FORGE: Fine-Tuning Over Reward Generations through Evolution

Anonymous Author(s)
Affiliation
Address
email

Abstract

Designing effective reward functions is a persistent bottleneck in reinforcement learning (RL). Recent work uses large language models (LLMs) to generate task-specific rewards and evolve them against a task-level fitness function. However, state-of-the-art LLM-driven reward search trains and discards a policy for each reward candidate, skipping the upstream-downstream factorization of modern RL: policy initialization is incidental, and reward ordering is ignored. In this study, we hypothesize that restoring this factorization (pretraining a task-agnostic skill prior once, then letting the ordered reward trajectory itself carry the specification) yields stronger policies at a fraction of the compute. To this end, we introduce FORGE (**F**ine-tuning **O**ver **R**eward **G**enerations through **E**volution), which pretrains a skill prior via temporal-distance successor features and then lets an LLM evolve task rewards while each policy adapts through closed-form skill projection. We evaluate FORGE across three tasks. On locomotion and balance maintenance, FORGE surpasses the previous state of the art in fully automated LLM reward search at $10\times$ less compute per reward candidate. On dexterous manipulation, where the upstream prior rarely observes success states, augmenting pretraining with only 100 expert episodes supplies the missing capability and enables FORGE to reach 89.1% of the performance of the published state-of-the-art human-in-the-loop LLM reward evolution.

1 Introduction

Designing effective reward functions remains a central bottleneck in reinforcement learning, especially for tasks whose objectives are easier to recognize than to specify [Amodei et al., 2016, Hadfield-Menell et al., 2017, Singh et al., 2010]. Recent LLM-based methods recast reward design as program search: an LLM proposes executable reward functions from a natural-language task description, each proposed reward is used to train a policy, and the resulting behavior is evaluated by a designer-specified task-level fitness function. The fitness values it produces, which are distinct from the generated training rewards, guide subsequent rounds of LLM proposals. A growing line of work [Ma et al., 2024a, Xie et al., 2024, Hazra et al., 2025] has made reward design more scalable under this formulation.

The limitation is that this fitness function remains static while the search process is sequential. Each proposal is informed by previous outcomes, yet each candidate is evaluated from a fresh policy initialization and the resulting checkpoint is discarded after scoring. The score therefore measures standalone reward quality, asking whether a reward can train a good policy in isolation, rather than whether it advances the policy state produced by previous rewards.

FORGE redefines reward fitness around trajectory-relative progress. Instead of searching for a single best standalone reward, it optimizes an ordered reward trajectory $(R^0, R^1, \ldots)$ that progres-

Submitted to 40th Conference on Neural Information Processing Systems (NeurIPS 2026). Do not distribute.

sively specializes an inherited policy. This trajectory-level view makes the starting policy and its representation load-bearing: downstream reward evolution should specialize reusable capabilities, not spend early generations rediscovering low-level embodiment behavior. FORGE therefore begins with a task-agnostic skill prior learned on the embodiment alone, paralleling goal-conditioned RL [Schaul et al., 2015], world-model pretraining [Hafner et al., 2023], and behavioral foundation models [Touati and Ollivier, 2021] that decouple capability acquisition from task specification. The prior provides reusable behaviors and a feature space in which LLM-generated rewards can be projected into reward-directed skill adaptations.

To realize this reframe, we introduce FORGE (**F**ine-tuning **O**ver **R**eward **G**enerations through **E**volution), which couples an upstream skill prior with downstream reward-policy co-evolution. Each LLM-generated reward function is relabeled on the offline pretraining buffer and projected into the pretrained feature space to obtain a reward-directed skill direction. The corresponding child policy then fine-tunes from its parent checkpoint rather than restarting from scratch, treating each generation as a step in a continual [Khetarpal et al., 2022] reward-driven adaptation that mixes pretrained and online experience analogously to recent offline-to-online RL methods [Nair et al., 2020, Ball et al., 2023, Nakamoto et al., 2023]. The prior supplies the behavioral basis; the ordered reward trajectory supplies the specialization.

We validate FORGE on three tasks that vary in whether upstream pretraining reaches downstream success states. For Humanoid Locomotion and Standup, RND rollouts already visit success states during pretraining — fast-forward gait and upright-balanced postures both fall within unsupervised coverage. For Adroit Hand, upstream exploration is effectively blind to success states: the prior is pretrained almost entirely without observing states that constitute downstream success. This contrast isolates when the skill prior alone is sufficient, when capability must be injected, and whether the ordered trajectory remains load-bearing across both regimes.

(1) A reusable prior guides generational policy evolution. A task-agnostic skill prior provides reusable behavioral structure for downstream evolution, allowing each generation to inherit useful capabilities rather than relearn low-level skills from scratch. This converts pretraining into a one-time upstream cost that can be amortized across downstream tasks sharing the same embodiment.

(2) Expert demonstrations restore missing success capabilities. When unsupervised pretraining does not reach success states, a one-time set of expert demonstrations can supply the missing capability. Unlike prior reward-search methods that rely on *signal-level* feedback at every generation, this intervention operates at the *capability level*, restoring access to successful behaviors before downstream evolution begins.

(3) Trajectory-level fitness captures cumulative policy improvement. Instead of treating each reward candidate as an independent RL problem, FORGE evaluates how rewards shape an inherited policy across generations. This shifts reward search from selecting the best standalone reward to optimizing an ordered reward trajectory, where each generation is judged by whether it advances the cumulative capabilities inherited from its predecessors.

2 Problem Formulation

2.1 Reward Search as Static Optimization

We consider a Markov Decision Process [Sutton and Barto, 1998] without a predefined reward function, $\mathcal{M} \setminus R = (\mathcal{S}, \mathcal{A}, P, \gamma)$. The goal is to find a reward function $R : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ and a corresponding policy $\pi : \mathcal{S} \to \mathcal{A}$ that maximizes a task-level fitness function $F(\pi)$, a black-box evaluation metric known to the system designer but not differentiable or directly usable as a training signal, akin to optimization objectives in AutoRL [Parker-Holder et al., 2022].

REvolve [Hazra et al., 2025] formulates this as evolutionary search over LLM-generated reward functions. Given an LLM reward designer G, the system maintains a population of reward functions across island-based sub-populations. Let $\pi_R = \mathrm{RL}(R; \theta_0)$ denote the policy obtained by training from initialization θ_0 under reward R. Static reward search selects

$$R^* = \arg \max_{R \in \mathrm{range}(G)} F(\pi_R) = \arg \max_{R \in \mathrm{range}(G)} F(\mathrm{RL}(R; \theta_0)) \,.$$

This formulation inherits an assumption shared by Eureka [Ma et al., 2024a], Text2Reward [Xie et al., 2024], and other LLM-based reward search methods: that the mapping $R \mapsto F(\mathrm{RL}(R, \theta_0))$ is well-behaved, and that finding a better reward reliably produces a better policy.

2.2 Reward-Policy Co-Evolution

We extend reward-only evolution to jointly evolve reward functions and policies. Let $\{(R_k^g, \pi_k^g, \sigma_k^g)\}$ denote the population at generation g, where R_k^g is the reward function, π_k^g the policy, and $\sigma_k^g = F(\pi_k^g)$ the fitness of individual k. Each child at generation $g{+}1$ inherits policy weights from a selected parent: $\pi_k^{g+1} = \text{FineTune}(\pi^g_{\text{parent}(k)}, R_k^{g+1})$. Since π_k^{g+1} inherits from a parent and continues training rather than restarting, its fitness $F(\pi_k^{g+1})$ reflects the capabilities accumulated along the entire reward sequence $(R^0, \ldots, R^{g+1})$ that produced π_k^{g+1}, rather than an algebraic sum of per-generation fitness scores. The search therefore optimizes $\pi^* = \arg\max_\pi F(\pi)$ where π emerges from $(R^0, R^1, \ldots, R^{N-1})$, lifting it from a static optimization over reward space into a sequential process over reward-policy trajectories.

3 Method

FORGE operationalizes trajectory-level fitness as a two-stage procedure. First, an upstream pretraining stage learns a task-agnostic skill prior that provides a reusable behavioral basis for the embodiment. Second, a downstream evolutionary stage evaluates LLM-generated rewards by projecting each reward into this basis, fine-tuning an inherited child policy, and scoring the resulting behavior with the task-level fitness function. We describe the prior, the reward-to-skill projection, and the inheritance rule below.

3.1 Task-Agnostic Skill Prior

FORGE begins with a one-time upstream pretraining phase that produces a skill prior $(\phi, \pi_0, \mathcal{B}_{\text{off}})$ shared across downstream tasks on the same embodiment. The feature map $\phi : \mathcal{S} \to \mathbb{R}^d$ is trained with HILP temporal-distance learning [Park et al., 2024], a member of the broader family of temporal contrastive state representations [Eysenbach et al., 2022], to capture reward-agnostic state-space geometry, while $\pi_0(a \mid s, z)$ and a successor-feature network ψ^π [Barreto et al., 2017, 2019] are trained on the same pretraining buffer $\mathcal{B}_{\text{off}}$. The skill space and feature space share the same dimension d, so the inner product below is well-defined. For any reward that is linear in the learned feature space,

$$R_z(s) = \phi(s)^\top z, \qquad \phi(s), z \in \mathbb{R}^d,$$

the action-value estimate factorizes as

$$Q^\pi(s, a; z) = \psi^\pi(s, a; z)^\top z.$$

This factorization provides the interface used by FORGE downstream: LLM-generated rewards can be projected into skill vectors, and the resulting z values condition π_0 or its descendants toward reward-directed behaviors.

3.2 Closed-Form Reward-to-Skill Adaptation

Before fine-tuning a child policy, FORGE maps each LLM-generated reward into the pretrained skill space. Given a reward candidate R, we relabel transitions from the offline buffer $\mathcal{B}_{\text{off}}$ to obtain reward values $r_i = R(s_i, a_i)$. Let Φ denote the feature matrix whose i-th row is $\phi(s_i)^\top$. We infer the skill direction by solving the least-squares projection

$$z^* = \arg\min_z \|\Phi z - r\|_2^2 = (\Phi^\top \Phi)^{-1} \Phi^\top r.$$

The inferred vector z^* initializes the skill input of the child policy before online fine-tuning. This projection gives the inherited policy a reward-directed initialization in the pretrained skill space. Because ϕ is a state-only feature map, projecting an action-dependent reward $R(s_i, a_i)$ onto the row space of Φ averages out action dependencies under the empirical action distribution in $\mathcal{B}_{\text{off}}$. This action-marginalization mismatch is a primary source of the residual approximation error, alongside any nonlinearity of R in ϕ, both of which subsequent online fine-tuning under R corrects.

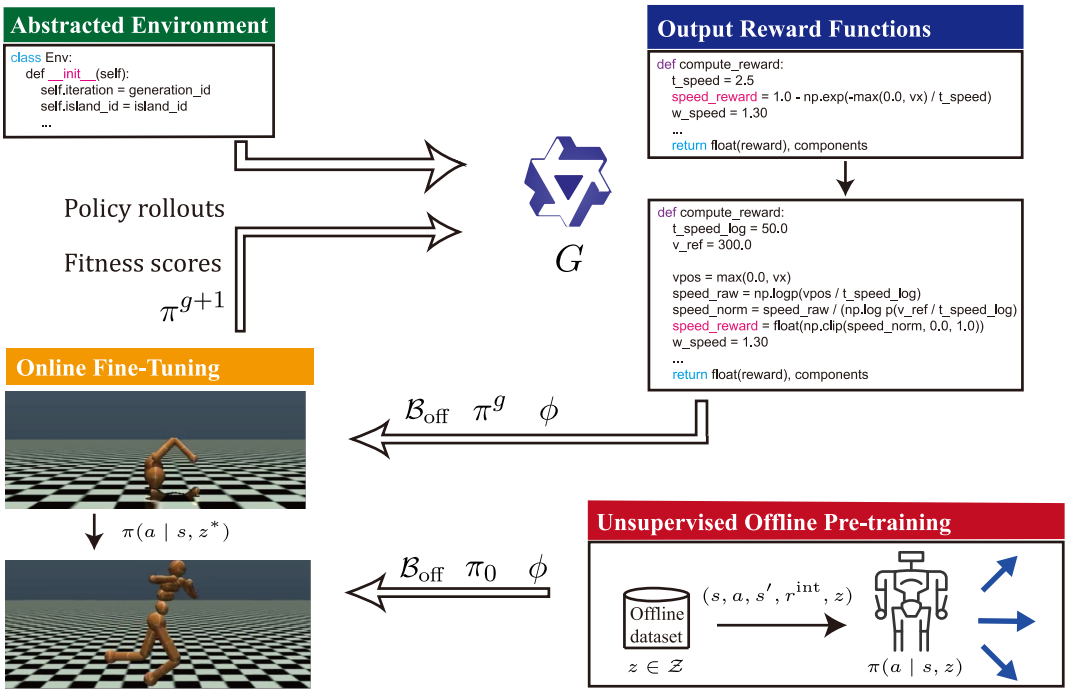

Figure 1: **FORGE pipeline.** Given a task and its environment variables (humanoid locomotion shown), a task-agnostic skill prior $(\phi, \pi_0, \mathcal{B}_{\text{off}})$ is pretrained once. Each generation, the LLM proposes a reward; we infer $z^* = (\Phi^\top \Phi)^{-1} \Phi^\top r$ in closed form and fine-tune a child policy inherited from its parent. Fitness scores the ordered reward-policy trajectory.

3.3 Cross-Generation Policy Inheritance

FORGE evaluates reward candidates through inherited policies rather than freshly initialized ones. At generation $g+1$, each child reward R_k^{g+1} is paired with a parent policy checkpoint from generation g. The child policy is warmstarted with the reward-specific skill vector z^* and then fine-tuned under R_k^{g+1}, a form of cross-task knowledge transfer related to policy distillation [Rusu et al., 2016]. We inherit the final fine-tuned checkpoint at the end of the per-individual budget, not the peak-fitness checkpoint encountered during training.

After fine-tuning, the child policy is evaluated with the task-level fitness function. Since each child is initialized from a parent checkpoint, this fitness is lineage-dependent: it reflects how R_k^{g+1} improves the inherited policy state produced by previous rewards, rather than how the reward performs under fresh training.

We evaluate two parent-selection rules, drawn from the two dominant LLM reward-search frameworks. **FORGE (island)**, the default variant, follows REvolve's [Hazra et al., 2025] island-based design: parent policies are selected within each island-local sub-population by fitness-weighted sampling, maintaining multiple reward-policy lineages in parallel. **FORGE (elite)** follows Eureka's [Ma et al., 2024a] single-population design, where all children inherit from the globally highest-fitness policy.

The complete procedure is provided in Algorithm 1.

4 Experiments

4.1 Experimental Setup

We evaluate FORGE on three MuJoCo environments [Todorov et al., 2012, Tassa et al., 2018]: Humanoid Locomotion, Humanoid Standup, and Adroit Hand [Rajeswaran et al., 2018]. The tasks

Table 1: Summary of observation and action spaces, objectives, and termination criteria for the three RL training tasks.

Task	Observation Space	Action Space	Objective & Termination
Locomotion	$\mathbb{R}^{376}$	continuous, $A \in \mathbb{R}^{17}$	Run forward for 1000 steps
Standup	$\mathbb{R}^{376}$	continuous, $A \in \mathbb{R}^{17}$	Stand up and balance for 1000 steps
Adroit Hand	$\mathbb{R}^{39}$	continuous, $A \in \mathbb{R}^{28}$	Open the door in 400 steps

differ in how much upstream pretraining covers downstream success states: the Humanoid tasks admit reusable locomotion and balance behaviors from unsupervised exploration, whereas Adroit Hand demands sparse contact-rich behaviors that random exploration rarely produces. Table 1 summarizes the three environments; full specifications are in Appendix B.

For the reward designer G, we use Qwen3-Coder (480B-A35B-Instruct) [Qwen Team, 2025]. The model receives a natural-language task description together with selected simulator observation and action variables, shown in Figure 1, and generates executable reward functions. Policies are trained on these generated rewards, while performance is evaluated using separate task-level fitness functions defined in Appendix D.

