barto (1998) beyond hand-designed rewards increasingly relies on task specifications derived from demonstrations Abbeel & Ng (2004); Ho & Ermon (2016), preferences Christiano et al. (2017), intrinsic objectives Burda et al. (2018); Pathak et al. (2019), implicit feedback Knox & Stone (2009), safety constraints Alshiekh et al. (2018); Baert et al. (2025), or other reward-free signals Jin et al. (2020). These signals are often easier to obtain than dense scalar rewards: a human, verifier, or perception system may identify an unsafe or unacceptable behavior without specifying how all other behaviors should be ranked. The resulting learned reward model, however, is only useful if it survives downstream policy optimization.

This is not guaranteed since a reward model can fit its inducing signal while still assigning high reward to states outside the intended support. Once optimized by an RL policy, these overgeneralized regions can become attractors: the policy is not passively evaluated by the reward model, but actively searches for behaviors that the model scores well. This connects reward-free task specification to reward misspecification, reward hacking, and reward-model overoptimization Amodei et al. (2016); Pan et al. (2022); Gao et al. (2023).

We study how to repair such failures without retraining the reward model. Our central contribution is *flywheel patching*: an Alignment Flywheel Malomgré & Simoens (2026a;b) identifies where a learned reward overgeneralizes or violates a constraint, and a patching mechanism compiles that correction into a localized suppression of the deployed reward. The learned evaluator remains fixed. Only the reward mapping exposed to downstream RL is edited. This gives users, monitors, or domain-specific verifiers a way to patch out unwanted reward in specific regions of the learned reward surface without rebuilding the reward model. We treat reward repair as a human- and verifier-in-the-loop editing problem: external reward-free signals identify regions where reward should be removed, and the repair mechanism compiles those judgments into the deployed reward.

We formulate this mechanism for Interactionless Inverse Reinforcement (IIRL) Malomgré & Simoens (2026b), where reward models are constructed from offline structure rather than online reward interaction. MoE-GUIDE Malomgré & Simoens (2025) is our concrete instantiation. It learns a mixture-of-experts reconstruction model from state-only expert observations and converts reconstruction consistency into an intrinsic reward. Its bottleneck representation gives a compact space in which local reward patches can be placed.

The immediate motivation is reward-model overgeneralization in downstream MoE-GUIDE, while it allows for future work to use flywheel patching for more advanced norms. Earlier MoE-GUIDE experiments used a short expert-state reset curriculum, initializing early policy rollouts near expert states, and did not produce a working HalfCheetah result. Here, the reward model is still learned from demonstrations, but downstream SAC training proceeds from the standard environment reset distribution. The evaluation is therefore a stricter test of whether learned and patched IIRL rewards can guide policy optimization without privileged expert-state warm starts.

This paper makes four contributions. First, we introduce flywheel-guided patching for IIRL reward models, compiling alignment-flywheel diagnostics into localized reward suppressions. Second, we instantiate and extend the Alignment Flywheel for MuJoCo continuous control, adding locomotion candidate generation, an expert-envelope verifier, a MoE-GUIDE oracle adapter, bottleneck-space patch synthesis, and regression testing against protected expert behavior. Third, we make patching practical for online RL by placing patches in compact MoE bottleneck space and evaluating only a retrieved top-K subset during reward computation. Fourth, we evaluate the system in MuJoCo using MoE-GUIDE under a stricter downstream protocol, including HalfCheetah, a prior failure case. In the reported experiments the verifier is deliberately simple: a state is outside the expert envelope when any coordinate falls outside the minimum or maximum value observed in the expert data. This is not a general OOD detector or a solution to reward hacking; it is a transparent test of whether even a weak reward-free signal can affect downstream learning when compiled into the reward mapping. More broadly, the same interface can use human review, hand-written rules, simulator diagnostics, perception models, or other monitors to mark regions where reward should be suppressed.

2 Related Work

The closest technical precedent for our method is lifelong model editing. GRACE maintains a frozen base model and attaches local corrections in representation space: incoming examples are matched against stored keys, and the corresponding edit changes the model output without updating the original weights Hartvigsen et al. (2023). Subsequent editing work has explored related mechanisms for preserving unrelated behaviour, including null-space-constrained edits that project updates away from preserved knowledge FANG et al.; Liu et al. (2026), dynamic or side-memory approaches to lifelong editing Wang et al. (2024), and patcher-style methods that associate localized errors with compact corrective modules Huang et al. (2023). These methods motivate the abstraction used here: deployed models can be maintained by accumulating local feature-space corrections rather than by retraining from scratch.

Our setting differs in the object being edited. Prior model-editing methods patch predictive behaviour; we patch a reward surface that is optimized by a downstream RL policy. This changes the failure mode. In standard model editing, an unpatched error causes a bad prediction. In RL, an over-

generalized reward region can become an attractor because policy optimization actively searches for high-scoring behaviours. This connects our setting to reward misspecification, reward hacking, and reward-model overoptimization Amodei et al. (2016); Pan et al. (2022); Gao et al. (2023). Our patches therefore do not replace a prediction with a new label. They suppress localized regions of the deployed reward mapping so that the learned intrinsic reward no longer incentivizes behaviours identified as overgeneralized, unwanted, or constraint-violating.

Flywheel-guided patching is also distinct from hard safety mechanisms such as constrained policy optimization and shielding Achiam et al. (2017); Alshiekh et al. (2018). Those methods enforce constraints during policy learning or action selection. Our method instead edits the reward signal consumed by policy optimization. It is a soft correction mechanism, not a formal safety shield. Its purpose is to make reward-free task specifications maintainable: demonstrations, safety events, social constraints, human review, or verifier outputs need only identify where the learned reward should change, rather than define a complete replacement reward function.

Finally, the method differs from runtime guardrails. A runtime guardrail must evaluate the relevant rule, perception model, simulator check, or human-supplied criterion whenever the policy acts. Flywheel-guided patching uses such signals offline during patch construction and deploys only the compiled suppression inside the reward mapping. This separation is important when the verifier signal is too slow, expensive, or human-dependent to run at control frequency.

3 Interactionless Inverse Reinforcement Artifacts

The reward artifact patched in this paper comes from the MoE-GUIDE lineage. MoE-GUIDE was the first concrete instantiation used here: it trained a mixture-of-experts reconstruction model from state-only expert observations and then used an explicit mapping function to convert reconstruction behavior into an intrinsic reward Malomgré & Simoens (2025). Interactionless Inverse Reinforcement (IIRL) later widened this construction into a broader family of methods in which reward artifacts are constructed from offline structure rather than online reward interaction Malomgré & Simoens (2026b). We use the IIRL terminology for the general class, while MoE-GUIDE supplies the concrete artifact evaluated in this paper.

The important property inherited from MoE-GUIDE is the separation between a learned evaluator and a reward mapping. We write the base reward as

$$r_0(s) = g(E_\theta(s), s),$$

where E_θ is the trained evaluator and g is the explicit mapping that turns the evaluator's outputs into a scalar intrinsic reward. In MoE-GUIDE, E_θ is a mixture-of-experts autoencoder and g maps expert assignment, bottleneck structure, and reconstruction consistency into reward. This separation is what makes post hoc repair possible: the learned evaluator can remain fixed while the deployed reward mapping is edited.

Flywheel patching operates on this mapping function. When a verifier identifies a state or region that should not continue receiving high reward, we do not retrain E_θ. Instead, the correction is compiled into a patched mapping

$$g_{\mathcal{P}},$$

where $\mathcal{P}$ is the set of accepted flywheel patches. The patched reward is

$$r_{\mathcal{P}}(s) = g_{\mathcal{P}}(E_\theta(s), s).$$

Thus, the object being maintained is not the downstream policy and not the learned evaluator, but the explicit reward mapping exposed to reinforcement learning.

The failure mode studied here is reward-model overgeneralization. A learned evaluator may assign high intrinsic reward to states outside the intended expert-support region, especially once a downstream policy optimizer actively searches for high-reward behavior. In the MoE-GUIDE/IIRL

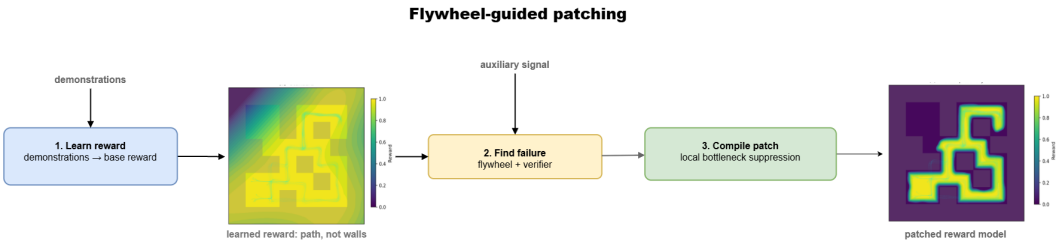


Figure 1: Flywheel-guided patching. Demonstrations train a base IIRL reward artifact. Reward-free auxiliary information is then used by the Alignment Flywheel to identify where the learned reward overgeneralizes. Verified failures are compiled into local bottleneck-space suppression patches, producing a patched reward mapping without retraining the learned evaluator.

artifact structure, this failure can be addressed by modifying g rather than relearning E_θ. Flywheel patching uses this editability: reward-free verifier signals identify regions where reward should be removed, and the patched mapping suppresses those regions while preserving the original learned evaluator.

4 Extending the Alignment Flywheel to MuJoCo

The Alignment Flywheel supplies the decision procedure behind flywheel patching: it searches for reward-model failures, verifies them with an external reward-free signal, refines accepted failures into corrections, and commits only corrections that preserve protected behavior Malomgré & Simoens (2026a). In this paper we extend that abstraction to MuJoCo continuous control by adding the components needed to operate on downstream locomotion rollouts and MoE-GUIDE reward artifacts: locomotion candidate generation, a reward-free expert-envelope verifier, a MoE-GUIDE oracle adapter, bottleneck-space patch synthesis, and regression testing against protected expert behavior. The flywheel protocol is kept fixed across the reported experiments; we do not redesign the search strategy, verifier, refinement procedure, or oracle update separately for each environment. Full implementation details are given in Appendix C.

The expert-envelope verifier is intentionally simple. Given an expert dataset $\mathcal{D}_E$, we compute coordinate-wise bounds

$$\ell_d = \min_{s \in \mathcal{D}_E} s_d, \qquad u_d = \max_{s \in \mathcal{D}_E} s_d,$$

and mark a rollout state s as outside demonstrated support when some coordinate satisfies $s_d < \ell_d$ or $s_d > u_d$. This is not a general OOD detector or a solution to reward hacking; it is a transparent reward-free signal used to test whether even weak external judgments can affect downstream learning when compiled into the reward mapping. The same verifier interface can support richer signals, including human review, hand-written rules, simulator diagnostics, perception models, or other monitors that mark regions where reward should be suppressed.

5 Flywheel-Guided Suppression Patching

Figure 1 shows the mechanism studied in this paper. A base IIRL reward is first learned from demonstrations. The Alignment Flywheel then uses reward-free auxiliary information to find where the learned reward overgeneralizes. Finally, the verified failure is compiled into a local suppression patch inside the reward-mapping function. The result is a patched reward artifact whose deployed reward surface has changed, even though the learned evaluator itself has not been retrained.

The central intervention is to extend the explicit reward-mapping function g. In the base artifact, a fixed learned evaluator E_θ produces reward-relevant features and g maps them to an intrinsic reward,

$$r_0(s) = g(E_\theta(s), s).$$

Given a flywheel-accepted patch set $\mathcal{P}$, we deploy instead

$$r_{\mathcal{P}}(s) = g_{\mathcal{P}}(E_\theta(s), s) = \mathrm{clip}_{[0,r_{\max}]}\left(r_0(s) - \lambda S_{\mathcal{P}}(s)\right).$$

Thus, patching does not update the parameters of E_θ. It updates the reward artifact by replacing g with $g_{\mathcal{P}}$, where $S_{\mathcal{P}}$ is a flywheel-constructed suppression field. This is the sense in which reward-free information updates the reward model: constraints, support checks, perception signals, or other auxiliary diagnostics are not used as rewards themselves, but are compiled into the mapping that produces reward.