4.2 Pretraining

The skill prior $(\phi, \pi_0, \mathcal{B}_{\text{off}})$ is pretrained once per embodiment. Because reward-to-skill projection is fit over features from $\mathcal{B}_{\text{off}}$, the buffer determines which downstream reward directions are represented in the prior.

We evaluate three buffer variants: (i) **RND**, using random network distillation rollouts [Burda et al., 2019], an instance of curiosity-driven exploration [Pathak et al., 2017]; (ii) **expert-only**, using expert demonstrations; and (iii) **RND + expert**, combining both sources. For Adroit Hand, expert episodes are drawn from D4RL [Fu et al., 2020]. For Humanoid Locomotion and Standup, expert episodes are collected from converged RND-only FORGE policies.

For each buffer, we train ϕ with HILP temporal-distance learning and jointly fit π_0 and ψ^π for 1M gradient steps. Unless otherwise stated, RND pretraining is the default on tasks where random exploration reaches task-relevant regions. Architecture and hyperparameter details are provided in Appendix B.

4.3 Training Setup

Downstream evolution runs for 7 generations with $K = 16$ reward candidates per generation. Each candidate fine-tunes the inherited skill-conditioned actor and successor-feature critic ψ^π for 500K environment interactions under the projected skill z^*, with the actor taking a deterministic policy-gradient step on $Q(s, a; z^*) = \psi^\pi(s, a; z^*)^\top z^*$ and ψ^π updated by temporal-difference (TD) learning on ϕ-features. Replay batches use a 50:50 mixture of offline transitions from $\mathcal{B}_{\text{off}}$ and newly collected online transitions.

We report mean $\pm$ standard deviation over 5 independent seeds. A single generation, parallelized over 16 candidates on 8 NVIDIA GeForce RTX 4090 GPUs, takes approximately 6 hours per environment.

4.4 Baselines

We compare against two baselines:

- **REvolve-auto (from-scratch evolution).** The automated version of REvolve [Hazra et al., 2025] without human feedback: island-based reward evolution with SAC [Haarnoja et al., 2018] policies trained from scratch for each reward candidate. Within the fully automated regime, REvolve-auto represents the strongest published LLM reward-search baseline, having outperformed Eureka [Ma et al., 2024a] on the shared benchmarks. We re-run REvolve-auto using the same LLM as FORGE.

- **REvolve published numbers and discovered rewards.** Because the original REvolve human-feedback loop requires per-generation human evaluators and cannot be exactly reproduced, we do not treat it as a live algorithmic baseline. We instead use REvolve in two ways. First, on Adroit Hand we cite REvolve's published peak fitness of ~ 0.875 [Hazra et al., 2025] directly as the reference point against which we compare FORGE. Second, where REvolve releases its discovered reward functions (Locomotion and Adroit Hand), we additionally train fresh SAC policies under those rewards using our protocol, isolating the contribution of the discovered reward itself from the human-guided search process that produced it.

4.5 Experimental Questions and Results

We organize the results around three questions aligned with the claims of FORGE: (i) whether inherited reward evolution improves over discard-and-restart baselines, (ii) when the upstream prior requires expert demonstrations, and (iii) whether the skill prior and cross-generation inheritance are both necessary.

Does inherited reward-policy evolution improve compute-efficient reward search? Cross-generation inheritance allows the population to reuse capabilities accumulated under previous rewards, but this reuse is beneficial only when multiple viable lineages are preserved. **FORGE (elite)** inherits from a single globally fittest checkpoint and exhibits a high-variance collapse: a few seeds reach high fitness early, while the remaining seeds lock into poor early lineages and stagnate, leaving the seed-averaged curve below REvolve-auto (Figure 2). In contrast, **FORGE (island)** maintains island-local lineages, preventing any single checkpoint from dominating the population. On Humanoid Locomotion and Humanoid Standup, whose success states are covered by RND pretraining, FORGE (island) exceeds REvolve-auto while using $10\times$ fewer environment interactions per individual. Including the one-time pretraining cost, FORGE uses 57M total environment interactions versus 560M for REvolve-auto.

Adroit Hand exposes a boundary condition: when upstream coverage misses success states, inheritance alone cannot rescue the population. Under RND-only pretraining, both FORGE variants stagnate, and inspecting $\mathcal{B}_{\text{off}}$ confirms the proximate cause: it contains zero door-opening trajectories (Appendix C). The inherited policies therefore have no door-success behavior to specialize from, so preserving lineages alone is insufficient. This identifies upstream coverage as the binding constraint, which we test next through expert-augmented pretraining.

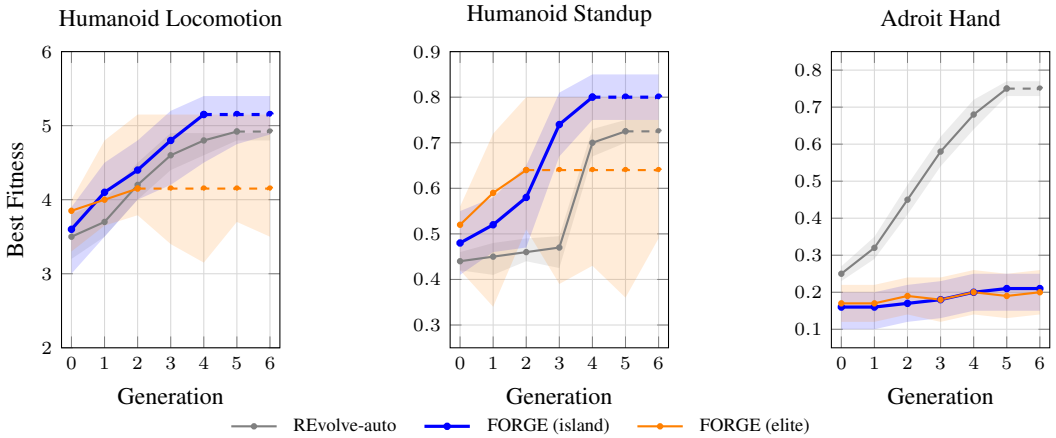

Figure 2: **Compute-efficient reward search.** Cumulative best fitness vs. generation. FORGE (island) exceeds REvolve-auto on Humanoid Locomotion and Standup at $10\times$ less per-individual compute; FORGE (elite) collapses under global-best inheritance. Solid: observed; dashed: projected plateau. Shaded: std across seeds.

When does expert injection help, and when does it hurt? Expert demonstrations rescue Adroit Hand but degrade Humanoid Locomotion; preliminary Standup runs exhibit the same pattern (Fig-

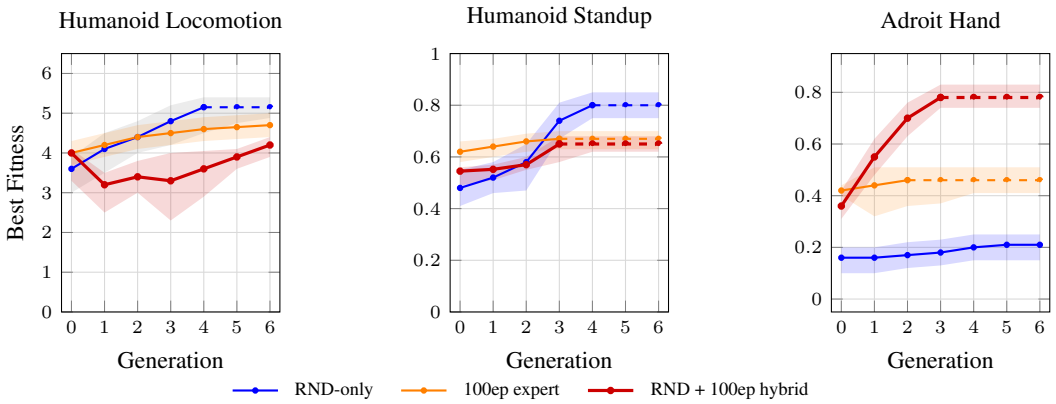

Figure 3: **Expert injection rescues or harms depending on the binding constraint.** Cumulative best fitness vs. generation under three buffers: RND-only (blue), expert-only (orange), and RND + 100ep hybrid (red). Adroit Hand uses 100 `door-expert-v1` ep; Humanoid uses 100 ep collected from converged RND-only FORGE policies. Full `door-expert-v1`: Fig. 4. Shaded: std across seeds.

ure 3). This split supports a two-condition view: downstream evolution requires both *capability*—the pretrained policy reaching task-relevant success states—and *task-relevant trajectory diversity*—its rollouts varying enough for different LLM rewards to specialize the policy differently. RND supplies diversity but supplies capability only when random exploration reaches success states; expert data supplies capability but retains diversity only when demonstrations are not near-deterministic. Both conditions reduce to one object: the column space of the pretrained feature map ϕ.

Every LLM reward reaches the child policy as the projected skill $z^* = (\Phi^\top \Phi)^{-1} \Phi^\top r$, which then conditions the inherited skill-conditioned actor and successor-feature critic ψ^π. Both were jointly trained on $\mathcal{B}_{\text{off}}$, so the buffer fixes both π_0's behavior repertoire and the value landscape ψ^π assigns over it. Downstream fine-tuning is a skill-conditioned successor-feature actor-critic update, drawing 50:50 minibatches from $\mathcal{B}_{\text{off}}$ that continually re-anchor the child to the buffer's distribution. Capability and diversity therefore reduce to two properties of $\mathcal{B}_{\text{off}}$ as the prior sees it: whether it contains success states at all, and how broadly π_0's induced rollouts span them.

On Adroit Hand, $\mathcal{B}_{\text{off}}$ under RND contains no successful trajectories (Appendix C), so π_0 never sees a door-opening transition during pretraining and stagnates near the random baseline regardless of z. Adding 100 `door-expert-v1` episodes introduces success states into $\mathcal{B}_{\text{off}}$, so π_0's rollouts can now reach the latch-and-push regime, and evolution lifts to 0.78 fitness, 89.1% of REvolve's published 0.875 peak, without per-generation human feedback. Expert-only pretraining caps at the imitation ceiling: the narrow, near-deterministic D4RL demonstrations restrict π_0's rollouts to a thin imitation manifold, so distinct LLM rewards cannot drive the policy toward meaningfully different door-opening strategies. Pretraining π_0 on the full `door-expert-v1` dataset already converges at gen-0 (Figure 4), confirming capability as the binding constraint.

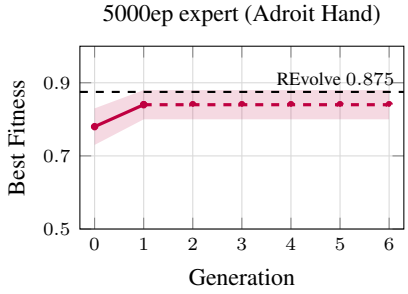

Figure 4: **Capability-ceiling check.** Upper bound when expert data is unconstrained: pretraining π_0 on the full 5000-ep `door-expert-v1` reaches 0.78 at gen-0 and saturates at 0.84 by gen-1.

On both Humanoid tasks, the binding constraint reverses. RND alone covers task-relevant states broadly enough that π_0's rollouts span a wide range of walking and balance behaviors, so distinct LLM rewards drive the policy in distinct directions. Augmenting RND with 100 ep from converged FORGE policies biases $\mathcal{B}_{\text{off}}$ toward a narrow band of near-optimal behaviors and concentrates π_0's rollout distribution around that mode; different LLM rewards then push the inherited policy along nearly overlapping directions, and hybrid underperforms RND-only. Expert-only fitness

sits slightly below RND-only because expert episodes pooled across multiple FORGE seeds retain some between-trajectory variation, but still cover a narrower behavior repertoire than the RND-only buffer.

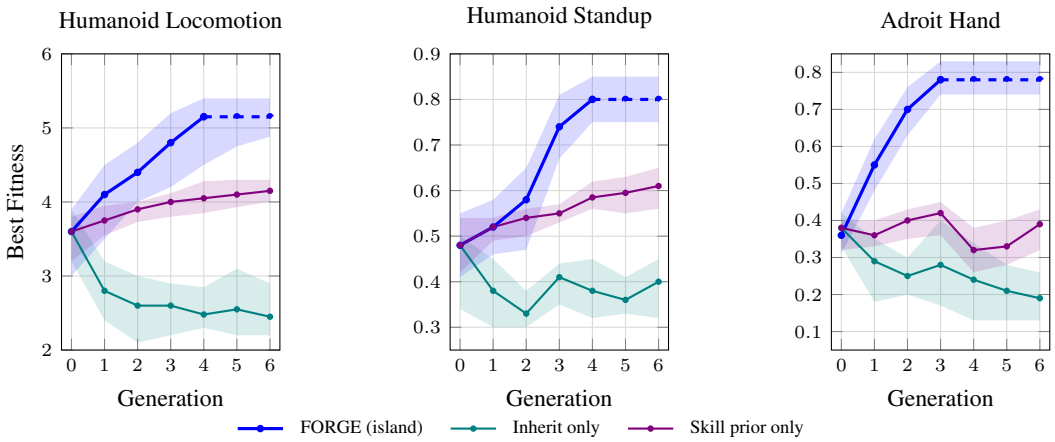

Figure 5: **Both the skill prior and cross-generation inheritance are required.** RND-only pretraining. Removing pretrained ϕ (*Inherit only*) or cross-generation inheritance (*Skill prior only*) underperforms the full system on Humanoid tasks; all variants stagnate on Adroit Hand (capability-limited).

Does trajectory-level fitness capture cumulative policy improvement? Standard LLM reward search trains a fresh policy under each reward and scores it. FORGE redefines fitness as a reward's contribution to an inherited reward-policy trajectory: a reward is fit only insofar as it advances a policy already shaped by previous rewards. We ask whether this trajectory-level fitness definition is load-bearing, or whether FORGE's gains can be reduced to simpler explanations: a strong pretrained prior, checkpoint inheritance alone, or a single best discovered reward.

Figure 5 rules out the first two explanations. Removing the pretrained representation (*Inherit only*) leaves only checkpoint continuity from the fittest parent. Fitness drifts downward across generations: without a task-agnostic representation, inherited checkpoints primarily preserve their own state distributions rather than providing reusable structure under new rewards. Inheritance without representation is therefore a liability rather than an asset.

Removing cross-generation inheritance (*Skill prior only*) restarts every generation from π_0. Gen-0 performance is strong because the pretrained prior provides a broad behavioral repertoire, but the population cannot accumulate the per-reward fine-tuning gains of later generations and plateaus below FORGE. The prior provides the initial repertoire; inheritance converts it into compounding progress.

Table 2: FORGE co-evolved fitness (top, bold; mean±std, 5 seeds) vs. best-reward retraining at three budgets (Δ from FORGE, negative = worse). No single-reward retraining matches the co-evolved peak.

Configuration	Init	Compute	Locomotion	Standup	Adroit Hand
FORGE	Pretrained π_0	57M	$\mathbf{5.1 \pm 0.3}$	$\mathbf{0.80 \pm 0.05}$	$\mathbf{0.78 \pm 0.05}$
Best reward	Pretrained π_0	0.5M	-0.8	-0.26	-0.28
Best reward	Pretrained π_0	3.5M	-0.5	-0.20	-0.23
Best reward	Random θ_0	5M	-0.6	-0.18	-0.13

Table 2 rules out the single-reward explanation. We retrain the highest-fitness reward discovered along the best-performing lineage at multiple budgets, including from-scratch training with an order of magnitude more per-individual compute. None matches the co-evolved peak across the three environments. The reward that performs best within the lineage does not generalize beyond it: under

trajectory-level fitness, a reward is selected because it advances an inherited policy, not because it is the best standalone training signal.

5 Related Work

LLM-based reward design. A growing line of work uses LLMs to generate or evolve reward code: Eureka [Ma et al., 2024a] pairs LLM generation with evolutionary search; Text2Reward [Xie et al., 2024] translates task descriptions into dense reward code; Language to Rewards [Yu et al., 2023] and Reward Design with LMs [Kwon et al., 2023] use language grounding for reward specification; DrEureka [Ma et al., 2024b] extends LLM rewards to sim-to-real transfer; and REvolve [Hazra et al., 2025] introduces island-based sub-populations. All of these train and discard policies each generation, treating the reward as the sole output. Non-LLM reward evolution [Faust et al., 2019, Co-Reyes et al., 2021] likewise optimizes reward structure in isolation from policy continuation.

Reward-policy co-evolution. Population-based training [Jaderberg et al., 2017] and evolution strategies [Salimans et al., 2017, Such et al., 2017] co-evolve hyperparameters or policy parameters. The closest concurrent work, ROSKA [Huang et al., 2025], also inherits policies across reward generations, but adapts via per-reward gradient updates and lacks a task-agnostic anchor. FORGE instead adapts via closed-form successor-feature projection anchored to a pretrained ϕ, and uses island-local rather than global-elite inheritance.

Unsupervised RL and successor features. Successor features [Barreto et al., 2017, Borsa et al., 2019, Barreto et al., 2020] decompose value into task-independent features and task-dependent weights, enabling transfer across reward functions. Unsupervised skill discovery (DIAYN [Eysenbach et al., 2019], APT [Liu and Abbeel, 2021], and the URLB benchmark [Laskin et al., 2021]) learns reusable behaviors without task rewards, and HILP [Park et al., 2024] learns temporal-distance features that capture state-space geometry; Unsupervised-to-Online RL [Kim et al., 2024] combines these for offline-to-online transfer. We repurpose this framework for rapid reward adaptation inside an evolutionary loop.

6 Discussion and Future Work

Limitations. FORGE assumes a fixed embodiment whose pretrained ϕ covers task-relevant states; cross-embodiment transfer is out of scope. Expert injection is task-dependent: it rescues capability-limited tasks (Adroit Hand) but degrades tasks where RND already supplied both capability and diversity (Locomotion, Standup), and diagnosing the binding constraint a priori currently requires running the ablation. Run-to-run variance is higher than from-scratch evolution because LLM-reward stochasticity compounds with inherited-policy drift across generations.

Future work. Three directions follow directly. (i) The post-peak plasticity loss matches the *primacy bias* phenomenon in deep RL [Nikishin et al., 2022, Lyle et al., 2023, 2024, Sokar et al., 2023]; co-evolutionspecific mitigations include generation-level rollback to the last improving checkpoint and online ϕ expansion (continuing ϕ training on new rollouts between generations). (ii) Whether the implicit curriculum uncovered by search can be designed explicitly to recover the benefit without the search cost remains open. Cross-embodiment transfer requires embodiment-agnostic upstream pretraining [Collaboration et al., 2023] and is a separate extension.

7 Conclusion

We reframe LLM-driven reward search around the upstream–downstream factorization of modern RL. A one-time task-agnostic skill prior is pretrained on the embodiment alone, and a downstream evolutionary loop then projects each LLM-generated reward into the prior's skill space and inherits its policy weights across generations. FORGE thereby replaces from-scratch policy training on every reward candidate with closed-form skill projection followed by skill-conditioned fine-tuning, and shows that ordered reward trajectories, not single best rewards, are the load-bearing object of evolutionary reward search. The same factorization redefines the role of human input: instead of

per-generation feedback on each candidate's signal, a one-time set of expert demonstrations at pre-training supplies the missing capability when the upstream prior rarely observes success states.

References

Dario Amodei, Chris Olah, Jacob Steinhardt, Paul Christiano, John Schulman, and Dan Mané. Concrete problems in ai safety, 2016. URL https://arxiv.org/abs/1606.06565.

Philip J. Ball, Laura Smith, Ilya Kostrikov, and Sergey Levine. Efficient online reinforcement learning with offline data. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, Proceedings of Machine Learning Research, pages 1577–1594. PMLR, 2023. URL https://proceedings.mlr.press/v202/ball23a.html.

André Barreto, Will Dabney, Rémi Munos, Jonathan J. Hunt, Tom Schaul, David Silver, and Hado van Hasselt. Successor features for transfer in reinforcement learning. In *Advances in Neural Information Processing Systems*, 2017.

André Barreto, Diana Borsa, John Quan, Tom Schaul, David Silver, Matteo Hessel, Daniel J. Mankowitz, Augustin Zídek, and Rémi Munos. Transfer in deep reinforcement learning using successor features and generalised policy improvement. *CoRR*, abs/1901.10964, 2019. URL http://arxiv.org/abs/1901.10964.

André Barreto, Shaobo Hou, Diana Borsa, David Silver, and Doina Precup. Fast reinforcement learning with generalized policy updates. *Proceedings of the National Academy of Sciences*, 117 (48):30079–30087, 2020.

Diana Borsa, André Barreto, John Quan, Daniel J. Mankowitz, Hado van Hasselt, Rémi Munos, David Silver, and Tom Schaul. Universal successor features approximators. In *International Conference on Learning Representations*, 2019.

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

John D. Co-Reyes, Yingjie Miao, Daiyi Peng, Esteban Real, Quoc V. Le, Sergey Levine, Honglak Lee, and Aleksandra Faust. Evolving reinforcement learning algorithms. In *International Conference on Learning Representations*, 2021.

Open X.-Embodiment Collaboration, Abhishek Padalkar, Acorn Pooley, Ajinkya Jain, Alex Bewley, Alexander Herzog, Alex Irpan, Alexander Khazatsky, Anant Raj, Anikait Singh, Anthony Brohan, Antonin Raffin, Ayzaan Wahid, Ben Burgess-Limerick, Beomjoon Kim, Bernhard Schölkopf, Brian Ichter, Cewu Lu, Charles Xu, Chelsea Finn, Chenfeng Xu, Cheng Chi, Chenguang Huang, Christine Chan, Chuer Pan, Chuyuan Fu, Coline Devin, Danny Driess, Deepak Pathak, Dhruv Shah, Dieter Büchler, Dmitry Kalashnikov, Dorsa Sadigh, Edward Johns, Federico Ceola, Fei Xia, Freek Stulp, Gaoyue Zhou, Gaurav S. Sukhatme, Gautam Salhotra, Ge Yan, Giulio Schiavi, Gregory Kahn, Hao Su, Haoshu Fang, Haochen Shi, Heni Ben Amor, Henrik I. Christensen, Hiroki Furuta, Homer Walke, Hongjie Fang, Igor Mordatch, Ilija Radosavovic, and et al. Open x-embodiment: Robotic learning datasets and RT-X models. *CoRR*, abs/2310.08864, 2023. doi: 10.48550/ARXIV.2310.08864. URL https://doi.org/10.48550/arXiv.2310.08864.

Benjamin Eysenbach, Abhishek Gupta, Julian Ibarz, and Sergey Levine. Diversity is all you need: Learning skills without a reward function. In *International Conference on Learning Representations*, 2019.

Benjamin Eysenbach, Tianjun Zhang, Sergey Levine, and Ruslan Salakhutdinov. Contrastive learning as goal-conditioned reinforcement learning. In Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, 2022. URL http://papers.nips.cc/paper_files/paper/2022/hash/e7663e974c4ee7a2b475a4775201ce1f-Abstract-Conference.html.

Aleksandra Faust, Anthony G. Francis, and Dar Mehta. Evolving rewards to automate reinforcement learning, 2019. URL https://arxiv.org/abs/1905.07628.

Justin Fu, Aviral Kumar, Ofir Nachum, George Tucker, and Sergey Levine. D4RL: Datasets for deep data-driven reinforcement learning, 2020. URL https://arxiv.org/abs/2004.07219.

Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International Conference on Machine Learning*, 2018.

Dylan Hadfield-Menell, Smitha Milli, Pieter Abbeel, Stuart J. Russell, and Anca D. Dragan. Inverse reward design. In *Advances in Neural Information Processing Systems*, 2017.

Danijar Hafner, Jurgis Pasukonis, Jimmy Ba, and Timothy P. Lillicrap. Mastering diverse domains through world models. *CoRR*, abs/2301.04104, 2023. doi: 10.48550/ARXIV.2301.04104. URL https://doi.org/10.48550/arXiv.2301.04104.

Rishi Hazra, Alkis Sygkounas, Andreas Persson, Amy Loutfi, and Pedro Zuidberg Dos Martires. REvolve: Reward evolution with large language models using human feedback. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=cJPUpL8mOw.

Changxin Huang, Yanbin Chang, Junfan Lin, Junyang Liang, Runhao Zeng, and Jianqiang Li. Efficient language-instructed skill acquisition via reward-policy co-evolution. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2025.

Max Jaderberg, Valentin Dalibard, Simon Osindero, Wojciech M. Czarnecki, Jeff Donahue, Ali Razavi, Oriol Vinyals, Tim Green, Iain Dunning, Karen Simonyan, Chrisantha Fernando, and Koray Kavukcuoglu. Population based training of neural networks, 2017. URL https://arxiv.org/abs/1711.09846.

Khimya Khetarpal, Matthew Riemer, Irina Rish, and Doina Precup. Towards continual reinforcement learning: A review and perspectives. *J. Artif. Intell. Res.*, 75:1401–1476, 2022. doi: 10.1613/JAIR.1.13673. URL https://doi.org/10.1613/jair.1.13673.

Junsu Kim, Seohong Park, and Sergey Levine. Unsupervised-to-online reinforcement learning, 2024. URL https://arxiv.org/abs/2408.14785.

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

Michael Laskin, Denis Yarats, Hao Liu, Kimin Lee, Albert Zhan, Kevin Lu, Catherine Cang, Lerrel Pinto, and Pieter Abbeel. URLB: Unsupervised reinforcement learning benchmark. In *Advances in Neural Information Processing Systems*, 2021.

Hao Liu and Pieter Abbeel. Behavior from the void: Unsupervised active pre-training. In *Advances in Neural Information Processing Systems*, 2021.

Clare Lyle, Zeyu Zheng, Evgenii Nikishin, Bernardo Ávila Pires, Razvan Pascanu, and Will Dabney. Understanding plasticity in neural networks. In *International Conference on Machine Learning*, 2023.

Clare Lyle, Zeyu Zheng, Khimya Khetarpal, Hado van Hasselt, Razvan Pascanu, James Martens, and Will Dabney. Disentangling the causes of plasticity loss in neural networks. In *Conference on Lifelong Learning Agents*, 2024.

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*, 2024a.

Yecheng Jason Ma, William Liang, Hung-Ju Wang, Yuke Zhu, Linxi Fan, Osbert Bastani, and Dinesh Jayaraman. DrEureka: Language model guided sim-to-real transfer. In *Robotics: Science and Systems*, 2024b.

Ashvin Nair, Murtaza Dalal, Abhishek Gupta, and Sergey Levine. Accelerating online reinforcement learning with offline datasets. *CoRR*, abs/2006.09359, 2020. URL https://arxiv.org/abs/2006.09359.

Mitsuhiko Nakamoto, Simon Zhai, Anikait Singh, Max Sobol Mark, Yi Ma, Chelsea Finn, Aviral Kumar, and Sergey Levine. Cal-ql: Calibrated offline RL pre-training for efficient online fine-tuning. In Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine, editors, *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, Decem-*

ber 10 - 16, 2023, 2023. URL http://papers.nips.cc/paper_files/paper/2023/hash/c44a04289beaf0a7d968a94066a1d696-Abstract-Conference.html.

Evgenii Nikishin, Max Schwarzer, Pierluca D'Oro, Pierre-Luc Bacon, and Aaron Courville. The primacy bias in deep reinforcement learning. In *International Conference on Machine Learning*, 2022.

Seohong Park, Tobias Kreiman, and Sergey Levine. Foundation policies with Hilbert representations. In *International Conference on Machine Learning*, 2024.

Jack Parker-Holder, Raghu Rajan, Xingyou Song, André Biedenkapp, Yingjie Miao, Theresa Eimer, Baohe Zhang, Vu Nguyen, Roberto Calandra, Aleksandra Faust, Frank Hutter, and Marius Lindauer. Automated reinforcement learning (autorl): A survey and open problems. *J. Artif. Intell. Res.*, 74:517–568, 2022. doi: 10.1613/JAIR.1.13596. URL https://doi.org/10.1613/jair.1.13596.

Deepak Pathak, Pulkit Agrawal, Alexei A. Efros, and Trevor Darrell. Curiosity-driven exploration by self-supervised prediction. In *2017 IEEE Conference on Computer Vision and Pattern Recognition Workshops, CVPR Workshops 2017, Honolulu, HI, USA, July 21-26, 2017*, pages 488–489. IEEE Computer Society, 2017. doi: 10.1109/CVPRW.2017.70. URL https://doi.org/10.1109/CVPRW.2017.70.

Qwen Team. Qwen3 technical report, 2025. URL https://arxiv.org/abs/2505.09388.

Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 22(268):1–8, 2021.

Aravind Rajeswaran, Vikash Kumar, Abhishek Gupta, Giulia Vezzani, John Schulman, Emanuel Todorov, and Sergey Levine. Learning complex dexterous manipulation with deep reinforcement learning and demonstrations. In *Robotics: Science and Systems*, 2018.

Andrei A. Rusu, Sergio Gomez Colmenarejo, Caglar Gulcehre, Guillaume Desjardins, James Kirkpatrick, Razvan Pascanu, Volodymyr Mnih, Koray Kavukcuoglu, and Raia Hadsell. Policy distillation. In *International Conference on Learning Representations*, 2016.

Tim Salimans, Jonathan Ho, Xi Chen, Szymon Sidor, and Ilya Sutskever. Evolution strategies as a scalable alternative to reinforcement learning, 2017. URL https://arxiv.org/abs/1703.03864.

Tom Schaul, Daniel Horgan, Karol Gregor, and David Silver. Universal value function approximators. In Francis R. Bach and David M. Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, JMLR Workshop and Conference Proceedings, pages 1312–1320. JMLR.org, 2015. URL http://proceedings.mlr.press/v37/schaul15.html.

Satinder Singh, Richard L. Lewis, Andrew G. Barto, and Jonathan Sorg. Intrinsically motivated reinforcement learning: An evolutionary perspective. *IEEE Transactions on Autonomous Mental Development*, 2(2):70–82, 2010.

Ghada Sokar, Rishabh Agarwal, Pablo Samuel Castro, and Utku Evci. The dormant neuron phenomenon in deep reinforcement learning. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, Proceedings of Machine Learning Research, pages 32145–32168. PMLR, 2023. URL https://proceedings.mlr.press/v202/sokar23a.html.

Felipe Petroski Such, Vashisht Madhavan, Edoardo Conti, Joel Lehman, Kenneth O. Stanley, and Jeff Clune. Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning, 2017. URL https://arxiv.org/abs/1712.06567.

Richard S. Sutton and Andrew G. Barto. *Reinforcement learning - an introduction*. Adaptive computation and machine learning. MIT Press, 1998. ISBN 978-0-262-19398-6. URL http://www.incompleteideas.net/book/first/the-book.html.

Yuval Tassa, Yotam Doron, Alistair Muldal, Tom Erez, Yazhe Li, Diego de Las Casas, David Budden, Abbas Abdolmaleki, Josh Merel, Andrew Lefrancq, Timothy P. Lillicrap, and Martin A. Riedmiller. Deepmind control suite. *CoRR*, abs/1801.00690, 2018. URL http://arxiv.org/

abs/1801.00690.

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*, 2012.

Ahmed Touati and Yann Ollivier. Learning one representation to optimize all rewards. In Marc'Aurelio Ranzato, Alina Beygelzimer, Yann N. Dauphin, Percy Liang, and Jennifer Wortman Vaughan, editors, *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, pages 13–23, 2021. URL https://proceedings.neurips.cc/paper/2021/hash/003dd617c12d444ff9c80f717c3fa982-Abstract.html.

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. In *Conference on Robot Learning*, 2023.

Appendix

A Algorithm

Algorithm 1 specifies the FORGE loop in full. After a one-time unsupervised pretraining phase that produces a skill encoder ϕ, a skill-conditioned policy π_0, and a pretrained buffer $\mathcal{B}_{\text{off}}$, each generation has the LLM propose reward functions, infers the optimal skill z^* for each via least squares, fine-tunes the inherited parent policy on a mix of the pretrained buffer and fresh rollouts, and updates the per-island archive by fitness.