5.1 Reward-Free Failure Signals

The Alignment Flywheel supplies the patch targets. It treats the current reward mapping as the object under audit and uses auxiliary information that is not itself a scalar reward to identify candidate failures. In the MuJoCo experiments, the auxiliary signal is an expert-envelope verifier derived from the demonstration dataset. Let $\mathcal{D}_E$ be the expert state dataset and define coordinate-wise bounds

$$\ell_d = \min_{s\in\mathcal{D}_E} s_d, \qquad u_d = \max_{s\in\mathcal{D}_E} s_d.$$

A rollout state is marked outside demonstrated support when

$$\mathrm{OOS}(s) = \mathbf{1}\left[\exists d:\ s_d < \ell_d \ \vee\ s_d > u_d\right].$$

When such a state also receives high intrinsic reward, the flywheel treats it as a reward-overgeneralization candidate: the artifact is assigning reward outside the support envelope of the demonstrations.

This verifier is intentionally simple and dataset-relative. It does not claim that every outside-envelope state is unsafe, only that high reward outside the demonstrated support is the overgeneralization signal studied here. The same interface can accept slower reward-free signals, such as perception checks, simulation-based diagnostics, or human review. The key point is that these signals are used offline to decide where g should be patched; they are not evaluated as runtime rewards.

5.2 Bottleneck-Space Suppression

Once the flywheel accepts a reward-overgeneralization candidate, it is compiled into a local suppression patch. In the MoE-GUIDE instantiation, each expert e maps a state s to a bottleneck code $z_e(s)$ and receives a gating weight $\alpha_e(s)$. We use this bottleneck space as the patch coordinate system, because it is the representation in which the reward artifact already measures expert similarity.

A patch is a record

$$p_j = (e_j, c_j, \sigma_j, \eta_j, \tau_j, m_j),$$

where e_j is the associated expert, c_j is the bottleneck-space center, σ_j is the kernel bandwidth, η_j is the suppression strength, τ_j records the patch type, and m_j stores provenance metadata from the flywheel. The patch center is placed at the bottleneck code of the verified failure state, typically under the highest-weight expert.

Let

$$\mathcal{P}_e = \{p_j \in \mathcal{P} : e_j = e\}$$

denote the accepted patches assigned to expert e. The exact suppression field defined by the patch set is

$$S_{\mathcal{P}}(s) = \min\left(1,\ \sum_{e\in\mathcal{E}} \alpha_e(s) \sum_{p_j\in\mathcal{P}_e} \eta_j \exp\left(-\frac{\|z_e(s) - c_j\|^2}{2\sigma_j^2}\right)\right),$$

where $\mathcal{E}$ is the set of MoE experts. The outer minimum bounds total suppression, preventing overlapping patches from subtracting unbounded reward.

The patched reward is then

$$r_{\mathcal{P}}(s) = g_{\mathcal{P}}(E_\theta(s), s) = \mathrm{clip}_{[0,r_{\max}]}\left(r_0(s) - \lambda S_{\mathcal{P}}(s)\right),$$

where $r_0(s) = g(E_\theta(s), s)$ and λ controls the scale of suppression. Thus, patches do not modify the learned evaluator E_θ; they modify the deployed reward mapping by subtracting localized suppression in bottleneck space.

Before commitment, each candidate patch is checked against a protected demonstration basin. Patches that reduce reward on protected expert states are rejected or narrowed. Thus, the verifier identifies where the reward should be reduced, while the regression check limits damage to demonstrated behavior.

5.3 Scalable Online Evaluation

The exact suppression field in Section 5.2 sums over all accepted patches. As the patch set grows over flywheel iterations, a naive implementation would make each reward query scale with the total number of committed corrections. We therefore evaluate suppression with local retrieval in bottleneck space.

For each expert e, the patch centers in $\mathcal{P}_e$ are stored in a locality-sensitive hash index over that expert's bottleneck codes. At reward evaluation time, the current code $z_e(s)$ is hashed using the same projections, and only patches in the matching or neighboring buckets are considered. Let

$$\mathcal{C}_e(s) = \mathrm{Retrieve}(z_e(s), \mathcal{P}_e)$$

be the retrieved candidate set for expert e. The deployed approximation to the suppression field is

$$\widehat{S}_{\mathcal{P}}(s) = \min\left(1, \sum_{e\in\mathcal{E}} \alpha_e(s) \sum_{p_j\in\mathcal{C}_e(s)} \eta_j \exp\left(-\frac{\|z_e(s) - c_j\|^2}{2\sigma_j^2}\right)\right).$$

The reward used by the downstream policy optimizer is therefore

$$r_{\mathcal{P}}(s) = \mathrm{clip}_{[0,r_{\max}]}\left(r_0(s) - \lambda \widehat{S}_{\mathcal{P}}(s)\right).$$

This changes the online cost from scaling with the full patch set, $O(\sum_e |\mathcal{P}_e|)$, to scaling with the number of retrieved local candidates, $O(|\mathcal{E}|C)$, where C is the average number of candidates retrieved per expert. In the reported experiments we evaluate all experts, because the MoE artifacts have few experts. For larger reward artifacts, the same retrieval scheme can be combined with top-K expert pruning by evaluating only the experts with the largest gating weights. This reduces the cost to $O(K_E C)$, but this additional pruning is a scaling extension rather than part of the reported experimental protocol.

This retrieval step is what makes patching usable inside the RL loop. The Alignment Flywheel may accumulate many accepted corrections, but the policy optimizer should not pay a linear cost in the full patch set at every environment step. Hashing keeps reward evaluation tied to local bottleneck neighborhoods rather than global patch count.

This also explains why patching is useful even when the verifier used in our MuJoCo experiments is cheap. The expert-envelope verifier is only a coordinate-wise dataset-support check, so it could be evaluated directly at control frequency. However, the same interface can use slower reward-free signals: perception models, simulation-based diagnostics, formal checkers, or human review can be run offline by the flywheel to decide where the reward artifact fails. Deployment then pays only the cost of local bottleneck lookup, not the cost of the original verifier. The experiments in this paper validate the mechanism in the cheap locomotion setting; slow verifier pipelines motivate the design and are left to future work.

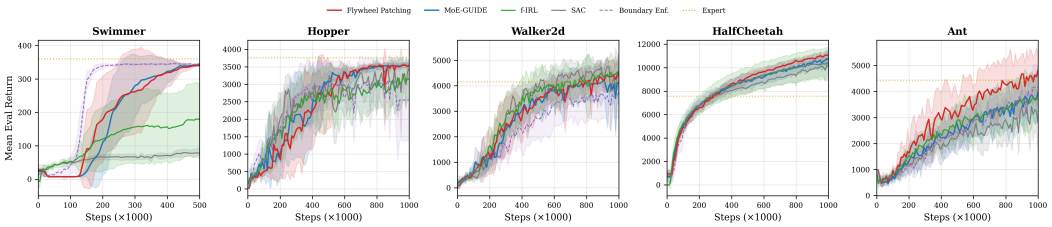


Figure 2: Learning curves for the main downstream evaluation in Table 1. Curves show mean undiscounted evaluation return across 5 seeds, with shaded bands for one standard deviation.

5.4 Downstream Policy Optimization

After patch construction, the downstream agent interacts with the environment using the patched reward mapping $g_{\mathcal{P}}$. At each environment step, the fixed reward artifact evaluates the current state, the suppression field is computed from the retrieved local patches, and the policy optimizer receives

$$r_{\mathcal{P}}(s) = g_{\mathcal{P}}(E_\theta(s), s).$$

The patching layer is therefore inside the reward computation used by the agent, not a post hoc analysis of completed trajectories.

We use SAC for downstream policy optimization. The reward model is trained from expert demonstrations, but the policy is not initialized from expert states and does not use an expert-state reset curriculum. Rollouts begin from the standard environment initial-state distribution. This is the key protocol change from the earlier MoE-GUIDE downstream evaluation: the policy must learn under the deployed reward surface rather than being kept near the demonstration support by privileged resets.

This stricter protocol makes reward-model overgeneralization observable. If the base reward assigns high value outside the expert envelope, the policy can discover and exploit those regions during ordinary rollouts. Flywheel-guided patching is evaluated as a correction mechanism for this setting: reward-free failure signals identify overgeneralized states, accepted patches extend g into $g_{\mathcal{P}}$, and SAC then optimizes against the patched reward. Runtime out-of-bounds enforcement, when included, is treated separately as a baseline that checks the verifier directly at each timestep rather than compiling its verdict into the reward mapping.

6 Results

Table 1 reports the main downstream evaluation. The metric is undiscounted evaluation return. Each method is run with 5 seeds; every checkpoint is evaluated every 10K environment steps using 10 evaluation episodes per seed. Table entries report the mean $\pm$ standard deviation across the seed-level evaluation returns at the reported checkpoint: 500K steps for Swimmer and 1M steps for all other environments. The same aggregation is used for appendix learning experiments unless stated otherwise. Each main-text and appendix downstream-learning table is accompanied by the corresponding learning curve, so the tables should be read as summaries of full training trajectories rather than isolated terminal measurements.

The baselines are chosen to cover distinct alternatives to flywheel patching. SAC tests whether the environment reward alone is sufficient. ER+RND and ER+Disagreement test whether standard exploration bonuses explain the gains. f-IRL is the closest adversarial IRL baseline and receives environment reward during training. Boundary Enforcement tests direct runtime enforcement of a formal support rule. MoE-GUIDE is the unpatched IIRL reward artifact. Flywheel Patching is our method: the same learned evaluator is kept fixed, while reward-free failure information is compiled into the deployed mapping $g_{\mathcal{P}}$.

Table 1: Final undiscounted evaluation return ± std across 5 seeds. Swimmer is evaluated at 500K steps; all others at 1M steps. Bold indicates the best method per environment. ‡Imperfect expert demonstrations.

Method	Swimmer	Hopper	Walker2d‡	HalfCheetah‡	Ant‡
Expert	360	3,761	4,150	7,582	4,443
SAC	78 ± 15	3459 ± 127	4353 ± 916	10125 ± 657	2807 ± 1149
f-IRL	329 ± 17	3464 ± 91	$\mathbf{4580 \pm 474}$	10996 ± 496	3454 ± 663
ER+RND	98 ± 29	3181 ± 330	4450 ± 778	10492 ± 529	4018 ± 771
ER+Disagreement	71 ± 11	3240 ± 261	4445 ± 466	10359 ± 282	3812 ± 866
Boundary Enf.	$\mathbf{352 \pm 2}$	3349 ± 109	3052 ± 1085	9711 ± 626	2687 ± 1241
Flywheel Patching	341 ± 3	$\mathbf{3544 \pm 76}$	4535 ± 722	$\mathbf{11091 \pm 307}$	$\mathbf{4816 \pm 594}$
MoE-GUIDE	342 ± 2	3540 ± 86	4146 ± 877	10646 ± 657	3974 ± 1099

Flywheel Patching achieves the highest final return in four of five environments and improves over the unpatched MoE-GUIDE reward in Hopper, Walker2d, HalfCheetah, and Ant, while essentially preserving performance in Swimmer. The strongest gains appear in the imperfect-demonstration environments, where reward-model overgeneralization is most likely to matter. In Ant, the highest-dimensional task, Flywheel Patching improves over MoE-GUIDE by 842 return points and exceeds the imperfect expert. In HalfCheetah, a prior MoE-GUIDE failure case attributed to overgeneralization, the patched reward reaches the best final return among all methods.