Algorithm 1 Co-Evolving Rewards and Policies with Unsupervised Skill Priors

Require: Fitness function F, reward designer G (LLM), pretrained agent $(\phi, \pi_0, \mathcal{B}_{\text{off}})$
Hyperparameters: Generations N, individuals per generation K, sub-populations I, island capacity C, migration prob. p_{mig}

Pretrain ϕ via HILP on $\mathcal{B}_{\text{off}}$; obtain skill-conditioned π_0 ▷ *Unsupervised pretraining (one-time)*
for generation $g = 0, 1, \ldots, N-1$ **do**
  for $k = 1, \ldots, K$ **do**
    if $g = 0$ **then**
      $R_k \leftarrow G(\text{task description})$; $\pi \leftarrow \pi_0$ ▷ *LLM generates initial rewards*
      Assign k to random sub-population P ▷ *Island assignment*
    else
      Sample sub-population P by fitness-weighted sampling ▷ *Island selection*
      Sample parents $S \subset D[P]$; $\pi_{\text{parent}} \leftarrow \pi_{\arg\max_{s \in S} \sigma_s}$
      $R_k \leftarrow G(S)$ ▷ *LLM mutates or crosses over*
      $\pi \leftarrow \pi_{\text{parent}}$ ▷ *Inherit parent policy weights*
    end if
    Relabel $\mathcal{B}_{\text{off}}$ with R_k; $z^* \leftarrow (\Phi^\top \Phi)^{-1} \Phi^\top r_k$ ▷ *Zero-shot skill inference*
    $\pi \leftarrow \text{FineTune}(\pi, R_k, z^*, \mathcal{B}_{\text{off}} \cup \mathcal{B}_{\text{on}})$ ▷ *Mixed offline-online fine-tuning*
    $\sigma_k \leftarrow F(\pi)$ ▷ *Fitness evaluation*
    if $g = 0$ **or** $\sigma_k \geq \bar{\sigma}^P$ **then**
      $D[P] \leftarrow D[P] \cup \{(R_k, \pi, \sigma_k)\}$ ▷ *Accept if above island mean*
      if $|D[P]| > C$ **then**
        $D[P] \leftarrow D[P] \setminus \arg\min_{(R,\pi,\sigma) \in D[P]} \sigma$ ▷ *Evict lowest-fitness if over capacity*
      end if
    end if
  end for
  if rand() $< p_{\text{mig}}$ **and** $|I| > 1$ **then**
    Reset weaker half of islands; seed with migrants from fitter islands ▷ *Island migration*
  end if
end for
return $\pi^* = \arg\max_\sigma D$

B Environments

This section specifies, for each of the three benchmark environments, the observation and action spaces, termination conditions, and SAC training hyperparameters used by every method we compare. A final subsection lists the skill-prior pretraining hyperparameters shared by FORGE (island) and FORGE (elite).

B.1 Humanoid Locomotion

Description. A 3D bipedal humanoid with a torso, two arms (2 segments each), and two legs (3 segments each) must walk forward as fast as possible.

Observation space. 376-dimensional continuous vector:

- Torso z-height, orientation quaternion (4), 17 joint angles (22 dims)
- Torso linear velocity (3), angular velocity (3), 17 joint velocities (23 dims)
- Center-of-mass velocities for 14 body parts (84 dims)
- Actuator forces (23 dims)
- External contact forces for 14 body parts (84 dims)
- Cinert (140 dims)

Action space. 17-dimensional continuous, $\in [-0.4, 0.4]$ Nm. Controls torques at hinge joints: abdomen (3), hips (6), knees (2), shoulders (4), elbows (2).

Termination. Episode truncated at 1000 steps. Terminated if torso height leaves $[1.0, 2.0]$ m (when `terminate_when_unhealthy=True`).

Training. SAC [Haarnoja et al., 2018] via Stable-Baselines3 [Raffin et al., 2021]. MlpPolicy with 2×256 hidden layers, learning rate 3×10^{-4}, batch size 256, $\gamma = 0.99$, soft update $\tau = 0.005$. Each individual trains for 500K steps (FORGE) or 5M steps (REvolve baseline) in chunks of 5K steps.

B.2 Humanoid Standup

Description. Same body as Humanoid Locomotion. The agent starts supine (torso $z \approx 0.2$ m, quaternion $[0.707, 0.707, 0, 0]$) and must stand up and maintain balance.

Observation space. 376-dimensional (same structure as Humanoid Locomotion). Key indices: obs[0] = torso z-height, obs[1:5] = orientation quaternion, obs[24] = vertical velocity, obs[25:28] = angular velocity.

Action space. Same as Humanoid Locomotion (17-dimensional, $\in [-0.4, 0.4]$ Nm).

Termination. Truncated at 1000 steps. Conditional termination: once the peak torso height exceeds 0.35 m, falling more than 0.2 m below the peak terminates the episode. An environment-level alive bonus scales linearly from 0 at $z = 0.2$ m to 5.0 at $z = 0.8$ m.

Training. Same SAC configuration as Humanoid Locomotion. 500K steps per individual.

B.3 Adroit Hand

Description. A 28-DoF anthropomorphic hand (24 finger + 4 arm DOF) must open a door by rotating the latch and pushing it open. The latch has significant dry friction and bias torque.

Observation space. 39-dimensional continuous vector:

- Joint angles: arm (3) + wrist (2) + fingers (22) = 27 dims (one of the 28 hand DOFs is not exposed in the observation; the action space still controls all 28)
- Latch angle (1), door hinge angle (1)
- Palm position (3), handle position (3), palm–handle delta (3)
- Door open indicator (1): $+1$ if door > 1.0 rad, -1 otherwise

Action space. 28-dimensional continuous, $\in [-1, 1]$, mapped to absolute angular positions of hand joints.

Termination. Truncated at 400 steps. Terminated (success) when door hinge angle ≥ 1.35 rad.

Training. Same SAC configuration as above. Frame skip = 5 (MuJoCo timestep 0.002 s $\rightarrow$ dt = 0.01 s). The default 100-episode hybrid condition combines RND exploration with 25 `door-human-v1` + 100 `door-expert-v1` expert demonstrations from D4RL [Fu et al., 2020]. The full `door-expert-v1` dataset ($\sim$5000 episodes) is used only in the separate ablation of Figure 4.

B.4 Skill Prior Pretraining Hyperparameters

The pretrained skill prior used by FORGE (island) and FORGE (elite) uses the following hyperparameters.

Humanoid (Locomotion and Standup).

- Skill dimension $d = 50$ (HILP $\phi(s) \in \mathbb{R}^{50}$ matches the skill dimension by construction), hidden dimension 1024 (actor, SF, and ϕ networks)
- Learning rate 1×10^{-4}, ϕ learning rate 5×10^{-4}, batch size 1024

- SF soft update $\tau = 0.01$, discount $\gamma = 0.98$
- Offline/online mix ratio: 0.5 (50% pretrained buffer, 50% online experience)
- Skill resampling every 300 steps, covariance update every 1000 steps
- Action noise stddev: 0.2, clip: 0.3
- Pretraining budget: 1M steps; fine-tuning budget per individual: 500K steps

Adroit Hand. Uses reduced network dimensions to match the smaller observation space (full values in released code). Pretraining budget: 1M gradient steps for the 100-episode expert-hybrid condition.

Hardware. All pretraining and evolution runs were performed on a single NVIDIA GPU (32 GB VRAM).

C Adroit Hand Pretraining Buffer Coverage

To ground the capability claim in Section 4.5, we measure how often each Adroit Hand pretraining buffer reaches a door-opening state (door_hinge ≥ 1.35 rad). Table 3 reports per-buffer success-state coverage across the three pretraining variants used in the expert-injection ablation (Figure 3).

Table 3: Door-opening coverage of the three Adroit Hand pretraining buffers. A successful episode is one reaching door_hinge ≥ 1.35 rad.

Buffer	Episodes	Successes	Max door_hinge (rad)
RND-only	10000	0	0.45
100ep expert (`door-expert-v1`)	100	98	1.61
RND + 100ep hybrid	10100	99	1.65

The RND-only buffer contains zero door-opening episodes, confirming that random network distillation in the 28-DoF action space does not reach the latch-release-and-push regime within our pretraining budget. Adding 100 `door-expert-v1` demonstrations injects exactly enough success states for downstream evolution to specialize from; the expert-only variant supplies success states without RND-driven diversity, producing the imitation-ceiling failure mode reported in Section 4.5.

D Fitness Functions

Fitness functions are external evaluation metrics that rank individuals during evolutionary selection; they are kept distinct from the LLM-generated training rewards so that selection pressure cannot be gamed by reward shaping. Each fitness below captures task-relevant performance with quantities the LLM does not control.

Humanoid Locomotion. Fitness is the average forward velocity (m/s) over an evaluation episode. Let v_t denote the instantaneous forward velocity at step t and T the episode length (truncated at 1000 steps):

$$\sigma = \frac{1}{T} \sum_{t=1}^{T} v_t.$$

We report the mean of σ over 5 evaluation episodes per seed.

Humanoid Standup. Standup needs a continuous fitness: forward velocity is meaningless, and a binary stood-up indicator would give evolution almost no gradient between generations. We define $\sigma \in [0, 1]$ as the product of two sub-scores evaluated over the last 100 steps of each episode (or before early termination), each clipped to $[0, 1]$:

$$H = \text{clip}\big(\bar{z}_{\text{torso}} / z_{\text{stand}},\ 0,\ 1\big) \qquad \text{(height ratio)}$$

$$U = \text{clip}\big(\bar{u},\ 0,\ 1\big) \qquad \text{(uprightness)}$$

where $\bar{z}_{\text{torso}}$ is the average torso height over the evaluation window, $z_{\text{stand}} = 1.4\,\text{m}$ is the nominal standing height, and $u_t = 1 - 2(q_{x,t}^2 + q_{y,t}^2)$ is the z-component of the torso's up-vector (1.0 upright, 0.0 lying flat) with $\bar{u}$ its window average. The fitness is the product

$$\sigma = H \times U. \tag{1}$$

The multiplicative form requires both height and uprightness without thresholds or hand-tuned weights: an agent that reaches full height but is inverted scores near zero, as does one that is upright but has not risen. Lying on the back ($H \approx 0.14, U = 0$) yields $\sim$0; standing upright ($H \approx 0.9$, $U \approx 0.85$) yields $\sim$0.77; a perfect standup approaches 1.0.

The environment also applies conditional termination: once the torso rises meaningfully above the ground, falling back ends the episode. This implicit alive signal (analogous to locomotion's `terminate_when_unhealthy`) discourages collapse without requiring the LLM to encode fall penalties explicitly.

Adroit Hand. The door-opening outcome is inherently binary, and no meaningful dense proxy exists: partial handle rotation does not reliably indicate progress toward opening. We therefore score successful episodes by completion speed and assign zero to all failures:

$$\sigma_{\text{episode}} = \begin{cases} -\text{steps}/700 + 75/70 & \text{if success} \\ 0 & \text{otherwise} \end{cases}$$

where steps is the number of timesteps until the door opens. The overall fitness is the mean of σ_{episode} across evaluation episodes, so non-succeeding agents receive exactly zero and successful agents are differentiated by speed.

E Training and Ablation Configurations

This appendix lists the five configurations used in the main results: two baselines, the two FORGE variants, and two ablations that disentangle policy inheritance from the pretrained skill prior.

REvolve-auto baseline. Island-based reward evolution with SAC [Haarnoja et al., 2018] policies trained from scratch for 5M steps per individual. This is the prior state of the art and serves as our primary baseline.

FORGE (island). Our main method. Each child inherits the skill-conditioned actor and successor-feature critic ψ^π of its island's fittest parent, and fine-tunes both for 500K steps under the projected z^* on top of the pretrained skill prior $(\phi, \pi_0, \mathcal{B}_{\text{off}})$.

FORGE (elite). A single-population variant in which every child inherits from the single globally fittest checkpoint. It mirrors Eureka's design and tests whether global elite selection can match island-local inheritance.

Ablation: checkpoint inheritance only. Same as FORGE (island) but without the pretrained skill prior: children fine-tune from the fittest parent's SAC checkpoint for 500K steps. This isolates the contribution of cross-generation policy continuity from that of unsupervised pretraining.

Ablation: skill prior only (no inheritance). Each individual starts from π_0 every generation; skill inference provides z^* and fine-tuning still uses the pretrained buffer, but the resulting policy is discarded after evaluation. This isolates the pretrained skill prior from cross-generation policy inheritance.

F Cost Analysis

We compare compute and human input costs across methods. All configurations use 7 generations with 16 individuals per generation (112 individuals total).

Table 4: Compute and human input comparison across methods.

Method	Steps/individual	Total steps	Pretraining	Human input
REvolve-auto	5M	560M	None	None
REvolve	5M	560M	None	$O(G)$ per-gen feedback
FORGE (island)	500K	57M	1M (one-time)	None
FORGE + demos	500K	57M	1M (one-time)	$O(1)$ demos at pretrain

Compute. FORGE uses an order of magnitude fewer environment steps per individual than REvolve: 500K versus 5M. Including the one-time 1M-step pretraining, FORGE's total budget is $7 \times 16 \times 500\text{K} + 1\text{M} = 57\text{M}$ steps against REvolve's 560M. The pretraining cost amortizes across downstream tasks — the same checkpoint serves Humanoid Locomotion and Standup, and a Hand-specific checkpoint covers Adroit Hand — so per-task overhead decreases as more tasks reuse a given pretrained skill prior.

Human input. REvolve requires qualitative feedback every generation, scaling as $O(G)$. FORGE removes this requirement for tasks where unsupervised pretraining provides sufficient state coverage (Humanoid Locomotion and Standup). For sparse tasks (Adroit Hand) FORGE injects expert demonstrations once during pretraining at $O(1)$ cost, strictly cheaper than REvolve's ongoing per-generation involvement.

Knowledge persistence. FORGE leaves a reusable artifact behind, while REvolve does not. REvolve produces a reward function and a policy; the human feedback that produced them is gone after the run. FORGE additionally produces a feature space ϕ and pretrained buffer $\mathcal{B}_{\text{off}}$ that transfer to new tasks on the same embodiment without retraining — in our experiments the same humanoid checkpoint serves both Locomotion and Standup.

Wall-clock time. The $10\times$ reduction in environment steps per individual translates directly into a roughly $10\times$ reduction in wall-clock time per generation under matched hardware. Skill inference via least-squares projection is effectively instantaneous next to gradient-based fine-tuning and contributes negligible overhead.

G No-Fitness Control

This ablation isolates the contribution of fitness-guided selection. We run FORGE (island) with mutation and crossover unchanged but withhold fitness scores from the LLM, so the reward designer mutates parents without any signal of which mutations succeeded; all other components (island structure, checkpoint inheritance, pretrained skill prior) match FORGE (island). Without fitness feedback the trajectory drifts around the starting level on all three tasks (Figure 6), confirming that fitness feedback is what drives the curriculum effect.

H Reward Replay

This experiment asks whether the order in which rewards arrive along the winning ancestry carries information beyond the rewards themselves. We extract the sequence of reward functions along the best individual's lineage and replay it as a single sequential training chain with no population, fine-tuning from the previous checkpoint at each step:

$$\pi_0 \xrightarrow{r_{\sigma(0)}} \pi_1 \xrightarrow{r_{\sigma(1)}} \pi_2 \xrightarrow{r_{\sigma(2)}} \cdots \xrightarrow{r_{\sigma(G)}} \pi_{G+1},$$

where the permutation σ defines the replay mode. We compare three conditions: *forward* ($\sigma = \text{id}$, original evolutionary order), *shuffle* (uniformly random permutation), and *skip* (three sparse rewards anchored at the converged-generation reward; per-task indices in Fig. 7). Forward replays the original sequence; shuffle tests whether order matters; skip tests whether intermediate rewards are necessary or whether coarse-grained jumps suffice.

To remove compute as a confound, we hold the total fine-tuning budget fixed at 3.5M environment interactions across all conditions (7×500K, matching FORGE's per-individual budget). Forward

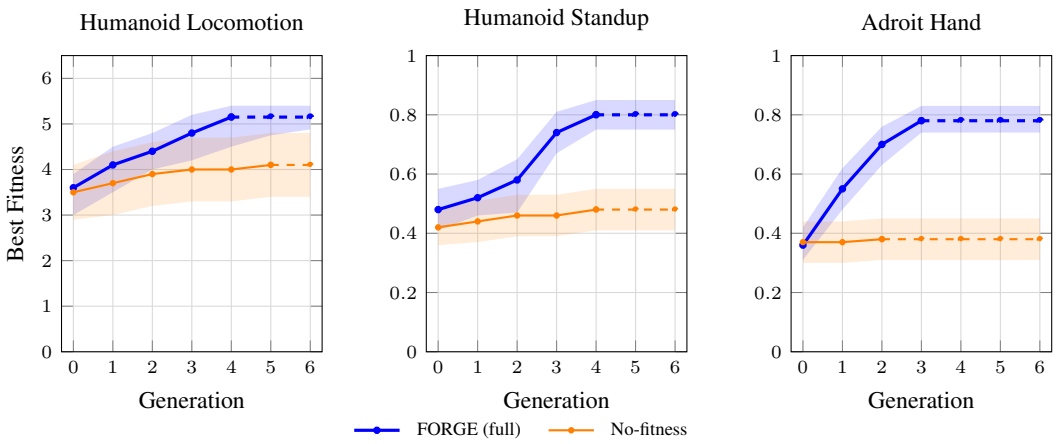

Figure 6: No-fitness ablation. Without fitness-guided selection (LLM mutates parent rewards but receives no scores), the trajectory drifts around the starting level. Shaded: std across 3 seeds.

and shuffle therefore use 500K steps per replay step (7 steps); skip uses ∼1.17M per step (3 steps). Each condition is run for 3 seeds (fresh SAC initialization, exploration noise, and an independently sampled permutation for *shuffle*); we report the per-step mean ± 1 std in Figure 7.

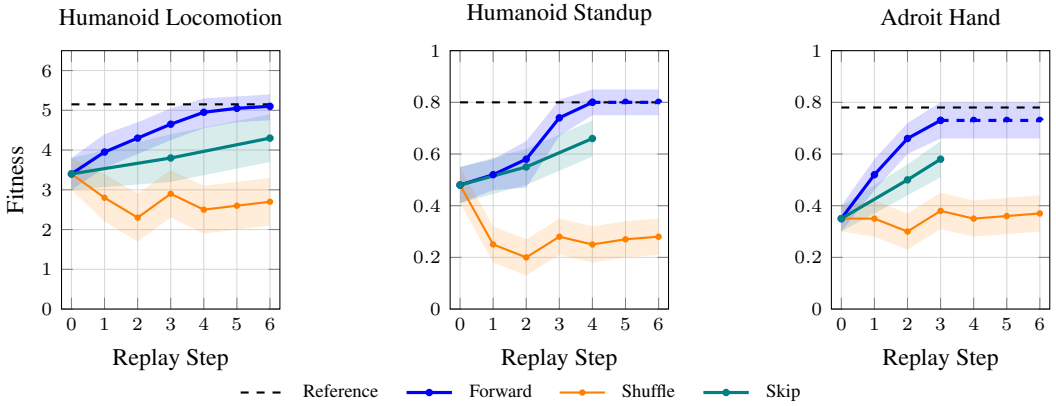

Figure 7: Reward sequence ablation. Forward replay reproduces the co-evolved reference (dashed); shuffle (random permutation) and skip both underperform — both order and intermediate steps matter. Skip uses three rewards anchored at each task's converged-generation reward: $r_0 \to r_3 \to r_6$ on Locomotion, $r_0 \to r_2 \to r_4$ on Standup, and $r_0 \to r_2 \to r_3$ on Adroit Hand. Shaded: std across 3 seeds.

I Post-Peak Recovery

This experiment looks at what happens past the peak. We extend FORGE (island) runs from the standard 7-generation horizon to 16 generations and report per-generation max fitness rather than the cumulative best used in the main results, so the dynamics past the peak are visible (Figure 8). Because the extended budget is more than 2× the standard run, we report mean ± std over 2 seeds rather than the 3 seeds used elsewhere in the appendix.

All three environments exhibit a U-shaped curve: fitness rises to a peak at the converged generation (gen 3 for Adroit Hand, gen 4–5 for Humanoid Locomotion/Standup), declines as the search explores increasingly speculative reward functions, and partially recovers as later generations re-discover productive reward–checkpoint combinations. Trough depth and recovery height vary by task.

The recovery does not return to the original peak. This is consistent with cumulative representation damage from extended co-evolution (cf. primacy bias, Section 6) and motivates early stopping or generation-level rollback as future mitigations.

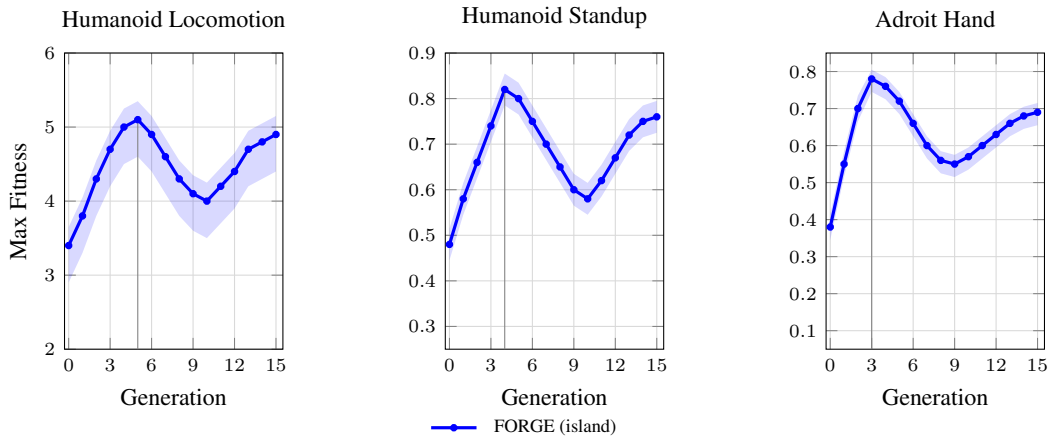

Figure 8: Per-generation max fitness under 15-generation FORGE (island). All three tasks show a U-shape: rise to a mid-run peak, decline through speculative rewards that damage inherited representations, partial recovery below the peak — evidence of cumulative plasticity loss. Vertical lines mark peak generations. Shaded: std across 2 seeds.

J Env Inputs and LLM Prompts

For each task, the reward designer LLM receives three concatenated components: a *system prompt* fixing the role and output format, an *env input* that describes the simulator's observation and action spaces and names the variables available in `compute_reward`, and a per-operation *mutation* or *crossover* prompt specifying the evolutionary operation. We reproduce all four artifacts verbatim for each of the three tasks below. The auto- and VLM-feedback variants used in ablations differ only in a feedback block appended to the mutation/crossover prompts and are omitted here.

J.1 Humanoid Locomotion

System prompt (`prompts/system_prompt`)

```
You are a reward engineer trying to write a reward function for the environment
    ↪ that will help the agent learn the task described in text.
Task description:
The task is to train a humanoid agent to run. The robot becomes unhealthy if its
    ↪ torso's z-position falls outside the 1.0 to 2.0 range. The episode ends
    ↪ if the robot becomes unhealthy or after 1000 timesteps, with success
    ↪ measured by speed. Agent locomotion should be more human like.
Your reward function should use useful variables from the environment as inputs.
    ↪ As an example the reward function signature can be:
def compute_reward(object_pos: type goal_pos: type) -> Tuple[type, Dict[str,
    ↪ type]]:
...
return reward, {}
The output of the reward function should consist of two items:
1: the total reward,
2: a dictionary of each individual reward component.
The code output should be formatted as a python code string: "'''python ... '''".
1: You may find it helpful to normalize the reward to a fixed reward range like
    ↪ -1,1 or 0,1 by applying transformations like np.exp() or numpy.exp() (use
    ↪ only numpy ) to the overall reward or its components.
```

```
2: If you choose to transform a reward component, then you must introduce a
    ↪ temperature parameter inside the transformation function. This parameter
    ↪ must be a named variable in the reward function and it must not be an
    ↪ input variable. Each transformed reward component should have its own
    ↪ temperature variable and you must carefully set it's value based on its
    ↪ effect in the overall reward.
3: Make sure the type of each input variable is correctly specified.
4: Most importantly, the reward's code input variables must contain only
    ↪ attributes of the environment. The only input variables for the reward
    ↪ function that you can and must use are from the function def
    ↪ _get_obs(self) from the observation variable (the concatenated output so
    ↪ one single input in the compute reward).  Under no circumstance can you
    ↪ introduce a new input variable. or assume anything.
5. Make sure you dont give contradictory reward components.
```

Env input (`prompts/env_input`)

```
class HumanoidEnv(MujocoEnv, utils.EzPickle):
    """
    ## Description

    ## Action Space
    The action space is a `Box(-1, 1, (17,), float32)`. An action represents the
        ↪ torques applied at the hinge joints.

    ## Observation Space
    Observations consist of positional values of different body parts of the
        ↪ Humanoid,
    followed by the velocities of those individual parts (their derivatives)
        ↪ with all the
    positions ordered before all the velocities.

    By default, observations do not include the x- and y-coordinates of the
        ↪ torso. These may
    be included by passing `exclude_current_positions_from_observation=False`
        ↪ during construction.
    In that case, the observation space will be a `Box(-Inf, Inf, (378,),
        ↪ float64)` where the first two observations
    represent the x- and y-coordinates of the torso.
    Regardless of whether `exclude_current_positions_from_observation` was set
        ↪ to true or false, the x- and y-coordinates
    will be returned in `info` with keys `"x_position"` and `"y_position"`,
        ↪ respectively.

    However, by default, the observation is a `Box(-Inf, Inf, (376,), float64)`.
        ↪ The elements correspond to the following:

    | Num | Observation
        ↪ | Min  | Max | Name (in corresponding XML file) | Joint | Unit
        ↪ |
    | --- | ------------------------------------------------------------------┘
        ↪ ----------------------------------------- | ---- | --- |
        ↪ -------------------------------- | ----- | --------------------------
        ↪ |
    | 0   | z-coordinate of the torso (centre)
        ↪ | -Inf | Inf | root                              | free  | position
        ↪ (m)              |
    | 1   | x-orientation of the torso (centre)
        ↪ | -Inf | Inf | root                              | free  | angle
        ↪ (rad)               |
    | 2   | y-orientation of the torso (centre)
        ↪ | -Inf | Inf | root                              | free  | angle
        ↪ (rad)               |
```

```
| 3   | z-orientation of the torso (centre)
    ↪ | -Inf | Inf | root                              | free  | angle
    ↪ (rad)               |
| 4   | w-orientation of the torso (centre)
    ↪ | -Inf | Inf | root                              | free  | angle
    ↪ (rad)               |
| 5   | z-angle of the abdomen (in lower_waist)
    ↪ | -Inf | Inf | abdomen_z                         | hinge | angle
    ↪ (rad)               |
| 6   | y-angle of the abdomen (in lower_waist)
    ↪ | -Inf | Inf | abdomen_y                         | hinge | angle
    ↪ (rad)               |
| 7   | x-angle of the abdomen (in pelvis)
    ↪ | -Inf | Inf | abdomen_x                         | hinge | angle
    ↪ (rad)               |
| 8   | x-coordinate of angle between pelvis and right hip (in right_thigh)
    ↪ | -Inf | Inf | right_hip_x                       | hinge | angle
    ↪ (rad)               |
| 9   | z-coordinate of angle between pelvis and right hip (in right_thigh)
    ↪ | -Inf | Inf | right_hip_z                       | hinge | angle
    ↪ (rad)               |
| 10  | y-coordinate of angle between pelvis and right hip (in right_thigh)
    ↪ | -Inf | Inf | right_hip_y                       | hinge | angle
    ↪ (rad)               |
| 11  | angle between right hip and the right shin (in right_knee)
    ↪ | -Inf | Inf | right_knee                        | hinge | angle
    ↪ (rad)               |
| 12  | x-coordinate of angle between pelvis and left hip (in left_thigh)
    ↪ | -Inf | Inf | left_hip_x                        | hinge | angle
    ↪ (rad)               |
| 13  | z-coordinate of angle between pelvis and left hip (in left_thigh)
    ↪ | -Inf | Inf | left_hip_z                        | hinge | angle
    ↪ (rad)               |
| 14  | y-coordinate of angle between pelvis and left hip (in left_thigh)
    ↪ | -Inf | Inf | left_hip_y                        | hinge | angle
    ↪ (rad)               |
| 15  | angle between left hip and the left shin (in left_knee)
    ↪ | -Inf | Inf | left_knee                         | hinge | angle
    ↪ (rad)               |
| 16  | coordinate-1 (multi-axis) angle between torso and right arm (in
    ↪ right_upper_arm)                                | -Inf | Inf |
    ↪ right_shoulder1                 | hinge | angle (rad)
    ↪ |
| 17  | coordinate-2 (multi-axis) angle between torso and right arm (in
    ↪ right_upper_arm)                                | -Inf | Inf |
    ↪ right_shoulder2                 | hinge | angle (rad)
    ↪ |
| 18  | angle between right upper arm and right_lower_arm
    ↪ | -Inf | Inf | right_elbow                       | hinge | angle
    ↪ (rad)               |
| 19  | coordinate-1 (multi-axis) angle between torso and left arm (in
    ↪ left_upper_arm)                                  | -Inf | Inf |
    ↪ left_shoulder1                  | hinge | angle (rad)
    ↪ |
| 20  | coordinate-2 (multi-axis) angle between torso and left arm (in
    ↪ left_upper_arm)                                  | -Inf | Inf |
    ↪ left_shoulder2                  | hinge | angle (rad)
    ↪ |
| 21  | angle between left upper arm and left_lower_arm
    ↪ | -Inf | Inf | left_elbow                        | hinge | angle
    ↪ (rad)               |
| 22  | x-coordinate velocity of the torso (centre)
    ↪ | -Inf | Inf | root                              | free  | velocity
    ↪ (m/s)            |
```

```
| 23  | y-coordinate velocity of the torso (centre)
    ↪ | -Inf | Inf | root                             | free  | velocity
    ↪ (m/s)            |
| 24  | z-coordinate velocity of the torso (centre)
    ↪ | -Inf | Inf | root                             | free  | velocity
    ↪ (m/s)            |
| 25  | x-coordinate angular velocity of the torso (centre)
    ↪ | -Inf | Inf | root                             | free  | anglular
    ↪ velocity (rad/s)  |
| 26  | y-coordinate angular velocity of the torso (centre)
    ↪ | -Inf | Inf | root                             | free  | anglular
    ↪ velocity (rad/s)  |
| 27  | z-coordinate angular velocity of the torso (centre)
    ↪ | -Inf | Inf | root                             | free  | anglular
    ↪ velocity (rad/s)  |
| 28  | z-coordinate of angular velocity of the abdomen (in lower_waist)
    ↪ | -Inf | Inf | abdomen_z                        | hinge | anglular
    ↪ velocity (rad/s)  |
| 29  | y-coordinate of angular velocity of the abdomen (in lower_waist)
    ↪ | -Inf | Inf | abdomen_y                        | hinge | anglular
    ↪ velocity (rad/s)  |
| 30  | x-coordinate of angular velocity of the abdomen (in pelvis)
    ↪ | -Inf | Inf | abdomen_x                        | hinge | aanglular
    ↪ velocity (rad/s) |
| 31  | x-coordinate of the angular velocity of the angle between pelvis and
    ↪ right hip (in right_thigh)                | -Inf | Inf |
    ↪ right_hip_x                     | hinge | anglular velocity (rad/s)
    ↪ |
| 32  | z-coordinate of the angular velocity of the angle between pelvis and
    ↪ right hip (in right_thigh)                | -Inf | Inf |
    ↪ right_hip_z                     | hinge | anglular velocity (rad/s)
    ↪ |
| 33  | y-coordinate of the angular velocity of the angle between pelvis and
    ↪ right hip (in right_thigh)                | -Inf | Inf |
    ↪ right_hip_y                     | hinge | anglular velocity (rad/s)
    ↪ |
| 34  | angular velocity of the angle between right hip and the right shin
    ↪ (in right_knee)                             | -Inf | Inf |
    ↪ right_knee                      | hinge | anglular velocity (rad/s)
    ↪ |
| 35  | x-coordinate of the angular velocity of the angle between pelvis and
    ↪ left hip (in left_thigh)                  | -Inf | Inf | left_hip_x
    ↪ | hinge | anglular velocity (rad/s)  |
| 36  | z-coordinate of the angular velocity of the angle between pelvis and
    ↪ left hip (in left_thigh)                  | -Inf | Inf | left_hip_z
    ↪ | hinge | anglular velocity (rad/s)  |
| 37  | y-coordinate of the angular velocity of the angle between pelvis and
    ↪ left hip (in left_thigh)                  | -Inf | Inf | left_hip_y
    ↪ | hinge | anglular velocity (rad/s)  |
| 38  | angular velocity of the angle between left hip and the left shin (in
    ↪ left_knee)                                | -Inf | Inf | left_knee
    ↪ | hinge | anglular velocity (rad/s)  |
| 39  | coordinate-1 (multi-axis) of the angular velocity of the angle
    ↪ between torso and right arm (in right_upper_arm) | -Inf | Inf |
    ↪ right_shoulder1                 | hinge | anglular velocity (rad/s)
    ↪ |
| 40  | coordinate-2 (multi-axis) of the angular velocity of the angle
    ↪ between torso and right arm (in right_upper_arm) | -Inf | Inf |
    ↪ right_shoulder2                 | hinge | anglular velocity (rad/s)
    ↪ |
| 41  | angular velocity of the angle between right upper arm and
    ↪ right_lower_arm                                       | -Inf | Inf |
    ↪ right_elbow                     | hinge | anglular velocity (rad/s)
    ↪ |
```

```
| 42  | coordinate-1 (multi-axis) of the angular velocity of the angle
    ↪ between torso and left arm (in left_upper_arm)   | -Inf | Inf |
    ↪ left_shoulder1                    | hinge | anglular velocity (rad/s)
    ↪ |
| 43  | coordinate-2 (multi-axis) of the angular velocity of the angle
    ↪ between torso and left arm (in left_upper_arm)   | -Inf | Inf |
    ↪ left_shoulder2                    | hinge | anglular velocity (rad/s)
    ↪ |
| 44  | angular velocity of the angle between left upper arm and
    ↪ left_lower_arm                                          | -Inf | Inf
    ↪ | left_elbow                        | hinge | anglular velocity
    ↪ (rad/s)  |
| excluded | x-coordinate of the torso (centre)
    ↪ | -Inf | Inf | root                              | free  | position
    ↪ (m)               |
| excluded | y-coordinate of the torso (centre)
    ↪ | -Inf | Inf | root                              | free  | position
    ↪ (m)               |

Additionally, after all the positional and velocity based values in the
    ↪ table,
the observation contains (in order):
- *cinert:* Mass and inertia of a single rigid body relative to the center
    ↪ of mass
(this is an intermediate result of transition). It has shape 14*10 (*nbody *
    ↪ 10*)
and hence adds to another 140 elements in the state space.
- *cvel:* Center of mass based velocity. It has shape 14 * 6 (*nbody * 6*)
    ↪ and hence
adds another 84 elements in the state space
- *qfrc_actuator:* Constraint force generated as the actuator force. This
    ↪ has shape
`(23,)`  *(nv * 1)* and hence adds another 23 elements to the state space.
- *cfrc_ext:* This is the center of mass based external force on the body.
    ↪ It has shape
14 * 6 (*nbody * 6*) and hence adds to another 84 elements in the state
    ↪ space.
where *nbody* stands for the number of bodies in the robot and *nv* stands
    ↪ for the
number of degrees of freedom (*= dim(qvel)*)

The body parts are:

| id (for `v2`,`v3`,`v4`) | body part |
| --- |  ------------  |
| 0   | worldBody (note: all values are constant 0) |
| 1   | torso |
| 2   | lwaist |
| 3   | pelvis |
| 4   | right_thigh |
| 5   | right_sin |
| 6   | right_foot |
| 7   | left_thigh |
| 8   | left_sin |
| 9   | left_foot |
| 10  | right_upper_arm |
| 11  | right_lower_arm |
| 12  | left_upper_arm |
| 13  | left_lower_arm |

The joints are:

| id (for `v2`,`v3`,`v4`) | joint |
| --- |  ------------  |
```

```
| 0   | root |
| 1   | root |
| 2   | root |
| 3   | root |
| 4   | root |
| 5   | root |
| 6   | abdomen_z |
| 7   | abdomen_y |
| 8   | abdomen_x |
| 9   | right_hip_x |
| 10  | right_hip_z |
| 11  | right_hip_y |
| 12  | right_knee |
| 13  | left_hip_x |
| 14  | left_hiz_z |
| 15  | left_hip_y |
| 16  | left_knee |
| 17  | right_shoulder1 |
| 18  | right_shoulder2 |
| 19  | right_elbow|
| 20  | left_shoulder1 |
| 21  | left_shoulder2 |
| 22  | left_elfbow |



## Episode End
The humanoid is said to be unhealthy if the z-position of the torso is no
    ↪ longer contained in the
closed interval specified by the argument `healthy_z_range`.

If `terminate_when_unhealthy=True` is passed during construction (which is
    ↪ the default),
the episode ends when any of the following happens:

1. Truncation: The episode duration reaches a 1000 timesteps
3. Termination: The humanoid is unhealthy

If `terminate_when_unhealthy=False` is passed, the episode is ended only
    ↪ when 1000 timesteps are exceeded.

    *position: indices from 0 to 21 -> obs[0:22]
    *velocity: indices from 22 to 44 -> obs[22:45]
    *com_inertia: indices from 45 to 184 -> obs[45:185]
    *com_velocity: indices from 185 to 268 -> obs[185:269]
    *actuator_forces: indices from 269 to 291 -> obs[269:292]
    *external_contact_forces: indices from 292 to 375 -> obs[292:376]


@property
def is_healthy(self):
    min_z, max_z = self._healthy_z_range
    is_healthy = min_z < self.data.qpos[2] < max_z

    return is_healthy


def _get_obs(self):
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

    position = self.data.qpos.flat.copy()
    velocity = self.data.qvel.flat.copy()

    com_inertia = self.data.cinert.flat.copy()
    com_velocity = self.data.cvel.flat.copy()
```

```
        actuator_forces = self.data.qfrc_actuator.flat.copy()
        external_contact_forces = self.data.cfrc_ext.flat.copy()


        observation = np.concatenate(
        (position, velocity, com_inertia, com_velocity, actuator_forces,
            ↪  external_contact_forces,)
    )

        return observation
```