The comparison also clarifies what the result is not. The gain is not explained by generic exploration: ER+RND and ER+Disagreement provide only marginal benefit over SAC. It is not simply an IRL effect: f-IRL is competitive and wins Walker2d, but Flywheel Patching is stronger on HalfCheetah and Ant while modifying only the deployed reward mapping. It is not merely formal rule enforcement: Boundary Enforcement wins Swimmer, but it depends on checking the support rule at every timestep and performs worse in the remaining environments. Flywheel Patching instead compiles reward-free support information into local reward edits, avoiding runtime dependence on the original verifier.

The remaining experiments are summarized in Table 2 and reported fully in the appendix. They test whether the mechanism behaves as intended across patch construction, online evaluation, representation choice, and constraint-targeted variants. The appendix now includes learning curves for all downstream-learning ablations, including the OOB-margin study. These results also expose the intended scope of the method: compiled suppression is a soft reward edit, not a hard safety shield, and direct runtime enforcement remains stronger when a formal rule must be checked at every timestep.

Taken together, the results support the intended claim: reward-free information can update a learned IIRL reward artifact by extending the explicit mapping g, and the resulting patched reward can improve downstream RL without retraining the evaluator. The auxiliary experiments qualify this claim: the method is a scalable reward-editing mechanism, not a hard safety shield.

7 Discussion and Conclusion

Flywheel patching treats reward repair as an edit to an explicit reward mapping, rather than as a full retraining problem. The learned evaluator E_θ remains fixed, while the deployed mapping g is replaced by $g_{\mathcal{P}}$, which subtracts local bottleneck-space suppression in regions accepted by the Alignment Flywheel. In the MuJoCo experiments, this mechanism improves or preserves downstream performance relative to unpatched MoE-GUIDE under a stricter protocol than the earlier MoE-GUIDE evaluation: the reward artifact is trained from demonstrations, but SAC starts from the standard environment reset distribution rather than from expert-state warm starts. The HalfCheetah result is particularly relevant because it was a prior failure case for downstream MoE-GUIDE and is improved by editing the deployed reward mapping rather than relearning the evaluator.

Table 2: Auxiliary findings. Detailed tables, learning curves for downstream learning ablations, and implementation measurements are reported in Appendices D–H.

Experiment	Finding
Patch budget sweep	Changing the patch cap from 200 to 1000 does not yield statistically significant return differences; point estimates are positive in Ant and HalfCheetah and mixed in Hopper and Walker2d.
OOB margin ablation	Relaxing the expert-envelope boundary changes patch aggressiveness. Zero-margin patching can over-suppress near-boundary behavior in Swimmer, while a moderate margin gives the strongest Ant return among the FK variants.
Tightness and patch identification	Performance is mostly robust to constraint tightness. Regression-based patch identification is the safer default; DBSCAN is comparable in several tasks but hurts Swimmer.
Bottleneck locality and dimension	Bottleneck-space coverage avoids the curse-of-dimensionality behavior of raw state-space patches. The Ant dimension sweep shows that moderate bottleneck sizes preserve reconstruction quality without inflating patch counts.
Hashed retrieval	Patch evaluation scales with retrieved local candidates rather than with the full accepted patch set, enabling patch lookup inside the RL reward loop.
Constraint-targeted Ant	Constraint-targeted patches reduce violations when evaluated as intrinsic suppression, but are weakened when combined with external task reward, showing the limit of soft reward edits.
Runtime OOB enforcement	Direct OOB enforcement gives stronger violation reduction in Ant, but it requires checking the rule online instead of compiling its verdict into $g_{\mathcal{P}}$.
Slow verifier motivation	Latency measurements show why offline compilation matters for human review, perception models, and other signals that are too slow to evaluate at control frequency.

The broader implication is that reward-free information can be used as an interface for maintaining learned reward artifacts. A human reviewer, support estimator, simulator diagnostic, perception model, or hand-written rule need not specify a complete scalar reward. It only needs to identify regions where the current learned reward should not remain high. The flywheel then converts that judgment into a local patch, regression-tests it against protected expert behavior, and deploys the compiled correction inside $g_{\mathcal{P}}$. This is most useful when the diagnostic signal is non-scalar, expensive, or too slow to run at control frequency, because the online policy optimizer only evaluates the compiled bottleneck-space suppression.

The scope of the claim is limited. The expert-envelope verifier used in the main experiments is a weak, dataset-relative signal: high reward outside the coordinate-wise demonstration envelope is treated as reward-model overgeneralization, not as a semantic proof of unsafe behavior. The patches are also soft reward edits, not hard shields. Direct runtime enforcement remains preferable when a formal constraint must be guaranteed, and strong task rewards can weaken a compiled suppression if the violating behavior remains externally profitable. The auxiliary Ant results therefore mark an important boundary: flywheel patching can reduce incentives created by the learned reward artifact, but it should not be presented as a substitute for formal constraint enforcement when guarantees are required.

Future work should replace the deliberately simple envelope verifier with richer reward-free signals, including human-in-the-loop review, learned support models, perception-based checks, simulator diagnostics, and task-specific monitors. It should also study patch interactions over longer maintenance horizons, automatic selection of patch bandwidth and strength, and applications beyond MoE-GUIDE, including intrinsic motivation objectives Bellemare et al. (2016); Burda et al. (2018);

Aubret et al. (2023) and manually specified Icarte et al. (2018; 2022) or learned Baert et al.; Shehab et al. (2026) reward machines.

Overall, flywheel patching establishes a practical maintenance layer for IIRL reward artifacts: use an Alignment Flywheel to surface reward overgeneralization or constraint violations, compile accepted findings into local suppressions of an editable reward mapping, and deploy the patched mapping during downstream RL without retraining the learned evaluator.

Acknowledgment

This research was supported by funding from the Flemish Government under the "Onderzoeksprogramma Artificiele Intelligentie (AI) Vlaanderen" program.

A Reward Artifact and Training Protocol

This appendix documents the reward artifact, downstream training protocol, baselines, patch-construction details, scaling measurements, and full auxiliary analyses supporting the main results. The main paper keeps the mechanism compact; the appendix provides the complete reproducibility details. Unless stated otherwise, training-result tables report mean evaluation return $\pm$ standard deviation across seeds using the same checkpoint evaluation protocol as the main text, and each tabulated training comparison is accompanied by its learning curve.

A.1 MoE Reward Model Architecture

Each expert in the Mixture of Gated Autoencoders is a single-hidden-layer autoencoder: a linear encoder ($d_{\text{obs}} \to d_{\text{bottleneck}}$) with ReLU activation, followed by a linear decoder ($d_{\text{bottleneck}} \to d_{\text{obs}}$). The gating network is a linear layer with softmax output over experts. Table 3 lists the per-environment model dimensions. All models are pre-trained on N_{traj} expert demonstration trajectories.

Table 3: MoE reward model architecture per environment.

Environment	d_{obs}	$d_{\text{bottleneck}}$	Experts	N_{traj}	Expert type
Swimmer-v2	8	3	1	5	Perfect
Hopper-v2	11	4	2	5	Perfect
Walker2d-v2	17	7	4	5	Imperfect
HalfCheetah-v2	17	7	4	5	Imperfect
Ant-v2	111	10	3	5	Imperfect

For expert e, the encoder and decoder are

$$\mathbf{z}_e = \mathrm{ReLU}(W_e^{\text{enc}}\,\mathbf{x} + \mathbf{b}_e^{\text{enc}}), \qquad W_e^{\text{enc}} \in \mathbb{R}^{B\times D},\ \mathbf{b}_e^{\text{enc}} \in \mathbb{R}^{B}, \tag{1}$$

$$\hat{\mathbf{x}}_e = W_e^{\text{dec}}\,\mathbf{z}_e + \mathbf{b}_e^{\text{dec}}, \qquad W_e^{\text{dec}} \in \mathbb{R}^{D\times B},\ \mathbf{b}_e^{\text{dec}} \in \mathbb{R}^{D}. \tag{2}$$

The gating network is a single linear layer followed by softmax,

$$\alpha_e(\mathbf{x}) = \frac{\exp(w_e^\top \mathbf{x} + b_e^g)}{\sum_{j=1}^{E} \exp(w_j^\top \mathbf{x} + b_j^g)}, \qquad w_e \in \mathbb{R}^D,\ b_e^g \in \mathbb{R}. \tag{3}$$

The gating-weighted reconstruction is $\hat{\mathbf{x}} = \sum_{e=1}^{E} \alpha_e(\mathbf{x})\,\hat{\mathbf{x}}_e$.

A.2 Intrinsic Reward Mapping

The intrinsic reward is computed by an exponential mapping from per-observation reconstruction loss to $[0, 1]$:

$$g(L) = \mathrm{clip}\left(\exp\left(-s \cdot \frac{L - L_{\min}}{L_{\max} - L_{\min}}\right),\ 0,\ 1\right), \tag{4}$$

where L is the per-observation MSE reconstruction loss. In downstream training, the total reward is $r = r_{\text{ext}} + \kappa_t \cdot g(L)$ with $\kappa_t = \kappa_{t-1} \cdot \kappa_{\text{decay}}$ applied at each SAC update step. Table 4 lists the per-environment mapping parameters used in the main results.

Table 4: Intrinsic reward mapping parameters per environment. These are the best hyperparameters selected via the L_{max} sweep.

Environment	L_{max}	L_{min}	Steepness s	κ	κ_{decay}	Steps
Swimmer-v2	0.2	0.11	25	1	1.0	500K
Hopper-v2	0.15	0.13	25	2	1.0	1M
Walker2d-v2	0.6	0.07	100	2	0.9999995	1M
HalfCheetah-v2	1.0	0.5	100	1	0.999995	1M
Ant-v2	0.1	0.003	100	1	0.9999995	1M

A.3 SAC and Replay Buffer Settings

All environments use the same SAC configuration, listed in Table 5. Swimmer uses a 100K-transition replay buffer; all other environments use 1M. Transitions store the combined reward $r_{\text{total}} = r_{\text{ext}} + \kappa_t \cdot g(L)$ at insertion time.

Table 5: SAC hyperparameters, shared across all environments.

Parameter	Value
Actor/Critic hidden layers	$[256, 256]$
Actor/Critic activation	ReLU
Learning rate (all networks)	3×10^{-4}
Discount factor γ	0.99
Polyak averaging τ	0.995
Automatic entropy tuning	Yes
Target entropy	$-d_{\text{act}}$
Batch size	256
Random exploration steps	10,000
Evaluation interval	10,000 steps
Evaluation episodes per checkpoint	10
Number of seeds	5

B Baseline Implementations

B.1 FK Patching Variant Used in the Main Table

The FK Patching row in Table 1 uses the best-performing FK variant per environment, selected from the tightness and patch-identification ablations (Tables 14 and 15). Table 6 lists the specific variant used in each environment. HalfCheetah and Ant use the original FK kernel, because the unmodified flywheel patching achieved the highest return in those environments.

B.2 Exploration Baselines: ER+RND and ER+Disagreement

Both exploration baselines use the SAC configuration in Table 5 with an additive intrinsic reward $r = r_{\text{ext}} + \lambda r_{\text{int}}$, where r_{int} is recomputed at sample time using the current module state.

RND. Random Network Distillation uses a fixed random target network and a trainable predictor, both $d_{\text{obs}} \to 64 \to 64 \to 64$ MLPs with ReLU activations and LayerNorm on the output. The intrinsic reward is $r_{\text{int}} = \frac{1}{d} \sum_i (f_\theta(s) - f_{\bar{\theta}}(s))_i^2$. The predictor is updated with Adam (lr $= 10^{-3}$) at every timestep.

Table 6: FK Patching variant used per environment in Table 1.

Environment	Tightness	Patch method
Swimmer-v2	Loose	Regression
Hopper-v2	Tight	Regression
Walker2d-v2	Tight	Regression
HalfCheetah-v2	(original)	(original)
Ant-v2	(original)	(original)

Disagreement. Ensemble Disagreement uses an ensemble of 5 forward dynamics models, each $(d_{\text{obs}} + d_{\text{act}}) \to 256 \to 256 \to d_{\text{obs}}$ with ReLU activations. The intrinsic reward is the variance of next-state predictions across ensemble members, averaged over observation dimensions. The ensemble is trained with Adam (lr $= 10^{-3}$) at every timestep.