Mutation prompt (`prompts/mutation`)

```
You are given a reward function used to train a humanoid robot to run. The
    ↪ reward functions has a fitness scores that reflects the agent's
    ↪ performance -- higher scores indicate superior running behavior.
    ↪ Additionally, we tracked the values of the individual components in the
    ↪ reward function after every <EPISODES> episodes.

<EXAMPLES>

Your task is to iterate on the reward function by mutating a single component to
    ↪ enhance the agent's performance. Some helpful tips for analyzing the
    ↪ reward components:
If the values for a certain reward component are near identical throughout, then
    ↪ this means the training is not able to optimize this component as it is
    ↪ written. You may consider
(a) Rescaling it to a proper range or the value of its temperature parameter
(b) Re-writing the reward component
(c) Discarding the reward component

The mutation process involves the following steps:
First, select a reward component based on the given guidelines, human feedback,
    ↪ and the values of individual reward components.
Next, clearly explain how you intend to mutate the selected component and your
    ↪ rationale behind improving the performance. This could involve adjusting
    ↪ its scale, rewriting its formula, or other modifications.
Finally, write the mutated reward function code.

The output of the reward function should consist of two items:
(1) the total reward,
(2) a dictionary of each individual reward component.

The code output should be formatted as a python code string: "```python ... ```".

Some helpful tips for writing the reward function code:
1: You may find it helpful to normalize the reward to a fixed reward range like
    ↪ -1,1 or 0,1 by applying transformations like np.exp() to the overall
    ↪ reward or its components.
2: If you choose to transform a reward component, then you must introduce a
    ↪ temperature parameter inside the transformation function. This parameter
    ↪ must be a named variable in the reward function and it must not be an
    ↪ input variable. Each transformed reward component should have its own
    ↪ temperature variable and you must carefully set it's value based on its
    ↪ effect in the overall reward.
3: Make sure the type of each input variable is correctly specified.
4: Most importantly, the reward's code input variables must contain only
    ↪ attributes of the environment. Under no circumstance can you introduce a
    ↪ new input variable.
```

Crossover prompt (`prompts/crossover`)

```
You are given a set of reward functions used to train humanoid robot to run. The
    ↪ reward functions have fitness scores that reflect the agent's
    ↪ performance higher scores indicate superior running behavior. For each
    ↪ reward function, we collected human feedback on what aspects of the
    ↪ agent are satisfactory and what aspects need improvement.  Additionally,
    ↪ we tracked the values of the individual components in the reward
    ↪ function after every <EPISODES> episodes.

<EXAMPLES>

Your task is to combine high-performing reward components from different reward
    ↪ functions and combine them to to enhance the agent's performance. Some
    ↪ helpful tips for analyzing the reward components:
If the values for a certain reward component are near identical throughout, then
    ↪ this means the training is not able to optimize this component as it is
    ↪ written.

The crossover process involves the following steps:
First, select high-performing reward components based on the human feedback and
    ↪ the values of individual reward components.
Next, clearly explain how you intend to combine them and your rationale behind
    ↪ improving the performance.
Finally, write the combined reward function code.

The output of the reward function should consist of two items:
1: the total reward,
2: a dictionary of each individual reward component.
The code output should be formatted as a python code string: "'''python ... '''".
1: You may find it helpful to normalize the reward to a fixed reward range like
    ↪ -1,1 or 0,1 by applying transformations like np.exp()to the overall
    ↪ reward or its components.
2: If you choose to transform a reward component, then you must introduce a
    ↪ temperature parameter inside the transformation function. This parameter
    ↪ must be a named variable in the reward function and it must not be an
    ↪ input variable. Each transformed reward component should have its own
    ↪ temperature variable and you must carefully set it's value based on its
    ↪ effect in the overall reward.
3: Make sure the type of each input variable is correctly specified.
4: Most importantly, the reward's code input variables must contain only
    ↪ attributes of the environment. The only input variables for the reward
    ↪ function that you can and must use are from the function def
    ↪ _get_obs(self) from the observation variable (the concatenated output so
    ↪ one single input in the compute reward).  Under no circumstance can you
    ↪ introduce a new input variable. or assume anything.
5. Make sure you dont give contradictory reward components.
```

J.2 Humanoid Standup

System prompt (`prompts/system_prompt_standup`)

```
You are a reward engineer trying to write a reward function for the environment
    ↪ that will help the agent learn the task described in text.

Task description:
The task is to train a humanoid agent to stand up from a lying-down position on
    ↪ the ground. The humanoid starts on its back and must learn to get up and
    ↪ achieve a stable upright standing posture. A successful standup should:
1. Transition from lying on the ground to an upright standing position.
2. Achieve a torso height close to the normal standing height (~1.4m).
3. Maintain an upright orientation (torso quaternion close to [1,0,0,0]).
4. Remain balanced and stable after standing up.
```

```
This is NOT a locomotion task. Do not optimize for running speed or forward
    ↪ movement. The reward should favor achieving and maintaining a stable
    ↪ standing posture rather than moving forward, crawling, or rolling.

The environment uses conditional termination: the agent starts lying down and
    ↪ will NOT be terminated initially. Once the agent has made meaningful
    ↪ upward progress, termination is armed -- if the agent then falls back
    ↪ (drops more than 0.2m from its peak height), the episode ends
    ↪ immediately. This means: every bit of upward progress matters, but
    ↪ collapsing back is punished by losing all remaining episode reward.
    ↪ Design your reward so that the agent is incentivized to rise up AND
    ↪ maintain its height.

Your reward function should use useful variables from the environment as inputs.
    ↪ A valid signature can use `observation` and/or other provided humanoid
    ↪ state attributes listed in the environment description, for example:
def compute_reward(observation: np.ndarray) -> Tuple[float, Dict[str, float]]:
...
return reward, {}

The output of the reward function should consist of two items:
1: the total reward,
2: a dictionary of each individual reward component.

The code output should be formatted as a python code string: "```python ... ```".

Important reward-design guidance:
1: Key observation indices for standup: obs[0] is torso z-height (starts ~0.2
    ↪ lying down, target ~1.4 standing), obs[1:5] is torso quaternion in
    ↪ [w,x,y,z] order (upright=[1,0,0,0], lying on back=[0.707,0.707,0,0]),
    ↪ obs[24] is vertical velocity, obs[25:28] is angular velocity.
2: Include an alive/height bonus that provides a continuous gradient: reward the
    ↪ agent proportionally to its current torso height (obs[0]). This ensures
    ↪ the agent always receives signal for making upward progress, similar to
    ↪ how locomotion tasks use an alive bonus to keep the agent standing.
3: Design phase-aware rewards. When torso height is low (< 0.5m), prioritize
    ↪ rewarding upward velocity (obs[24] > 0) over upright orientation. When
    ↪ height is higher (> 0.8m), shift weight heavily toward uprightness
    ↪ (obs[1] close to 1.0) and stability (low angular velocity obs[25:28]).
    ↪ This prevents contradictory signals at different stages.
4: Once the agent is standing (obs[0] > 1.0m), stability is the most important
    ↪ objective -- more important than standing up quickly. An agent that
    ↪ stands up slowly but remains perfectly still is far better than one that
    ↪ stands up fast but wobbles.
5: Early in training the agent is on the ground -- reward any upward progress,
    ↪ not just the final standing state. Do not over-penalize non-upright
    ↪ orientations during the transition phase. The agent must go through
    ↪ intermediate postures (rolling, crouching) to stand up.
6: You may find it helpful to normalize the reward to a fixed reward range like
    ↪ -1,1 or 0,1 by applying transformations like np.exp() or numpy.exp() (use
    ↪ only numpy) to the overall reward or its components.
7: If you choose to transform a reward component, then you must introduce a
    ↪ temperature parameter inside the transformation function. This parameter
    ↪ must be a named variable in the reward function and it must not be an
    ↪ input variable. Each transformed reward component should have its own
    ↪ temperature variable and you must carefully set its value based on its
    ↪ effect in the overall reward.
8: Make sure the type of each input variable is correctly specified.
9: Most importantly, the reward code may only use attributes that are actually
    ↪ provided by the environment. Under no circumstance can you introduce a
    ↪ new input variable or assume unavailable state.
10: Make sure you do not create contradictory reward components, for example
    ↪ strongly rewarding upright posture at all times while also needing the
    ↪ agent to push off the ground through non-upright positions.
```