Scale sweep. Both methods are evaluated with $\lambda \in \{0.001, 0.01, 0.1\}$ for each environment. The scale yielding the highest mean return at evaluation time, 500K for Swimmer and 1M for all other environments, is selected for Table 1. All runs use 2M total timesteps, a 1M replay buffer, 10K random exploration steps, and 5 seeds.

B.3 f-IRL Baseline

We use the original f-IRL implementation with the reverse KL divergence objective. f-IRL alternates between training a discriminator on expert versus agent state distributions, optimizing a reward function with the f-divergence loss, and running SAC with the learned reward.

Reward network. The reward network is an MLP with ReLU activations and output clamped to $[-10, 10]$. Hidden sizes are $[64, 64]$ for all environments except Ant, which uses $[128, 128]$. It is trained with Adam (lr $= 10^{-4}$, weight decay $= 10^{-3}$), one gradient step per IRL iteration.

Discriminator. The discriminator is ResNet-style with tanh activations: 3 residual blocks ($d_{\text{hid}} = 128$) for all environments except Ant, which uses 6 blocks. It is trained with Adam (lr $= 3 \times 10^{-4}$, weight decay $= 10^{-4}$) for 1,200 steps per IRL iteration with batch size 800.

SAC integration and scale sweep. f-IRL uses the same SAC architecture and hyperparameters as MoE-GUIDE. At each SAC update, the reward is $r = r_{\text{ext}} + \lambda r_{f\text{-IRL}}(s)$. Each environment is swept over $\lambda \in \{0.01, 0.1, 1.0\}$ with 5 seeds. The best scale per environment is: Swimmer $= 1.0$, Hopper $= 0.1$, Walker2d $= 0.1$, HalfCheetah $= 0.01$, Ant $= 0.01$. Swimmer uses 500K steps; all other environments use 1M steps, matching the MoE-GUIDE training budget.

B.4 Boundary and OOB Enforcement Baselines

Out-of-bounds enforcement checks active observation dimensions against the demonstrated $[\min, \max]$ range. Hard mode clips intrinsic reward to 0 when any dimension is out of bounds. Smooth mode applies a Gaussian decay penalty. Optional safety functions add constraint-specific checks. Full OOB and constraint-targeted results are reported in Section G.

C MoE-GUIDE System Details

This section provides the full specification of the MoE-GUIDE pipeline: the Flywheel governance engine, the OODA role decomposition, and the implementations used for MuJoCo locomotion environments.

C.1 Alignment Flywheel Architecture

The Alignment Flywheel is an architecture-agnostic governance loop that wraps any AI system, the Oracle, to continuously discover violations, verify them, and deploy targeted corrections without retraining. The engine decomposes corrective action into four roles, each following the OODA pattern: Observe, Orient, Decide, and Act.

1. **Red Team:** searches for flaws in the current reward surface. *Observe:* sample candidate observations from the demonstration data. *Orient:* rank by distance to the safe basin. *Decide:* select the farthest-N or highest-combined candidates. *Act:* submit as verification requests.
2. **Verifier:** confirms whether each candidate is a true violation. *Observe:* load norms from the flywheel overlay. *Orient:* match the candidate to norm constraints, such as spatial bounds, velocity limits, or domain-specific rules. *Decide:* classify as VIOLATION, SAFE, or AMBIGUOUS. *Act:* emit a `VerificationResult` with severity score ρ.
3. **Triage:** prioritises verified violations by severity before refinement, using FIFO or a priority heap.
4. **Refinement:** plans and deploys corrections. *Observe:* dequeue verified violations. *Orient:* compute patch bandwidth and strength from severity. *Decide:* regression-test the candidate patch against the demonstration basin. *Act:* deploy a `GovernanceBatch` to the Oracle.

Additionally, a **Blue Team** monitors for collateral damage, defined as reward loss on safe observations, and an **Enforcement** layer applies optional post-decision filters.

Governance Engine. The `GovernanceEngine` orchestrates the iterative loop:

1. Initialise the Oracle with the MoE model and the demonstration basin.
2. **Repeat:**
   (a) Red Team discovers flaws on the current patched reward surface.
   (b) Verifier confirms violations and computes severities.
   (c) Triage prioritises the violation queue.
   (d) Refinement proposes, regression-tests, and commits patches.
   (e) Oracle updates the kernel with accepted patches.
   (f) Early stopping: if the flaw count has not improved for p consecutive iterations, with default $p = 3$, terminate.
3. Export the accumulated kernel.

All decisions are recorded in an append-only **Knowledge Base**, providing an audit trail from violation discovery to patch deployment.

Factory and Registry. Each OODA step is a pluggable component registered by name in a central `FactoryRegistry`. Over 40 implementations are auto-registered at import time, covering grid observers, locomotion observers, medical case generators, adaptive bandwidth orienters, bottleneck regression deciders, cluster deciders, and more. This allows the same governance engine to drive spatial, locomotion, medical, and custom domains by swapping YAML configurations.

C.2 Role Implementations for Locomotion

Table 7 lists the OODA step implementations used in all MuJoCo locomotion experiments. Every component listed below is new and was developed for MoE-GUIDE; none existed in the base Flywheel framework.

C.3 Red Team: Flaw Discovery

The Red Team generates candidate observations that are simultaneously *out-of-bounds* (OOB) on at least one dimension group and *highly rewarded* by the oracle, the most dangerous class of model error.

Table 7: OODA step implementations for locomotion environments. All components are new contributions.

Role	Observe	Orient
Red Team	LocomotionObserver	LocomotionOrienter
Verifier	LocomotionNormLoader	LocomotionNormMatcher
Refinement	LocomotionQueueObs.	BottleneckBWOrienter

Role	Decide	Act
Red Team	HighestCombinedDecider	LocomotionCandidateSubmitter
Verifier	LocomotionViolationDecider	VerificationEmitter
Refinement	BottleneckRegressionDecider	LocomotionBatchDeployer (alt: BottleneckClusterDecider)

Observe, `LocomotionObserver`. Generates candidate observations through four parallel sampling strategies:

1. **Gaussian perturbation** over 5 noise scales. For each scale $\sigma_m \in \{0.25, 0.5, 0.75, 1.0, 1.5\} \times \boldsymbol{\sigma}_{\text{demo}}$, draw approximately 2,000 samples by adding $\mathcal{N}(0, \sigma_m)$ noise to randomly selected demonstration observations.
2. **Boundary probing** with approximately 1,000 points. Push single dimensions 1–10% beyond the demonstration range to probe the reward surface at the boundary of the safe region.
3. **Correlated group perturbation.** Scale entire semantic groups, such as position or velocity, by factors $\{1.1, 1.2, 1.5\}$ to test for joint dimensional violations.
4. **Adaptive re-sampling around prior flaws.** If the previous iteration found flaws, draw $5\times$ samples from the last 500 flaws with 0.3σ noise, exploring neighbourhoods of known problem regions.

All candidates are scored by the oracle; the full set of points and their safety values is passed to the Orient step.

Orient, `LocomotionOrienter`. Identifies flaws: observations that are both OOB and highly rewarded. For each dimension group g, such as position or velocity,

$$\rho_g(\mathbf{x}) = \max_d \max\Big(\frac{\text{lo}_d - x_d}{\text{hi}_d - \text{lo}_d},\ \frac{x_d - \text{hi}_d}{\text{hi}_d - \text{lo}_d},\ 0\Big), \tag{5}$$

where lo_d and hi_d are the per-dimension effective bounds. A point is flagged as a flaw if $\max_g \rho_g > 0$ and $r(\mathbf{x}) > \tau_{\text{floor}}$.

Decide, `HighestCombinedDecider`. Sorts flaws by oracle safety value in descending order. Observations with the highest safety scores are prioritised because they represent the most confident model errors.

Act, `LocomotionCandidateSubmitter`. Emits one `CandidateFlaw` artifact per flaw, carrying the observation vector, violated group, OOB ratio, safety value, and an uncertainty estimate $u = \min(0.9,\ 0.2 + \rho)$ that scales with violation severity.

C.4 Verifier: Violation Confirmation

The Verifier confirms that each candidate flaw genuinely violates a domain norm before it proceeds to refinement.

Observe, `LocomotionNormLoader`. Loads the candidate and the set of active norms, including spatial boundary norms, velocity norms, or safety constraint norms, from the flywheel overlay.

Orient, `LocomotionNormMatcher`. Matches the candidate to applicable norms. For spatial boundary norms, the matcher checks whether $\rho > 0$ in the candidate's context. For safety constraint norms, the matcher evaluates the explicit constraint function, such as velocity cap, torso tilt, or hip angular velocity. It returns matched (norm, evidence) pairs.

Decide, `LocomotionViolationDecider`. Declares a violation if $\rho > 0$ and $r > 0.3$, meaning the oracle still gives non-negligible reward. It returns VIOLATION or NO-VIOLATION.

Act, `VerificationEmitter`. Wraps the decision into a `VerificationResult` artifact with the violated norm ID, the outcome, and an evidence string, for example `oob=0.5 safety=0.8 group=velocity`.

C.5 Refinement: Patch Generation and Regression Testing

Refinement designs Gaussian suppression patches in the MoE bottleneck space and validates each patch against the demonstration basin.

Observe, `LocomotionQueueObserver`. Dequeues verified violations from the triage queue and loads the oracle reference, strictness parameters $(\sigma_{\min}, \sigma_{\max}, \Delta_{\max})$, and current oracle version.

Orient, `BottleneckBandwidthOrienter`. Assigns a proposed bandwidth and strength to each flaw based on its severity ρ. Two modes are supported:

- **Fixed mode:** positive severity–bandwidth relationship.

$$\sigma = \mathrm{clip}\big(\sigma_{\min} + (\sigma_{\max} - \sigma_{\min}) \cdot \min(\rho, 1),\ \sigma_{\min},\ \sigma_{\max}\big), \tag{6}$$
$$s = \mathrm{clip}(1.2 \cdot r(\mathbf{x}),\ 0.5,\ 1.0). \tag{7}$$

  Higher severity gives wider bandwidth.
- **Adaptive mode:** inverted relationship.

$$\sigma = \mathrm{clip}\big(\sigma_{\max} - (\sigma_{\max} - \sigma_{\min}) \cdot \min(\rho, 1),\ \sigma_{\min},\ \sigma_{\max}\big), \tag{8}$$
$$s = \mathrm{clip}(0.5 + 0.45 \cdot \min(\rho, 1),\ 0.5,\ 0.95). \tag{9}$$

  Marginal violations receive wide bandwidth to catch the boundary region; severe violations receive narrow, high-strength patches for surgical correction.

Decide, `BottleneckRegressionDecider`. The core anti-corruption mechanism tests each patch in the dominant expert's bottleneck space, weighted by gating probabilities. For each proposed patch at flaw $\mathbf{x}$:

1. **Expert selection.** $e^* = \arg\max_e g_e(\mathbf{x})$.
2. **Encoding.** $\mathbf{c} = \mathrm{ReLU}(W_{e^*}\mathbf{x} + b_{e^*})$, the patch centre in expert e^*'s bottleneck space.
3. **Demo suppression.** For each demo point $\mathbf{x}_d$, compute the gating-weighted Gaussian suppression:
$$\delta_d = g_{e^*}(\mathbf{x}_d) \cdot s \cdot \exp\Big(-\frac{\|\mathbf{z}_{e^*}(\mathbf{x}_d) - \mathbf{c}\|^2}{2\sigma^2}\Big). \tag{10}$$
4. **Cumulative tracking.** $\Delta_d^{\text{total}} = \Delta_d^{\text{prior}} + \delta_d$, where Δ_d^{prior} is the accumulated suppression from previously accepted patches. This is $O(N)$ per patch rather than $O(KN)$.
5. **Bandwidth shrinking.** If $\max_d \Delta_d^{\text{total}} > \Delta_{\max}$, binary-search for the largest $\sigma' \in [\sigma_{\min}, \sigma]$ that satisfies the constraint, up to 15 bisection iterations.
6. **Strength reduction.** If shrinking alone is insufficient, solve for the maximum safe strength: $s' = (\Delta_{\max} - \Delta_{d^*}^{\text{prior}})/(g_{e^*}(\mathbf{x}_{d^*}) \cdot \gamma_{d^*})$, where $d^* = \arg\max_d \Delta_d^{\text{total}}$ and γ_{d^*} is the Gaussian value.
7. **Rejection.** If $\sigma' < \sigma_{\min}$ or $s' < 0.1$, the patch is rejected.