Env input (`prompts/env_input_standup`)

```
class HumanoidStandupEnv(HumanoidEnv):
    """
    ## Description
    The humanoid starts lying on its back on the ground and must learn to stand
        ↪ up.
    Uses `terminate_when_unhealthy=False` so the agent is never terminated for
        ↪ low torso height.
    Episodes always run for the full 1000 timesteps.

    ## Action Space
    The action space is a `Box(-1, 1, (17,), float32)`. An action represents the
        ↪ torques applied at the hinge joints.

    ## Observation Space
    Observations consist of positional values of different body parts of the
        ↪ Humanoid,
    followed by the velocities of those individual parts (their derivatives)
        ↪ with all the
    positions ordered before all the velocities.

    By default, observations do not include the x- and y-coordinates of the
        ↪ torso. These may
    be included by passing `exclude_current_positions_from_observation=False`
        ↪ during construction.
    In that case, the observation space will be a `Box(-Inf, Inf, (378,),
        ↪ float64)` where the first two observations
    represent the x- and y-coordinates of the torso.
    Regardless of whether `exclude_current_positions_from_observation` was set
        ↪ to true or false, the x- and y-coordinates
    will be returned in `info` with keys `"x_position"` and `"y_position"`,
        ↪ respectively.

    However, by default, the observation is a `Box(-Inf, Inf, (376,), float64)`.
        ↪ The elements correspond to the following:

    | Num | Observation
        ↪ | Min  | Max | Name (in corresponding XML file) | Joint | Unit
        ↪ |
    | --- | ------------------------------------------------------------------┘
        ↪ ----------------------------------------- | ---- | --- |
        ↪ -------------------------------- | ----- | -------------------------
        ↪ |
    | 0   | z-coordinate of the torso (centre)
        ↪ | -Inf | Inf | root                             | free  | position
        ↪ (m)              |
    | 1   | w-orientation of the torso quaternion (centre)
        ↪ | -Inf | Inf | root                             | free  | quaternion
        ↪ |
    | 2   | x-orientation of the torso quaternion (centre)
        ↪ | -Inf | Inf | root                             | free  | quaternion
        ↪ |
    | 3   | y-orientation of the torso quaternion (centre)
        ↪ | -Inf | Inf | root                             | free  | quaternion
        ↪ |
    | 4   | z-orientation of the torso quaternion (centre)
        ↪ | -Inf | Inf | root                             | free  | quaternion
        ↪ |
    | 5   | z-angle of the abdomen (in lower_waist)
        ↪ | -Inf | Inf | abdomen_z                        | hinge | angle
        ↪ (rad)               |
```

```
| 6   | y-angle of the abdomen (in lower_waist)
    ↪ | -Inf | Inf | abdomen_y                          | hinge | angle
    ↪ (rad)              |
| 7   | x-angle of the abdomen (in pelvis)
    ↪ | -Inf | Inf | abdomen_x                          | hinge | angle
    ↪ (rad)              |
| 8   | x-coordinate of angle between pelvis and right hip (in right_thigh)
    ↪ | -Inf | Inf | right_hip_x                        | hinge | angle
    ↪ (rad)              |
| 9   | z-coordinate of angle between pelvis and right hip (in right_thigh)
    ↪ | -Inf | Inf | right_hip_z                        | hinge | angle
    ↪ (rad)              |
| 10  | y-coordinate of angle between pelvis and right hip (in right_thigh)
    ↪ | -Inf | Inf | right_hip_y                        | hinge | angle
    ↪ (rad)              |
| 11  | angle between right hip and the right shin (in right_knee)
    ↪ | -Inf | Inf | right_knee                         | hinge | angle
    ↪ (rad)              |
| 12  | x-coordinate of angle between pelvis and left hip (in left_thigh)
    ↪ | -Inf | Inf | left_hip_x                         | hinge | angle
    ↪ (rad)              |
| 13  | z-coordinate of angle between pelvis and left hip (in left_thigh)
    ↪ | -Inf | Inf | left_hip_z                         | hinge | angle
    ↪ (rad)              |
| 14  | y-coordinate of angle between pelvis and left hip (in left_thigh)
    ↪ | -Inf | Inf | left_hip_y                         | hinge | angle
    ↪ (rad)              |
| 15  | angle between left hip and the left shin (in left_knee)
    ↪ | -Inf | Inf | left_knee                          | hinge | angle
    ↪ (rad)              |
| 16  | coordinate-1 (multi-axis) angle between torso and right arm (in
    ↪ right_upper_arm)                                  | -Inf | Inf |
    ↪ right_shoulder1                | hinge | angle (rad)
    ↪ |
| 17  | coordinate-2 (multi-axis) angle between torso and right arm (in
    ↪ right_upper_arm)                                  | -Inf | Inf |
    ↪ right_shoulder2                | hinge | angle (rad)
    ↪ |
| 18  | angle between right upper arm and right_lower_arm
    ↪ | -Inf | Inf | right_elbow                        | hinge | angle
    ↪ (rad)              |
| 19  | coordinate-1 (multi-axis) angle between torso and left arm (in
    ↪ left_upper_arm)                                    | -Inf | Inf |
    ↪ left_shoulder1                 | hinge | angle (rad)
    ↪ |
| 20  | coordinate-2 (multi-axis) angle between torso and left arm (in
    ↪ left_upper_arm)                                    | -Inf | Inf |
    ↪ left_shoulder2                 | hinge | angle (rad)
    ↪ |
| 21  | angle between left upper arm and left_lower_arm
    ↪ | -Inf | Inf | left_elbow                         | hinge | angle
    ↪ (rad)              |
| 22  | x-coordinate velocity of the torso (centre)
    ↪ | -Inf | Inf | root                               | free  | velocity
    ↪ (m/s)            |
| 23  | y-coordinate velocity of the torso (centre)
    ↪ | -Inf | Inf | root                               | free  | velocity
    ↪ (m/s)            |
| 24  | z-coordinate velocity of the torso (centre)
    ↪ | -Inf | Inf | root                               | free  | velocity
    ↪ (m/s)            |
| 25  | x-coordinate angular velocity of the torso (centre)
    ↪ | -Inf | Inf | root                               | free  | anglular
    ↪ velocity (rad/s)  |
```

```
| 26  | y-coordinate angular velocity of the torso (centre)
    ↪ | -Inf | Inf | root                              | free  | anglular
    ↪ velocity (rad/s)  |
| 27  | z-coordinate angular velocity of the torso (centre)
    ↪ | -Inf | Inf | root                              | free  | anglular
    ↪ velocity (rad/s)  |
| 28  | z-coordinate of angular velocity of the abdomen (in lower_waist)
    ↪ | -Inf | Inf | abdomen_z                         | hinge | anglular
    ↪ velocity (rad/s)  |
| 29  | y-coordinate of angular velocity of the abdomen (in lower_waist)
    ↪ | -Inf | Inf | abdomen_y                         | hinge | anglular
    ↪ velocity (rad/s)  |
| 30  | x-coordinate of angular velocity of the abdomen (in pelvis)
    ↪ | -Inf | Inf | abdomen_x                         | hinge | aanglular
    ↪ velocity (rad/s) |
| 31  | x-coordinate of the angular velocity of the angle between pelvis and
    ↪ right hip (in right_thigh)                | -Inf | Inf |
    ↪ right_hip_x                      | hinge | anglular velocity (rad/s)
    ↪ |
| 32  | z-coordinate of the angular velocity of the angle between pelvis and
    ↪ right hip (in right_thigh)                | -Inf | Inf |
    ↪ right_hip_z                      | hinge | anglular velocity (rad/s)
    ↪ |
| 33  | y-coordinate of the angular velocity of the angle between pelvis and
    ↪ right hip (in right_thigh)                | -Inf | Inf |
    ↪ right_hip_y                      | hinge | anglular velocity (rad/s)
    ↪ |
| 34  | angular velocity of the angle between right hip and the right shin
    ↪ (in right_knee)                             | -Inf | Inf |
    ↪ right_knee                       | hinge | anglular velocity (rad/s)
    ↪ |
| 35  | x-coordinate of the angular velocity of the angle between pelvis and
    ↪ left hip (in left_thigh)                  | -Inf | Inf | left_hip_x
    ↪ | hinge | anglular velocity (rad/s)  |
| 36  | z-coordinate of the angular velocity of the angle between pelvis and
    ↪ left hip (in left_thigh)                  | -Inf | Inf | left_hip_z
    ↪ | hinge | anglular velocity (rad/s)  |
| 37  | y-coordinate of the angular velocity of the angle between pelvis and
    ↪ left hip (in left_thigh)                  | -Inf | Inf | left_hip_y
    ↪ | hinge | anglular velocity (rad/s)  |
| 38  | angular velocity of the angle between left hip and the left shin (in
    ↪ left_knee)                                | -Inf | Inf | left_knee
    ↪ | hinge | anglular velocity (rad/s)  |
| 39  | coordinate-1 (multi-axis) of the angular velocity of the angle
    ↪ between torso and right arm (in right_upper_arm) | -Inf | Inf |
    ↪ right_shoulder1                  | hinge | anglular velocity (rad/s)
    ↪ |
| 40  | coordinate-2 (multi-axis) of the angular velocity of the angle
    ↪ between torso and right arm (in right_upper_arm) | -Inf | Inf |
    ↪ right_shoulder2                  | hinge | anglular velocity (rad/s)
    ↪ |
| 41  | angular velocity of the angle between right upper arm and
    ↪ right_lower_arm                                       | -Inf | Inf |
    ↪ right_elbow                      | hinge | anglular velocity (rad/s)
    ↪ |
| 42  | coordinate-1 (multi-axis) of the angular velocity of the angle
    ↪ between torso and left arm (in left_upper_arm)   | -Inf | Inf |
    ↪ left_shoulder1                   | hinge | anglular velocity (rad/s)
    ↪ |
| 43  | coordinate-2 (multi-axis) of the angular velocity of the angle
    ↪ between torso and left arm (in left_upper_arm)   | -Inf | Inf |
    ↪ left_shoulder2                   | hinge | anglular velocity (rad/s)
    ↪ |
```

```
| 44  | angular velocity of the angle between left upper arm and
    ↪ left_lower_arm                                          | -Inf | Inf
    ↪ | left_elbow                        | hinge | anglular velocity
    ↪ (rad/s)  |
| excluded | x-coordinate of the torso (centre)
    ↪ | -Inf | Inf | root                              | free  | position
    ↪ (m)               |
| excluded | y-coordinate of the torso (centre)
    ↪ | -Inf | Inf | root                              | free  | position
    ↪ (m)               |

Additionally, after all the positional and velocity based values in the
    ↪ table,
the observation contains (in order):
- *cinert:* Mass and inertia of a single rigid body relative to the center
    ↪ of mass
(this is an intermediate result of transition). It has shape 14*10 (*nbody *
    ↪ 10*)
and hence adds to another 140 elements in the state space.
- *cvel:* Center of mass based velocity. It has shape 14 * 6 (*nbody * 6*)
    ↪ and hence
adds another 84 elements in the state space
- *qfrc_actuator:* Constraint force generated as the actuator force. This
    ↪ has shape
`(23,)`  *(nv * 1)* and hence adds another 23 elements to the state space.
- *cfrc_ext:* This is the center of mass based external force on the body.
    ↪ It has shape
14 * 6 (*nbody * 6*) and hence adds to another 84 elements in the state
    ↪ space.
where *nbody* stands for the number of bodies in the robot and *nv* stands
    ↪ for the
number of degrees of freedom (*= dim(qvel)*)

## Standup-Specific Signal Guidance

For the standup task, the most relevant observation signals are:
- **Height/Progress**: obs[0] (torso z-height) -- starts ~0.2 (lying),
    ↪ target ~1.4 (standing)
- **Uprightness**: obs[1:5] (torso quaternion [w,x,y,z]) -- upright =
    ↪ [1,0,0,0], lying on back = [0.707,0.707,0,0]
- **Vertical velocity**: obs[24] (z-velocity) -- positive = moving upward
- **Stability**: obs[25:28] (angular velocity) -- low = stable
- **Horizontal velocity**: obs[22], obs[23] -- NOT the primary objective

## Episode End
The environment uses conditional termination. The agent starts lying on its
    ↪ back (torso z ~ 0.2)
and is NOT terminated initially. Once the agent has risen meaningfully above
    ↪ the ground,
termination is armed: if the agent falls back (drops more than 0.2m from its
    ↪ peak height),
the episode ends immediately. Episodes are truncated at 1000 timesteps if
    ↪ not terminated earlier.

    *position: indices from 0 to 21 -> obs[0:22]
    *velocity: indices from 22 to 44 -> obs[22:45]
    *com_inertia: indices from 45 to 184 -> obs[45:185]
    *com_velocity: indices from 185 to 268 -> obs[185:269]
    *actuator_forces: indices from 269 to 291 -> obs[269:292]
    *external_contact_forces: indices from 292 to 375 -> obs[292:376]


@property
def is_healthy(self):
```

```
        min_z, max_z = self._healthy_z_range
        is_healthy = min_z < self.data.qpos[2] < max_z

        return is_healthy


    def _get_obs(self):
        position = self.data.qpos.flat.copy()
        velocity = self.data.qvel.flat.copy()

        com_inertia = self.data.cinert.flat.copy()
        com_velocity = self.data.cvel.flat.copy()

        actuator_forces = self.data.qfrc_actuator.flat.copy()
        external_contact_forces = self.data.cfrc_ext.flat.copy()

        # x,y coordinates are excluded by default
            ↪  (exclude_current_positions_from_observation=True)
        position = position[2:]  # starts from z-coordinate

        observation = np.concatenate(
            (position, velocity, com_inertia, com_velocity, actuator_forces,
                ↪  external_contact_forces,)
        )

        return observation
```

Mutation prompt (`prompts/mutation_standup`)

```
You are given a reward function used to train a humanoid robot to stand up from a
    ↪ lying-down position. The reward functions has a fitness scores that
    ↪ reflects the agent's performance -- higher scores indicate superior
    ↪ standup behavior. Additionally, we tracked the values of the individual
    ↪ components in the reward function after every <EPISODES> episodes.

<EXAMPLES>

Your task is to iterate on the reward function by mutating a single component to
    ↪ enhance the agent's performance. Some helpful tips for analyzing the
    ↪ reward components:
If the values for a certain reward component are near identical throughout, then
    ↪ this means the training is not able to optimize this component as it is
    ↪ written. You may consider
(a) Rescaling it to a proper range or the value of its temperature parameter
(b) Re-writing the reward component
(c) Discarding the reward component

The mutation process involves the following steps:
First, select a reward component based on the given guidelines, human feedback,
    ↪ and the values of individual reward components.
Next, clearly explain how you intend to mutate the selected component and your
    ↪ rationale behind improving the performance. This could involve adjusting
    ↪ its scale, rewriting its formula, or other modifications.
Finally, write the mutated reward function code.

The output of the reward function should consist of two items:
(1) the total reward,
(2) a dictionary of each individual reward component.

The code output should be formatted as a python code string: "```python ... ```".

Some helpful tips for writing the reward function code:
```

```
1: You may find it helpful to normalize the reward to a fixed reward range like
    ↪ -1,1 or 0,1 by applying transformations like np.exp() to the overall
    ↪ reward or its components.
2: If you choose to transform a reward component, then you must introduce a
    ↪ temperature parameter inside the transformation function. This parameter
    ↪ must be a named variable in the reward function and it must not be an
    ↪ input variable. Each transformed reward component should have its own
    ↪ temperature variable and you must carefully set it's value based on its
    ↪ effect in the overall reward.
3: Make sure the type of each input variable is correctly specified.
4: Most importantly, the reward's code input variables must contain only
    ↪ attributes of the environment. Under no circumstance can you introduce a
    ↪ new input variable.
```

Crossover prompt (`prompts/crossover_standup`)

```
You are given a set of reward functions used to train humanoid robot to stand up
    ↪ from a lying-down position. The reward functions have fitness scores
    ↪ that reflect the agent's performance higher scores indicate superior
    ↪ standup behavior. For each reward function, we collected human feedback
    ↪ on what aspects of the agent are satisfactory and what aspects need
    ↪ improvement.  Additionally, we tracked the values of the individual
    ↪ components in the reward function after every <EPISODES> episodes.

<EXAMPLES>

Your task is to combine high-performing reward components from different reward
    ↪ functions and combine them to to enhance the agent's performance. Some
    ↪ helpful tips for analyzing the reward components:
If the values for a certain reward component are near identical throughout, then
    ↪ this means the training is not able to optimize this component as it is
    ↪ written.

The crossover process involves the following steps:
First, select high-performing reward components based on the human feedback and
    ↪ the values of individual reward components.
Next, clearly explain how you intend to combine them and your rationale behind
    ↪ improving the performance.
Finally, write the combined reward function code.

The output of the reward function should consist of two items:
1: the total reward,
2: a dictionary of each individual reward component.
The code output should be formatted as a python code string: "'''python ... '''".
1: You may find it helpful to normalize the reward to a fixed reward range like
    ↪ -1,1 or 0,1 by applying transformations like np.exp()to the overall
    ↪ reward or its components.
2: If you choose to transform a reward component, then you must introduce a
    ↪ temperature parameter inside the transformation function. This parameter
    ↪ must be a named variable in the reward function and it must not be an
    ↪ input variable. Each transformed reward component should have its own
    ↪ temperature variable and you must carefully set it's value based on its
    ↪ effect in the overall reward.
3: Make sure the type of each input variable is correctly specified.
4: Most importantly, the reward's code input variables must contain only
    ↪ attributes of the environment. The only input variables for the reward
    ↪ function that you can and must use are from the function def
    ↪ _get_obs(self) from the observation variable (the concatenated output so
    ↪ one single input in the compute reward).  Under no circumstance can you
    ↪ introduce a new input variable. or assume anything.
5. Make sure you dont give contradictory reward components.
```