Decide (alternative), `BottleneckClusterDecider`. An alternative decider uses geometric exclusion zones instead of point-wise suppression testing. DBSCAN is run per expert in bottleneck space to identify dense manifold regions. Each demo point receives a safeguard ball: core samples use k-NN distance and outliers use 1-NN distance. A patch is rejected if its centre falls inside any safeguard ball, or if its Gaussian tail leaks past the ball boundary by more than the leakage threshold. This provides continuous geometric guarantees over the interpolation region between demo points.

Act, `LocomotionBatchDeployer`. Bundles accepted patches into a `GovernanceBatch` artifact with:

- One `SPATIAL_FLAW_PATCH` local correction per patch, carrying the flaw point, bandwidth, strength, violated group, and expert index.
- Regression evidence metadata, including accepted, rejected, and shrunk counts.
- Oracle version transition, from and to version strings.
- Signature `regression-verified`.

The batch is submitted to the oracle, which encodes each flaw through its dominant expert, stores the patch centre $\mathbf{c} = \mathbf{z}_{e^*}$, and increments the oracle version.

C.6 Oracle Adapter: `MoELocomotionOracle`

The oracle wraps the trained MoE autoencoder and accumulates patches.

Initialisation. Precomputes per-expert bottleneck encodings of all demo observations ($\mathbf{Z}_e \in \mathbb{R}^{N\times B}$ for each expert e) and demo gating weights ($G \in \mathbb{R}^{N\times E}$). Optionally runs DBSCAN per expert to build safeguard balls for the cluster decider.

Query. For a batch of M observations:

1. Compute reconstruction MSE and map to reward via Eq. (4).
2. Compute suppression via Eq. (15) over all accumulated patches.
3. Return safety score $r = \max(0,\ r_{\text{moe}} - \text{supp})$ and uncertainty $u = \min(0.9,\ 0.2 + \rho)$.

Patch deployment. For each `SPATIAL_FLAW_PATCH` correction in a governance batch, the oracle determines the dominant expert, $e^* = \arg\max_e g_e(\mathbf{x})$, encodes the flaw through that expert, and appends the resulting centre, bandwidth, and strength to expert e^*'s patch list.

In-basin regression check. After deployment, the oracle can perform a secondary regression check against noisy in-basin test points, i.e. demo observations plus $\mathcal{N}(0, 0.1\sigma_d)$ clipped to effective bounds. Patches whose gating-weighted suppression exceeds max_ib_supp $= 0.01$ on any in-basin point undergo bandwidth shrinking by a factor of 0.9 per step and, if necessary, strength reduction.

C.7 Supporting Roles

Triage (`FIFOTriage`). A simple FIFO queue buffers verified violations and releases them to refinement in order of verification.

Blue Team (`CollateralMonitor`). After each governance iteration, re-queries the oracle on all previously discovered but not directly patched flaws. It counts how many are now suppressed below τ_{floor} by nearby patches, quantifying collateral suppression, a desirable side-effect where one patch covers multiple nearby violations.

Enforcement (`DefaultEnforcement`). Gates deployment decisions at inference time using oracle uncertainty, safety score, and coverage. Observations with uncertainty above threshold are

escalated; those with safety below a margin are blocked; the remainder are allowed with optional audit logging.

Proposer (`PassthroughProposer`). Wraps context data as a message trajectory with minimal processing, used for data forwarding in the governance pipeline.

C.8 Constraint Types

Two families of constraints are supported.

Dimension-bound constraints. Per-dimension bounds are derived from demonstration statistics:

$$\text{lo}_d = \mu_d^{\min}, \tag{11}$$
$$\text{hi}_d = \mu_d^{\max}. \tag{12}$$

Each dimension group, such as position or velocity, has independent bounds. The out-of-bounds margin can be extended asymmetrically using per-side standard deviations:

$$\text{effective_lo}_d = \mu_d^{\min} - m \cdot \sigma_d^-, \tag{13}$$
$$\text{effective_hi}_d = \mu_d^{\max} + m \cdot \sigma_d^+, \tag{14}$$

where m is the OOB margin multiplier and σ_d^-, σ_d^+ are the standard deviations of values below and above the per-dimension mean, respectively.

Safety constraints (Ant). Three explicit physical constraints are defined for the Ant environment: velocity cap $\sqrt{v_x^2 + v_y^2} \leq 6.0$, torso tilt $2 \arccos |q_w| \leq 46°$, and hip angular velocity $\max_i |v_i| \leq 8.0$. These can be combined by logical OR for a single "any-constraint" kernel. The same regression-testing procedure applies: patches are tested against constraint-satisfying observations to prevent over-suppression. The downstream constraint-targeted analysis is reported in Section G.

D Scalable Patch Evaluation

D.1 Suppression Function

Given patches $\mathcal{P}_e = \{(\mathbf{c}_k, \sigma_k, s_k)\}_{k=1}^{K_e}$ for expert e, the total suppression for observation $\mathbf{x}$ is

$$\text{supp}(\mathbf{x}) = \min\left(1, \sum_{e=1}^{E} \alpha_e(\mathbf{x}) \sum_{k=1}^{K_e} s_k \exp\left(-\frac{\|\mathbf{z}_e(\mathbf{x}) - \mathbf{c}_k\|^2}{2\sigma_k^2}\right)\right). \tag{15}$$

Any Gaussian term below 10^{-3} is clamped to zero. This Gaussian floor gives finite support in practice and enables locality-sensitive hashing.

D.2 Vectorized Brute-Force Evaluation

The vectorized implementation pre-transposes patch centers into $(E, B, K_{\max})$ arrays, precomputes $\|\mathbf{c}_k\|^2$ and $1/(2\sigma_k^2)$, encodes all experts for a batch of N observations, and uses $\|a - b\|^2 = \|a\|^2 + \|b\|^2 - 2a^\top b$ to compute all Gaussian values. The aggregate suppression is $\min(1, \sum_e \alpha_e \sum_k G_{e,k})$. The cost is $O(N \cdot E \cdot K_{\max})$ per batch.

D.3 LSH Build and Query

For every expert, patch centers are stored in an approximate nearest-neighbor index over bottleneck space. Each hash table samples normalized Gaussian hyperplanes, projects patch centers, thresholds

each bit by the median projection for that expert, and groups patch indices by integer hash code. The default settings are reported in Table 8.

At query time, the current bottleneck code is hashed with the same projections and thresholds. Candidates from matching buckets are deduplicated across tables. The implementation then gathers the relevant (o, e, k) triples, evaluates the Gaussian suppression only for those triples, and scatters the weighted values back to observations using the corresponding gating weights.

Table 8: LSH default hyperparameters.

Parameter	Value
Number of tables T	4
Bits per table b	auto-scaled, range 14–24
Bit cap	$B + 2$
Gaussian floor	10^{-3}
Gaussian cutoff radius	$3.72\,\sigma$
RNG seed	42

Optional top-K expert pruning. The reported experiments use all experts, because the number of experts is small. For larger artifacts, the same LSH query can be combined with top-K expert pruning by evaluating only the experts with the largest gating weights, masking and renormalizing inactive experts, and flattening only the active observation–expert pairs. This option is a scaling extension, not part of the reported experimental protocol.

D.4 Scaling Measurements

The ablation study isolates the suppression step only; encoding and gating are excluded to measure pure scaling behavior. The default setup uses bottleneck dimension $B = 10$, batch size $N = 1000$, query vectors $\mathrm{ReLU}(\mathcal{N}(0, 3))$, patch centers sampled uniformly from $[0, 5]^B$, bandwidth $\sigma = 0.3$, strength $s = 0.8$, and median timing over 5 runs with 1 warmup.

Table 9: Suppression time vs. patch count ($E = 1$, $B = 10$, $N = 1000$).

K	Bits	Brute (ms)	LSH (ms)	Speedup
100	14	1.1	0.3	3.7×
1,000	14	12.4	1.2	10.1×
10,000	18	175.2	2.5	71.4×
100,000	21	1,420	6.2	228×
1,000,000	24	19,227	15.2	1,263×

Table 10: Suppression time vs. expert count ($K_e = 10{,}000$, $B = 10$, $N = 1000$).

E	K_{total}	Brute (ms)	LSH (ms)	Brute/E	LSH/E
1	10,000	158	4.8	158	4.8
4	40,000	858	15.6	215	3.9
16	160,000	4,615	108	288	6.7
32	320,000	8,384	215	262	6.7
64	640,000	17,318	482	271	7.5

Table 11: Optional top-K scaling study: suppression cost vs. expert count ($K_{\text{top}} = 2$, $K_e = 10{,}000$, $B = 10$, $N = 1000$). This is a large-scale extension and is not used in the reported MuJoCo experiments.

E	Dense BF	Top-2 BF	Dense LSH	Top-2 LSH	$\frac{\text{BF}_K}{\text{BF}}$	$\frac{\text{LSH}_K}{\text{LSH}}$
2	502	518	13.0	13.5	103%	104%
4	1,257	971	28.2	28.8	77%	102%
8	2,048	1,467	55.6	24.2	72%	44%
16	4,319	822	120	36.3	19%	30%
32	8,479	334	216	33.6	3.9%	16%
64	18,090	287	432	36.9	1.6%	8.5%

Table 12: FK patch cap sweep: mean eval return $\pm$ std. No comparison reaches significance (Welch's t-test, all $p > 0.05$). *Fewer than 5 seeds due to log parser failures.

Method	Ant-v2 (400K)	HalfCheetah-v2 (600K)	Hopper-v2 (600K)	Walker2d-v2 (600K)
MoE-GUIDE	2692 ± 792	9236 ± 604	3369 ± 116	3887 ± 600
FK cap200	2639 ± 799	9303 ± 879	$3371 \pm 221^*$	$3535 \pm 449^*$
FK cap400	3433 ± 783	9241 ± 1003	3354 ± 122	$4367 \pm 316^*$
FK cap600	2947 ± 859	9870 ± 375	3388 ± 117	$4270 \pm 663^*$
FK cap800	2995 ± 618	9637 ± 164	2417 ± 1255	$3832 \pm 370^*$
FK cap1000	2867 ± 614	9744 ± 253	2815 ± 989	3587 ± 967

E Patch Robustness and Variant Ablations

E.1 FK Patch Budget Sweep

To assess whether the flywheel kernel's patch budget affects task reward, we sweep the maximum number of patches from 200 to 1000 across four environments and compare against the unpatched MoE-GUIDE baseline. All variants share the same hyperparameters and differ only in the `-flywheel_variant tight_regression_cap`N flag, which limits how many suppression patches the flywheel kernel may accumulate. Table 12 reports the mean evaluation return at matched training steps where all seeds have data; the accompanying learning curves show the corresponding training trajectories. No comparison reaches statistical significance; point estimates trend positive in Ant and HalfCheetah and are mixed in Hopper and Walker2d.

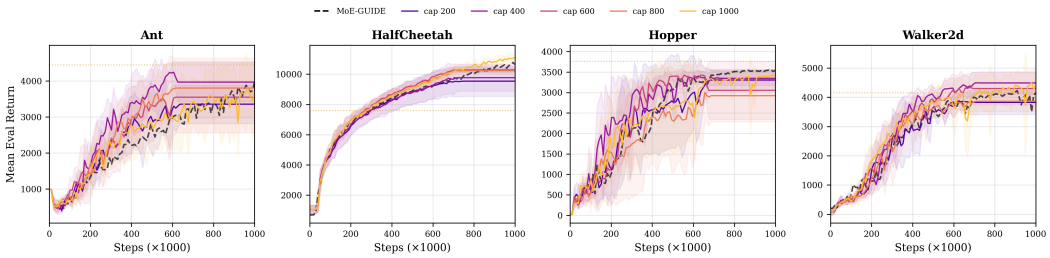


Figure 3: Learning curves for the FK patch budget sweep in Table 12. Curves show mean evaluation return across seeds, with shaded bands for one standard deviation.

E.2 OOB Margin Ablation

During FK kernel construction, the governance loop identifies *out-of-bounds* (OOB) observations—those whose observation dimensions fall outside the demonstrated $[\min, \max]$ range—and places

Table 13: OOB margin ablation: final mean evaluation return ± std across 5 seeds. Swimmer at 500K steps; all others at 1M steps. All variants use regression patch identification with cap 1,000. Each environment uses its best tightness level. Bold indicates best FK variant per environment.

Method	Swimmer	Hopper	Walker2d‡	HalfCheetah‡	Ant‡
MoE-GUIDE	342 ± 2	3540 ± 86	4146 ± 877	10646 ± 657	3974 ± 1099
FK (m=0.0)	36 ± 17	3423 ± 101	$\mathbf{4424 \pm 536}$	$\mathbf{11096 \pm 216}$	3647 ± 926
FK (m=0.5)	117 ± 5	3265 ± 536	4260 ± 284	10382 ± 819	$\mathbf{5086 \pm 217}$
FK (m=1.0)	$\mathbf{191 \pm 4}$	$\mathbf{3495 \pm 116}$	4268 ± 344	10800 ± 524	4443 ± 370

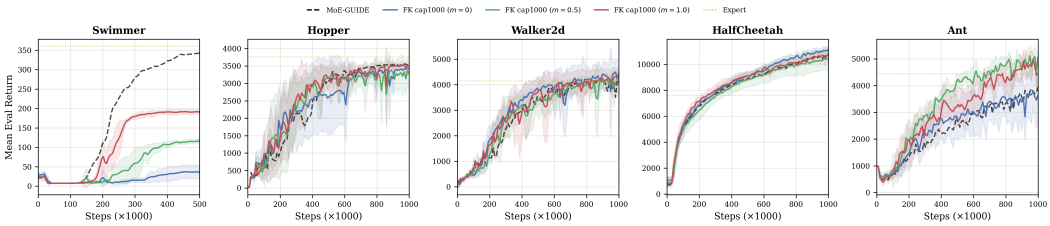


Figure 4: OOB margin ablation across five MuJoCo environments. Bars show mean evaluation return (± std) for FK kernels with margin $m \in \{0, 0.5, 1.0\}$. Dashed line: MoE-GUIDE baseline; dotted line: expert demonstration return.

suppression patches at the corresponding bottleneck embeddings. The **margin** parameter m relaxes this detection boundary by m standard deviations per dimension:

$$\text{effective}_{\text{lo}} = \min -m \cdot \sigma, \qquad \text{effective}_{\text{hi}} = \max +m \cdot \sigma, \tag{16}$$

where σ is the per-dimension standard deviation of the demo data. With $m = 0$ (no margin), any observation outside the exact demo range triggers patching; with $m > 0$, observations must deviate further before patches are placed, reducing overly aggressive suppression near the demonstration boundary.

Table 13 compares FK kernels built with margin $m \in \{0, 0.5, 1.0\}$, all using the regression patch-identification method and a 1,000-patch cap. Each environment uses its best tightness level (Swimmer: very loose; all others: tight). Figure 4 visualises the results.

The margin has a pronounced effect in two environments (Figure 5):

- **Swimmer** ($d_{\text{obs}} = 8$): Without margin, FK with cap 1,000 collapses to 36 ± 17—well below both MoE-GUIDE (342) and SAC (78). Zero-margin patching is overly aggressive in this low-dimensional, tight-distribution environment: too many near-boundary observations receive suppression, starving the agent of intrinsic reward. Increasing the margin progressively recovers performance (m=0.5: 117; m=1.0: 191), though neither fully reaches the MoE-GUIDE baseline. Note that the uncapped FK variant used in Table 1 (341 ± 3) does not suffer this collapse, indicating that the interaction between the patch cap and margin determines which observations receive patches.
- **Ant** ($d_{\text{obs}} = 111$): Margin m=0.5 achieves 5086 ± 217, the highest FK result across all ablations—surpassing both the no-margin cap-1,000 variant (3647 ± 926, +39%) and the uncapped FK Patching entry in Table 1 (4816 ± 594). A moderate relaxation of the OOB boundary in the high-dimensional Ant observation space prevents patch overgeneralisation while still suppressing genuinely out-of-distribution states.

In Hopper, Walker2d, and HalfCheetah the differences are within one standard deviation, indicating that the margin is less critical in these environments. HalfCheetah favours m=0, consistent with its strong performance under the original FK kernel.

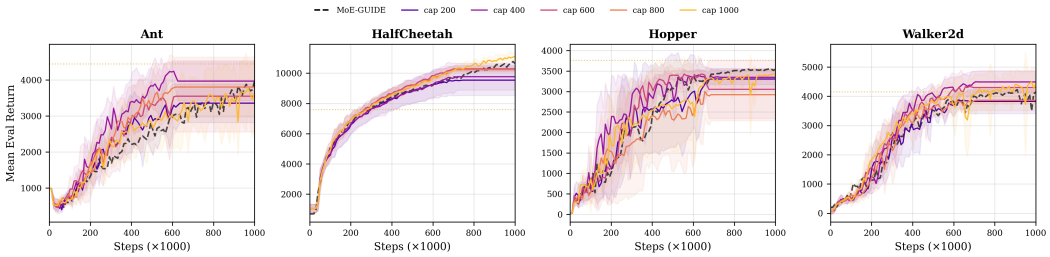
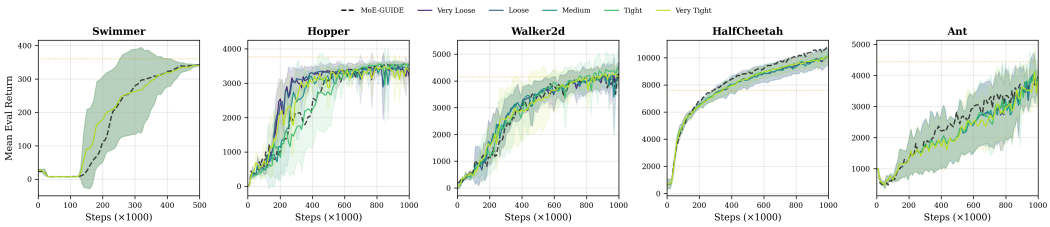


Figure 5: Effect of OOB margin on FK Patching in Swimmer (left) and Ant (right). Swimmer shows progressive recovery from collapse as margin increases. Ant peaks at m=0.5, achieving the highest FK result across all ablations.



Tightness	Margin	Ant[‡]	HalfCheetah[‡]
Tight	m=0.0	3647 ± 926	$\mathbf{11096} \pm \mathbf{216}$
Tight	m=0.5	$\mathbf{5086} \pm \mathbf{217}$	10382 ± 819
Tight	m=1.0	4443 ± 370	10800 ± 524
Medium	m=1.0	4680 ± 835	10542 ± 612
Loose	m=1.0	4263 ± 946	10790 ± 606

Table **??** and Figure 6 report additional tightness–margin combinations for Ant and HalfCheetah, which had the most kernel variants available. In Ant, the margin consistently improves over the corresponding no-margin baseline across all tightness levels.

E.3 Constraint Tightness Ablation

Table 14 reports the effect of varying FK constraint tightness from very loose to very tight using the regression patch-identification method. MoE-GUIDE is shown as the baseline. Performance is largely robust to the tightness setting: variations are within one standard deviation in most environments, suggesting that the method is not highly sensitive to this hyperparameter.

Table 14: FK constraint tightness ablation: final mean evaluation return $\pm$ std across 5 seeds. Swimmer at 500K steps; all others at 1M steps.

Method	Swimmer	Hopper	Walker2d	HalfCheetah	Ant
MoE-GUIDE	342 ± 2	3540 ± 86	4146 ± 877	10646 ± 657	3974 ± 1099
Very Loose	339 ± 3	3191 ± 425	4174 ± 457	10004 ± 587	3858 ± 366
Loose	341 ± 3	3335 ± 198	4259 ± 405	10010 ± 661	3858 ± 366
Medium	339 ± 3	3344 ± 264	4280 ± 374	9980 ± 627	3941 ± 527
Tight	339 ± 3	3544 ± 76	4535 ± 722	10097 ± 457	3827 ± 681
Very Tight	340 ± 2	3322 ± 318	4172 ± 354	10016 ± 555	3875 ± 403

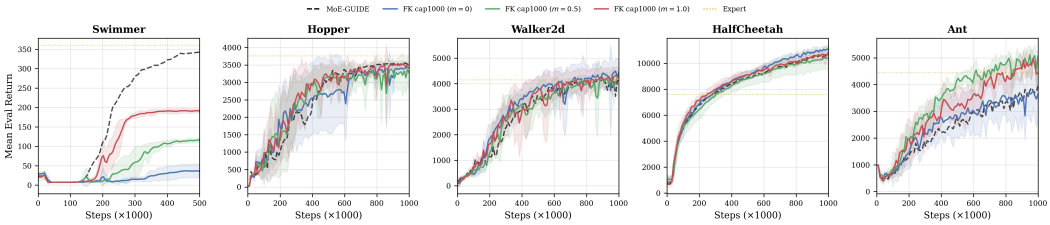


Figure 6: Tightness–margin interaction for Ant-v2 (left) and HalfCheetah-v2 (right). Ant benefits from margin across all tightness levels. HalfCheetah performs best without margin at tight tightness.

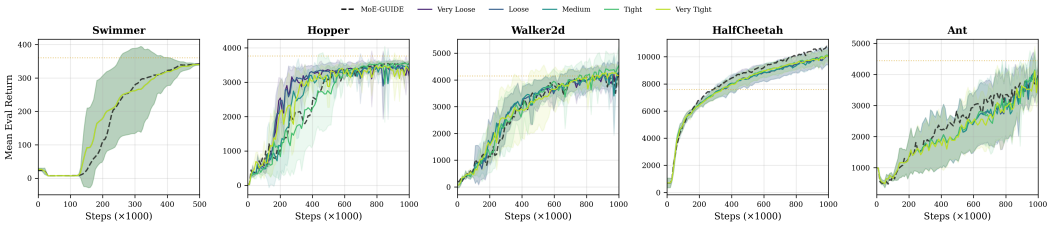


Figure 7: Learning curves for the constraint-tightness ablation in Table 14. Curves show mean evaluation return across seeds, with shaded bands for one standard deviation.

E.4 Patch-Identification Method: Regression vs. DBSCAN

Table 15 compares two patch-identification strategies for the FK kernel at tight strictness: linear regression of reconstruction error on observation dimensions and DBSCAN clustering of high-error observations. Both methods yield similar final returns across environments, with regression performing slightly better in Walker2d and DBSCAN showing a notable drop in Swimmer.

Table 15: Regression vs. DBSCAN patch identification: final mean evaluation return $\pm$ std across 5 seeds. Swimmer at 500K steps; all others at 1M steps.

Method	Swimmer	Hopper	Walker2d	HalfCheetah	Ant
MoE-GUIDE	342 ± 2	3540 ± 86	4146 ± 877	10646 ± 657	3974 ± 1099
Regression	339 ± 3	3544 ± 76	4535 ± 722	10097 ± 457	3827 ± 681
DBSCAN	250 ± 4	3542 ± 94	4186 ± 293	9864 ± 693	3917 ± 498

F Bottleneck Locality Ablations

This section provides the full per-coverage-level results for the bottleneck-versus-raw-space ablation. The experiment measures how many greedy ball-cover centers are needed to cover 2,000 OOB test points at several coverage thresholds. The ball radius is the median 10-nearest-neighbor distance in each respective space. Three conditions are compared: bottleneck space with its own radius (BN), raw observation space with its own radius (Raw), and raw space with the bottleneck radius (Raw@BN-r).

At 50% coverage, bottleneck space can appear worse than raw space because the first few greedy centers in raw space cover more points when the raw radius is large. At 75% coverage and above, the bottleneck consistently requires fewer patches. The Raw@BN-r condition shows the cost of using a fine bottleneck-scale radius in raw space: nearly every test point needs its own patch because fixed-radius balls become sparse in high-dimensional observation space.

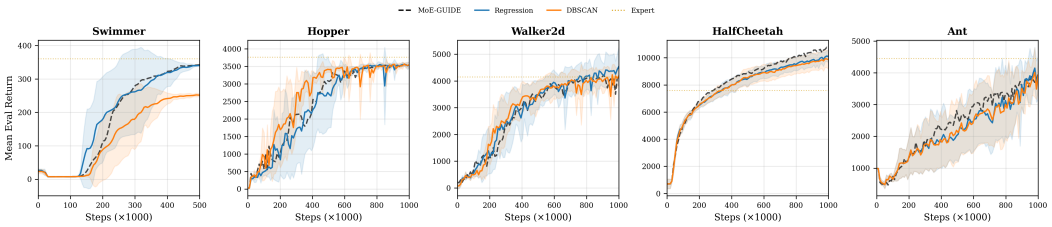


Figure 8: Learning curves for the patch-identification ablation in Table 15. Curves show mean evaluation return across seeds, with shaded bands for one standard deviation.

Table 16: Ant ($D = 111 \rightarrow B = 10$, 3 experts): patches at each coverage level.

Coverage	BN	Raw	Raw@BN-r	Ratio	CoD
50%	37	26	996	0.7×	26.9×
75%	192	322	1496	1.7×	7.8×
90%	492	622	1796	1.3×	3.7×
95%	592	722	1896	1.2×	3.2×

F.1 Bottleneck Dimension Sweep

This ablation varies the bottleneck dimension B for Ant ($D = 111$) while holding the number of experts fixed at 3. For each B, a fresh MoE is trained for 2,000 epochs with Adam (lr $= 10^{-3}$), MSE loss, and full-batch training on all 26,594 demonstration observations. Patch efficiency is measured by greedy ball-cover over 2,000 OOB test points with median 10-NN radius.

The sweep shows three regimes. Aggressive compression ($B \leq 5$) maximizes patch efficiency but has higher reconstruction loss. The range $B \in [7, 10]$ gives low reconstruction loss and stable patch counts. For $B > 10$, additional dimensions add little reconstruction accuracy while increasing the intrinsic dimensionality of the patch space.

G Constraint-Targeted Patching and OOB Enforcement

This section provides the complete Ant-v2 constraint analysis summarized in the main paper. Results are reproduced by `reproduce_constraint_analysis.py` from cached data in `ant_constraint_cache.pkl`.

G.1 Experimental Setup

We define three Ant-v2 constraints inspired by Safety Gymnasium: speed $\sqrt{v_x^2 + v_y^2} \leq 6.0$, torso inclination from upright $\leq 45°$, and hip velocity $\max_i |v_{\text{hip},i}| \leq 5.0$. Constraint violations are measured per evaluation episode as the fraction of transitions violating each constraint. The any-rate is the fraction of transitions violating at least one constraint. Constraint episodes are paired with the nearest evaluation checkpoint within $\pm$20K steps.

The expert demonstrations contain 26,594 observations from 12 trajectory files. Of these, 85.8% violate at least one constraint, reflecting the nature of Ant-v2 locomotion: agile movement involves high speeds, tilting, and rapid joint motion.

We evaluate CT, FK, and OOB regimes. CT places local suppression patches in MoE bottleneck space at constraint-violating demonstration observations. FK uses high-reconstruction-error flywheel patches across all observation dimensions with a cap of 1000 patches. OOB checks all 27 active observation dimensions against the demonstrated range, with hard clipping or smooth Gaussian decay, and optional safety functions.

Table 17: HalfCheetah ($D = 17 \rightarrow B = 7$, 4 experts): patches at each coverage level.

Coverage	BN	Raw	Raw@BN-r	Ratio	CoD
50%	35	30	885	0.9×	25.3×
75%	128	160	1385	1.2×	10.8×
90%	340	436	1685	1.3×	5.0×
95%	440	536	1785	1.2×	4.1×

Table 18: Hopper ($D = 11 \rightarrow B = 4$, 2 experts): patches at each coverage level.

Coverage	BN	Raw	Raw@BN-r	Ratio	CoD
50%	47	39	988	0.8×	21.0×
75%	129	157	1488	1.2×	11.5×
90%	284	402	1788	1.4×	6.3×
95%	384	502	1888	1.3×	4.9×

G.2 Violation Rates by Training Phase

Table 24 shows how violation rates evolve during training. CT methods start with violation rates comparable to MoE-GUIDE and show no consistent decrease over time. OOB methods show an initial dip followed by a gradual increase as the policy learns higher-return behavior.

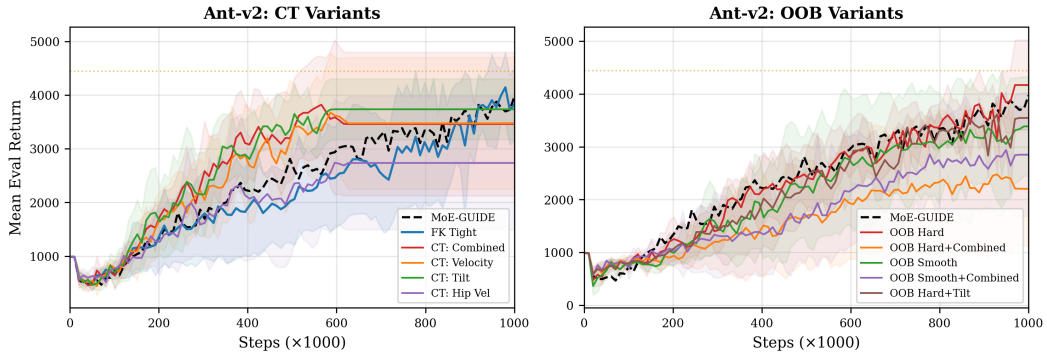


Figure 9: Ant constraint-targeted learning curves accompanying Table 24. Curves show evaluation return and violation rates over training for SAC, MoE-GUIDE, FK, CT, and OOB variants.

G.3 Per-Constraint Violation Rates

G.4 Reward–Violation Correlation

Speed violations show a strong positive correlation with reward across methods, confirming that faster locomotion yields higher returns and is difficult to suppress without affecting performance. Tilt shows weaker and sometimes negative correlations, suggesting it can be reduced more independently.

G.5 Patch Suppression Mechanism Verification

To verify that CT patches function as designed, we evaluate patched and unpatched MoE models directly on the demonstration dataset, split into constraint-satisfying observations ($n = 3{,}784$) and constraint-violating observations ($n = 22{,}810$). Table 27 confirms that patches suppress violated observations while leaving safe observations virtually unchanged, but coverage is limited: CT Combined achieves meaningful suppression ($s > 0.1$) on 21.4% of violated demo observations.

Table 19: Swimmer ($D = 8 \rightarrow B = 3$, 1 expert): patches at each coverage level.

Coverage	BN	Raw	Raw@BN-r	Ratio	CoD
50%	57	44	967	0.8×	17.0×
75%	127	156	1467	1.2×	11.6×
90%	227	397	1767	1.7×	7.8×
95%	313	497	1867	1.6×	6.0×

Table 20: Walker2d ($D = 17 \rightarrow B = 7$, 4 experts): patches at each coverage level.

Coverage	BN	Raw	Raw@BN-r	Ratio	CoD
50%	48	34	981	0.7×	20.4×
75%	150	188	1481	1.3×	9.9×
90%	385	488	1781	1.3×	4.6×
95%	485	588	1881	1.2×	3.9×

Two factors explain why CT patches reduce violations in IL-only evaluation but not during RL training: the patch budget covers only a fraction of the violation manifold, and environment reward dominates the intrinsic suppression scale.

G.6 Matched Reward Levels and OOB Variants

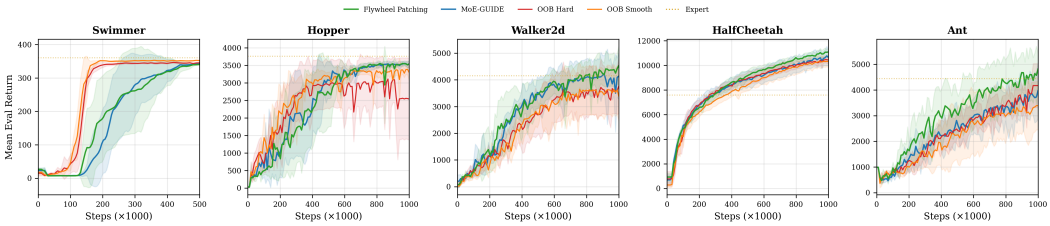


Figure 10: Learning curves for the OOB variants summarized in Table 29. Curves show evaluation return and any-constraint violation rate over training.

H Slow Verifier Motivation

Tables 30 and 31 report representative latency measurements. They motivate offline compilation but are not part of an implemented image-based system in this paper.

I Implementation and Artifact Format

Each kernel is serialized as a PyTorch `.pt` file via `torch.save`. Table 32 lists the stored fields. The kernel includes model weights, model configuration, reward-mapping parameters, dimension statistics, patch records, strictness metadata, environment name, and number of demonstration points.

The flywheel roles exchange typed artifacts. `CandidateFlaw` carries the observation vector, OOB ratio, violated group, safety value, uncertainty estimate, and oracle version. `VerificationResult` records the verification outcome, violated norm ID, and evidence string. `GovernanceBatch` contains local corrections, regression evidence, oracle-version transitions, and deployment signatures. These records provide an audit trail from candidate discovery to patch deployment.

Table 21: AntMaze ($D = 29 \rightarrow B = 8$, 3 experts): patches at each coverage level.

Coverage	BN	Raw	Raw@BN-r	Ratio	CoD
50%	39	29	998	0.7×	25.6×
75%	159	184	1498	1.2×	9.4×
90%	408	484	1798	1.2×	4.4×
95%	508	584	1898	1.1×	3.7×

Table 22: Average neighbours within the ball radius.

Env	D	B	BN radius	Raw radius	Avg neigh (BN)	Avg neigh (Raw@BN-r)
Ant	111	10	1.22	4.60	21.1	1.0
HalfCheetah	17	7	0.96	5.35	23.3	1.2
Hopper	11	4	0.45	3.91	15.4	1.0
Swimmer	8	3	0.25	2.56	11.8	1.0
Walker2d	17	7	1.17	5.15	16.0	1.0
AntMaze	29	8	1.15	7.55	29.2	1.0

References

Pieter Abbeel and Andrew Y Ng. Apprenticeship learning via inverse reinforcement learning. In *Proceedings of the twenty-first international conference on Machine learning*, pp. 1, 2004.

Joshua Achiam, David Held, Aviv Tamar, and Pieter Abbeel. Constrained policy optimization. In *International conference on machine learning*, pp. 22–31. Pmlr, 2017.

Mohammed Alshiekh, Roderick Bloem, Rüdiger Ehlers, Bettina Könighofer, Scott Niekum, and Ufuk Topcu. Safe reinforcement learning via shielding. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32, 2018.

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

Arthur Aubret, Laetitia Matignon, and Salima Hassas. An information-theoretic perspective on intrinsic motivation in reinforcement learning: A survey. *Entropy*, 25(2):327, 2023.

Mattijs Baert, Elias Malomgré, Sam Leroux, and Pieter Simoens. Reward machine inference for robotic manipulation. In *Inductive Biases in Reinforcement Learning Workshop@ RLC 2025*.

Mattijs Baert, Pietro Mazzaglia, Sam Leroux, and Pieter Simoens. Maximum causal entropy inverse constrained reinforcement learning. *Machine Learning*, 114(4):103, 2025.

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. *arXiv preprint arXiv:1810.12894*, 2018.

Paul F Christiano, Jan Leike, Tom Brown, Miljan Martic, Shane Legg, and Dario Amodei. Deep reinforcement learning from human preferences. *Advances in neural information processing systems*, 30, 2017.

Junfeng FANG, Houcheng JIANG, Kun WANG, Yunshan MA, Jie SHI, Xiang WANG, Xiangnan HE, and Tat-Seng AlphaEdit CHUA. Null-space constrained knowledge editing for language models.(2025). In *Proceedings of the 13th International Conference on Learning Representations (ICLR 2025), Singapore, April*, pp. 24–28.

Table 23: Full bottleneck dimension sweep for Ant ($D = 111$, 3 experts, 2,000 epochs, 2,000 OOB test points).

B	Loss	Avg neigh	P_{50}	P_{75}	P_{90}	P_{95}	Final P	Final cov.	Train (s)
2	0.0999	13.5	51	125	241	336	416	99%	124
3	0.0430	17.7	47	136	311	411	491	99%	127
5	0.0080	19.5	42	159	408	508	588	99%	126
7	0.0032	29.1	35	188	487	587	667	99%	139
10	0.0026	25.5	37	191	489	589	669	99%	136
15	0.0017	26.4	37	242	542	642	722	99%	152
20	0.0014	24.6	33	237	537	637	717	99%	148
30	0.0008	31.4	33	239	539	639	719	99%	189
50	0.0002	25.5	34	263	563	663	743	99%	221
80	0.0003	28.1	31	247	547	647	727	99%	271
111	—	42.5	26	322	622	722	802	99%	—

Table 24: Violation rate (any_rate) by training phase.

Method	0–100K	100–200K	200–400K	400–600K
SAC	0.804	0.806	0.808	0.801
MoE-GUIDE	0.734	0.690	0.776	0.794
FK tight	0.734	0.645	0.655	0.684
CT: Combined	0.785	0.773	0.785	0.822
CT: Velocity	0.770	0.763	0.806	0.820
CT: Tilt	0.792	0.780	0.813	0.839
CT: Hip Vel	0.791	0.640	0.748	0.799
OOB Hard+Comb.	0.485	0.095	0.170	0.290
OOB Hard+Tilt	0.512	0.173	0.415	0.604
OOB Smooth+Comb.	0.492	0.160	0.178	0.244
OOB Smooth+Tilt	0.561	0.271	0.464	0.617

Leo Gao, John Schulman, and Jacob Hilton. Scaling laws for reward model overoptimization. In *International Conference on Machine Learning*, pp. 10835–10866. PMLR, 2023.

Tom Hartvigsen, Swami Sankaranarayanan, Hamid Palangi, Yoon Kim, and Marzyeh Ghassemi. Aging with grace: Lifelong model editing with discrete key-value adaptors. *Advances in Neural Information Processing Systems*, 36:47934–47959, 2023.

Jonathan Ho and Stefano Ermon. Generative adversarial imitation learning. *Advances in neural information processing systems*, 29, 2016.

Zeyu Huang, Yikang Shen, Xiaofeng Zhang, Jie Zhou, Wenge Rong, and Zhang Xiong. Transformer-patcher: One mistake worth one neuron. *arXiv preprint arXiv:2301.09785*, 2023.

Rodrigo Toro Icarte, Toryn Klassen, Richard Valenzano, and Sheila McIlraith. Using reward machines for high-level task specification and decomposition in reinforcement learning. In *International Conference on Machine Learning*, pp. 2107–2116. PMLR, 2018.

Rodrigo Toro Icarte, Toryn Q Klassen, Richard Valenzano, and Sheila A McIlraith. Reward machines: Exploiting reward function structure in reinforcement learning. *Journal of Artificial Intelligence Research*, 73:173–208, 2022.

Chi Jin, Akshay Krishnamurthy, Max Simchowitz, and Tiancheng Yu. Reward-free exploration for reinforcement learning. In *International Conference on Machine Learning*, pp. 4870–4879. PMLR, 2020.

Table 25: Per-constraint violation rates, global average across all steps and seeds.

Method	Speed	Tilt	Hip vel
SAC	0.001	0.430	0.534
MoE-GUIDE	0.027	0.473	0.469
FK tight	0.014	0.375	0.460
CT: Combined	0.033	0.457	0.529
CT: Velocity	0.019	0.479	0.503
CT: Tilt	0.027	0.519	0.503
CT: Hip Vel	0.001	0.466	0.399
OOB Hard+Comb.	0.014	0.148	0.262
OOB Hard+Tilt	0.008	0.136	0.446
OOB Smooth+Comb.	0.016	0.128	0.273
OOB Smooth+Tilt	0.024	0.166	0.452

Table 26: Pearson r between evaluation reward and violation rate.

Method	r(any)	r(speed)	r(tilt)	r(hip)
SAC	-0.02	$+0.33$	-0.27	$+0.31$
MoE-GUIDE	$+0.25$	$+0.70$	$+0.14$	$+0.25$
FK tight	$+0.08$	$+0.66$	$+0.02$	$+0.20$
CT: Combined	$+0.20$	$+0.74$	$+0.17$	$+0.16$
CT: Tilt	$+0.16$	$+0.68$	$+0.10$	$+0.25$
OOB Hard+Comb.	$+0.54$	$+0.81$	$+0.22$	$+0.52$
OOB Smooth+Comb.	$+0.50$	$+0.75$	-0.00	$+0.52$

W Bradley Knox and Peter Stone. Interactively shaping agents via human reinforcement: The tamer framework. In *Proceedings of the fifth international conference on Knowledge capture*, pp. 9–16, 2009.

Bingqing Liu, Wei Liu, and Yuhua Li. Betaedit: Null-space constrained sequential model editing. *arXiv preprint arXiv:2605.09285*, 2026.

Elias Malomgré and Pieter Simoens. Mixture of autoencoder experts guidance using unlabeled and incomplete data for exploration in reinforcement learning. *arXiv preprint arXiv:2507.15287*, 2025.

Elias Malomgré and Pieter Simoens. The alignment flywheel: A governance-centric hybrid mas for architecture-agnostic safety. *arXiv preprint arXiv:2603.02259*, 2026a.

Elias Malomgré and Pieter Simoens. Interactionless inverse reinforcement learning: A data-centric framework for durable alignment. *arXiv preprint arXiv:2602.14844*, 2026b.

Alexander Pan, Kush Bhatia, and Jacob Steinhardt. The effects of reward misspecification: Mapping and mitigating misaligned models. *arXiv preprint arXiv:2201.03544*, 2022.

Deepak Pathak, Dhiraj Gandhi, and Abhinav Gupta. Self-supervised exploration via disagreement. In *International conference on machine learning*, pp. 5062–5071. PMLR, 2019.

Mohamad Louai Shehab, Antoine Aspeel, and Necmiye Ozay. Active reward machine inference from raw state trajectories. *arXiv preprint arXiv:2604.07480*, 2026.

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, Cambridge, MA, 1998.

Table 27: Patch suppression on demonstration data. r_{base} is unpatched MoE reward, s is suppression magnitude, r_{final} is reward after suppression, and Δr is change from unpatched.

		Violated ($n = 22{,}810$)					Safe ($n = 3{,}784$)	
Kernel	Patches	r_{base}	s	r_{final}	Δr	%>0.1	s	Δr
CT Combined	1,063	0.633	0.137	0.539	-0.094	21.4%	0.000	-0.000
CT Velocity	416	0.633	0.011	0.624	-0.009	3.9%	0.000	-0.000
CT Tilt	735	0.633	0.027	0.608	-0.025	7.2%	0.000	-0.000
CT Hip Vel	699	0.633	0.121	0.557	-0.076	15.4%	0.000	-0.000
FK cap1000	922	0.633	0.000	0.633	-0.000	0.0%	0.000	-0.000

Table 28: Violation rate (any_rate) at matched reward levels.

Method	$R \in [500, 1K)$	$R \in [1K, 1.5K)$	$R \in [2K, 2.5K)$	$R \in [2.5K, 3.5K)$
SAC	0.811	0.811	0.803	0.791
MoE-GUIDE	0.687	0.748	0.788	0.809
FK tight	0.698	0.614	0.607	0.667
CT: Combined	0.783	0.773	0.779	0.769
CT: Tilt	0.796	0.771	0.805	0.835
OOB Hard+Comb.	0.194	0.312	0.586	0.638
OOB Smooth+Comb.	0.218	0.213	0.482	0.630

Peng Wang, Zexi Li, Ningyu Zhang, Ziwen Xu, Yunzhi Yao, Yong Jiang, Pengjun Xie, Fei Huang, and Huajun Chen. Wise: Rethinking the knowledge memory for lifelong model editing of large language models. *Advances in Neural Information Processing Systems*, 37:53764–53797, 2024.

Table 29: Full OOB results: final performance over the last 100K steps.

Method	Steps $R{\geq}2K$	Final R	Final Viol.
OOB Hard (no safety)	354K (5)	2379	—
OOB Hard+Combined	447K (3)	3590	0.677
OOB Hard+Velocity	308K (5)	3675	0.773
OOB Hard+Tilt	386K (5)	3253	0.655
OOB Hard+Hip Vel	590K (5)	2860	0.820
OOB Smooth (no safety)	316K (5)	2094	—
OOB Smooth+Combined	578K (5)	3332	0.577
OOB Smooth+Velocity	460K (5)	3579	0.803
OOB Smooth+Tilt	466K (5)	3872	0.654
OOB Smooth+Hip Vel	435K (4)	2684	0.812

Table 30: Per-frame latency: vision pipelines vs. patch evaluation (single CPU core, 640×480 image, median-of-5).

Pipeline stage	Latency (ms)	vs. Patches	Realtime?
MoE + 800 patches (brute)	**0.36**	1×	✓ all freqs
MoE + 800 patches (LSH)	**0.40**	1.1×	✓ all freqs
MobileNetV2 classify	19.8	55×	≤50 Hz only
ResNet-50 classify	62.4	173×	≤15 Hz only
KeypointRCNN (R50-FPN)	5,772	16,033×	<1 Hz
Classify + Keypoint	5,864	16,289×	<1 Hz

Table 31: Safety-signal latency spectrum: from formal logic to human review. Patches compile offline signals into runtime patch lookup.

Safety signal	Example	Latency	Realtime?
Formal logic	Bound check, velocity cap	< 0.01 ms	✓
Light neural net	MobileNetV2 classify	∼20 ms	Marginal
Heavy neural net	Keypoint + detection	∼6,000 ms	×
Multi-model consensus	Ensemble + voting	∼10,000 ms	×
Simulation rollout	Forward dynamics check	∼30,000 ms	×
Human review	Norm / ethical judgment	minutes–hours	×
Compiled patch	**Any of the above**	$< \mathbf{0.5}$ **ms**	✓

Table 32: Kernel file contents.

Key	Description
`moe_state_dict`	MoE model weights
`moe_config`	`{input_dim, bottleneck_dim, num_experts}`
`estimator_config`	`{l_min, l_max, steepness}`
`dim_stats`	Per-dimension `{min, max, mean, std}` tensors
`dim_groups`	Position and velocity dimension groups
`flywheel_patches`	Per-expert patch centers, bandwidths, and strengths
`strictness`	Strictness label and associated patch parameters
`env_name`	Environment name string
`n_demo_points`	Number of demonstration observations used