J.3 Adroit Hand

System prompt (`prompts/system_prompt_adroit`)

```
You are a reward engineer trying to write a reward function for the environment
    ↪ that will help the agent learn the task described in text.

Task description:
The task involves using an anthropomorphic robotic hand to rotate a door handle
    ↪ and open the door as fast as possible. The robot must manipulate the
    ↪ handle effectively, overcoming the friction and resistance of the door
    ↪ mechanism. The episode ends after 400 steps or if it opened the door
    ↪ successfully.

Your reward function should use useful variables from the environment as inputs.
    ↪ An example reward function signature can be:

def compute_reward(object_pos: type, goal_pos: type) -> Tuple[type, Dict [str,
    ↪ type]]:
    ...
    return reward, {}


The output of the reward function should consist of two items:
1. The total reward,
2. A dictionary of each individual reward component.

The code output should be formatted as a Python code string: """python ... """.

1. You may find it helpful to normalize to a fixed reward range like -1,1 or 0,1
    ↪ by applying transformations like np.exp() to the overall reward and/or
    ↪ its reward components.
2. If you choose to transform a reward component, then you must introduce a
    ↪ temperature parameter inside the transformation function. This parameter
    ↪ must be a named variable in the reward function and it must not be an
    ↪ input variable.
3. Make sure the type of each input variable is correctly specified.
4. Most importantly, the reward's code input variables must contain only
    ↪ attributes of the environment. Under no circumstance can you introduce a
    ↪ new input variable and assume variables that are not available.
5. Your output must always be the def function code with the output being the
    ↪ total reward.
6. Make sure you don't give contradictory reward components.
```

Env input (`prompts/env_input_adroit`)

```
class AdroitHandDoorEnv(MujocoEnv, EzPickle):
    """
    ## Description

    The environment is based on the Adroit manipulation platform, a 28 degree of
        ↪ freedom system
    which consists of a 24 degrees of freedom ShadowHand and a 4 degree of
        ↪ freedom arm.
    The task to be completed consists on undoing the latch and swing the door
        ↪ open.
    The latch has significant dry friction and a bias torque that forces the
        ↪ door to stay closed.
    The position of the door is randomized. Task is considered complete when the
        ↪ door touches
    the door stopper at the other end (door_hinge >= 1.35 rad).

    ## Action Space
```

```
The action space is a `Box(-1.0, 1.0, (28,), float32)`. The control actions
    ↪ are absolute
angular positions of the Adroit hand joints. The input of the control
    ↪ actions is set to a
range between -1 and 1 by scaling the real actuator angle ranges in radians.

## Observation Space

The observation space is of the type `Box(-inf, inf, (39,), float64)`. It
    ↪ contains information
about the angular position of the finger joints, the pose of the palm of the
    ↪ hand, the position
of the door handle, as well as state of the latch and door.

| Num | Observation
    ↪ | Min  | Max | Unit             |
|-----|-------------------------------------------------------------------┘
    ↪ --|------|-----|------------------|
| 0-26  | Angular positions of 27 hand joints (arm + wrist + fingers)       |
    ↪ -Inf | Inf | angle (rad)      |
| 27    | Angular position of the door latch                                |
    ↪ -Inf | Inf | angle (rad)      |
| 28    | Angular position of the door hinge                                |
    ↪ -Inf | Inf | angle (rad)      |
| 29-31 | Position of the center of the palm (x, y, z)                      |
    ↪ -Inf | Inf | position (m)     |
| 32-34 | Position of the door handle (x, y, z)                             |
    ↪ -Inf | Inf | position (m)     |
| 35-37 | Positional difference from palm to door handle (x, y, z)          |
    ↪ -Inf | Inf | position (m)     |
| 38    | Door open indicator (1.0 if open, -1.0 if closed)                 |
    ↪ -1   | 1   | binary           |

## Available Reward Function Variables

The following named variables are available as input arguments to
    ↪ `compute_reward`.
You must ONLY use these variable names as function parameters:

- `hand_qpos`: np.ndarray of shape (27,) - angular positions of all hand
    ↪ joints (rad)
- `latch_pos`: float - angular position of the door latch (rad)
- `door_pos`: float - angular position of the door hinge (rad). Door is open
    ↪ when >= 1.35
- `door_hinge`: float - same as door_pos
- `palm_pos`: np.ndarray of shape (3,) - xyz position of the center of the
    ↪ palm (m)
- `handle_pos`: np.ndarray of shape (3,) - xyz position of the door handle
    ↪ (m)
- `palm_handle_delta`: np.ndarray of shape (3,) - vector from handle to
    ↪ palm: palm_pos - handle_pos (m)
- `door_open`: float - 1.0 if the door is open (hinge > 1.0 rad), -1.0
    ↪ otherwise
- `action`: np.ndarray of shape (28,) - control actions applied at the
    ↪ current step
- `joint_velocities`: np.ndarray of shape (30,) - angular velocities of all
    ↪ joints (rad/s)
- `joint_forces`: np.ndarray of shape (28,) - forces/torques applied by each
    ↪ actuator

## Episode End

The episode ends when the door hinge angle >= 1.35 radians (success) or
    ↪ after 400 timesteps.
```

```
    """

    def _get_obs(self):
        qpos = self.data.qpos.ravel()
        handle_pos = self.data.site_xpos[self.handle_site_id].ravel()
        palm_pos = self.data.site_xpos[self.grasp_site_id].ravel()
        door_pos = np.array([self.data.qpos[self.door_hinge_addrs]])
        if door_pos > 1.0:
            door_open = 1.0
        else:
            door_open = -1.0
        latch_pos = qpos[-1]

        return np.concatenate(
            [
                qpos[1:-2],
                [latch_pos],
                door_pos,
                palm_pos,
                handle_pos,
                palm_pos - handle_pos,
                [door_open],
            ]
        )
```

Mutation prompt (`prompts/mutation_adroit`)

```
You are given a reward function used to train a robotic hand to open a door. The
    ↪ reward functions has a fitness scores that reflects the agent's
    ↪ performance -- higher scores indicate the agent opens the door faster
    ↪ and more reliably. Additionally, we tracked the values of the individual
    ↪ components in the reward function after every <EPISODES> episodes.

<EXAMPLES>

Your task is to iterate on the reward function by mutating a single component to
    ↪ enhance the agent's performance. Some helpful tips for analyzing the
    ↪ reward components:
If the values for a certain reward component are near identical throughout, then
    ↪ this means the training is not able to optimize this component as it is
    ↪ written. You may consider
(a) Rescaling it to a proper range or the value of its temperature parameter
(b) Re-writing the reward component
(c) Discarding the reward component

The mutation process involves the following steps:
First, select a reward component based on the given guidelines, human feedback,
    ↪ and the values of individual reward components.
Next, clearly explain how you intend to mutate the selected component and your
    ↪ rationale behind improving the performance. This could involve adjusting
    ↪ its scale, rewriting its formula, or other modifications.
Finally, write the mutated reward function code.

The output of the reward function should consist of two items:
(1) the total reward,
(2) a dictionary of each individual reward component.

The code output should be formatted as a python code string: "```python ... ```".

Some helpful tips for writing the reward function code:
1: You may find it helpful to normalize the reward to a fixed reward range like
    ↪ -1,1 or 0,1 by applying transformations like np.exp() to the overall
    ↪ reward or its components.
```

```
2: If you choose to transform a reward component, then you must introduce a
    ↪ temperature parameter inside the transformation function. This parameter
    ↪ must be a named variable in the reward function and it must not be an
    ↪ input variable. Each transformed reward component should have its own
    ↪ temperature variable and you must carefully set it's value based on its
    ↪ effect in the overall reward.
3: Make sure the type of each input variable is correctly specified.
4: Most importantly, the reward's code input variables must contain only
    ↪ attributes of the environment. Under no circumstance can you introduce a
    ↪ new input variable.
```

Crossover prompt (`prompts/crossover_adroit`)

```
You are given a set of reward functions used to train a robotic hand to open a
    ↪ door. The reward functions have fitness scores that reflect the agent's
    ↪ performance -- higher scores indicate the agent opens the door faster
    ↪ and more reliably. For each reward function, we collected human feedback
    ↪ on what aspects of the agent are satisfactory and what aspects need
    ↪ improvement. Additionally, we tracked the values of the individual
    ↪ components in the reward function after every <EPISODES> episodes.

<EXAMPLES>

Your task is to combine high-performing reward components from different reward
    ↪ functions and combine them to enhance the agent's performance. Some
    ↪ helpful tips for analyzing the reward components:
If the values for a certain reward component are near identical throughout, then
    ↪ this means the training is not able to optimize this component as it is
    ↪ written.

The crossover process involves the following steps:
First, select high-performing reward components based on the human feedback and
    ↪ the values of individual reward components.
Next, clearly explain how you intend to combine them and your rationale behind
    ↪ improving the performance.
Finally, write the combined reward function code.

The output of the reward function should consist of two items:
1: the total reward,
2: a dictionary of each individual reward component.
The code output should be formatted as a python code string: "'''python ... '''".
1: You may find it helpful to normalize the reward to a fixed reward range like
    ↪ -1,1 or 0,1 by applying transformations like np.exp() to the overall
    ↪ reward or its components.
2: If you choose to transform a reward component, then you must introduce a
    ↪ temperature parameter inside the transformation function. This parameter
    ↪ must be a named variable in the reward function and it must not be an
    ↪ input variable. Each transformed reward component should have its own
    ↪ temperature variable and you must carefully set it's value based on its
    ↪ effect in the overall reward.
3: Make sure the type of each input variable is correctly specified.
4: Most importantly, the reward's code input variables must contain only
    ↪ attributes of the environment. Under no circumstance can you introduce a
    ↪ new input variable.
5. Make sure you dont give contradictory reward components.
```

K Discovered Reward Function Sequences

This appendix lists the complete winning-ancestry reward sequences $r_0, r_1, \ldots, r_G$ that FORGE (island) discovered on each task and that drive the replay ablation in Figure 7. *Forward* applies them in the order below, *shuffle* uses a uniformly random permutation, and *skip* uses three sparse rewards anchored at the converged-generation reward (per-task indices in Fig. 7). Each reward is

a Python function mapping an observation tuple to a scalar signal; fitness σ is a separate external metric (Appendix D).

The three trajectories also illustrate how ordering carries information, and each environment does so through a different mechanism. Together they support the "trajectory is load-bearing" claim of Finding 3 (Section 4.5) and the implicit-curriculum interpretation of Section 6:

- **Humanoid Locomotion:** *qualitative reformulation at a single generation*. r_2 diagnoses a numerical failure in the earlier height reward and rewrites the formulation; every later generation builds on that scaffold.
- **Humanoid Standup:** *recombination rather than linear refinement*. Two within-sequence regressions are closed by an r_5 breakthrough that reuses elements from earlier generations rather than refining its immediate parent.
- **Adroit Hand:** *incremental refinement of a shared template*. No single generation dominates; fitness 0.78 is reached by r_3 through an ordered schedule of temperature, weight, and formulation tweaks.

Each subsection below opens with a *Reading the shaping trajectory* paragraph walking through the design moves, followed by the Python source of every r_g.

K.1 Humanoid Locomotion

We show generations 0–5, covering the rising phase up to the peak; generation 6 onward begins the post-peak plasticity-loss decline characterized in Appendix I and is thus omitted here.

Reading the shaping trajectory. The Humanoid Locomotion trajectory is dominated by one qualitative reformulation rather than cumulative tuning: generation 2 diagnoses a numerical failure in the height reward, producing a large fitness jump that carries the rest of the sequence. Fitness climbs through two peaks at r_2 and r_5 separated by an r_4 dip, with the dip closed by r_5's recombination of components that survived earlier evaluation. Four design moves are visible:

- **Height reformulation (the driver).** r_0–r_1 use a Gaussian $\exp(-(z-1.4)^2/\tau)$ that assumes `torso_z` near nominal scale. r_2's source comment flags the failure, "your logs show `torso_z` is enormous, making the old Gaussian effectively 0 always", and replaces it with a scale-robust band gate $\exp(-\max(0, |z-z_{\text{mid}}|-h)^2/\tau^2)$ that is flat inside $[1, 2]$. r_3 and r_5 inherit it; r_4 reverts to Gaussian and the score drops.
- **Symmetry paradigm shift.** r_0 rewards gait through knee and elbow velocities with a target swing speed; r_1 drops gait entirely; r_2 reintroduces it as joint-angle anti-symmetry $(q_R+q_L)^2 + (q_{R,\text{knee}}-q_{L,\text{knee}})^2$ with an abdomen-angle posture term. The angle-based formulation survives all later generations.
- **Speed-reward try-and-revert.** r_0–r_2 use saturating $1-\exp(-v_x/\tau)$; r_3 switches to log-shaped $\log(1+v_x/\tau)$ to preserve gradient at high v_x; r_4 reverts and adds a cap penalty at $v_x>6$; r_5 drops the cap and instead penalizes only negative v_x.
- **Selective log-compression of penalties.** r_2 switches effort to $\log(1+\|u\|_2)$ for robustness under large actuator scales; r_4 extends log-compression to impact; r_5 keeps log-effort but drops log-impact.

Generation 0 (r_0)

```
def compute_reward(obs: np.ndarray) -> Tuple[float, Dict[str, float]]:
    # --------------------------
    # Extract core signals
    # --------------------------
    z_torso = float(obs[0])           # torso height
    torso_quat = obs[1:5]             # (x,y,z,w) orientation quaternion
    v_forward = float(obs[22])        # x-velocity of torso (forward speed)
    v_lateral = float(obs[23])        # y-velocity (sideways)
    v_vertical = float(obs[24])       # z-velocity
    w_torso = obs[25:28]              # torso angular velocity (roll/pitch/yaw
        ↪ rates)
```

```
qfrc_act = obs[269:292]           # actuator forces proxy (energy/effort)
cfrc_ext = obs[292:376].reshape(14, 6)  # external contact forces/torques
    ↪ per body

# --------------------------
# Temperatures (for exp transforms)
# --------------------------
temp_speed = 2.0          # bigger -> broader reward for speed
temp_upright = 0.25       # tighter upright constraint
temp_height = 0.08        # tightness around target torso height
temp_side = 0.75          # penalize sideways drift
temp_vert = 0.6           # penalize bouncing (vertical vel)
temp_angvel = 2.5         # penalize excessive torso rotation
temp_effort = 120.0       # scale for actuator-force magnitude
temp_impact = 250.0       # scale for external contact (impact) magnitude
temp_gait = 1.2           # symmetry regularizer tolerance
temp_arm = 1.8            # arm swing regularizer tolerance

# --------------------------
# 1) Primary objective: forward running speed (bounded in [0,1])
# --------------------------
# Encourage forward speed up to around ~4-5 m/s; negative speeds get ~0
    ↪ reward.
v_pos = max(0.0, v_forward)
r_speed = 1.0 - np.exp(-v_pos / temp_speed)

# --------------------------
# 2) Stay healthy (z in [1,2]) and prefer a nominal running height
# --------------------------
# Use a soft penalty that becomes strong outside the healthy range.
z_min, z_max = 1.0, 2.0
z_target = 1.4

# distance outside healthy interval
z_violation = max(0.0, z_min - z_torso) + max(0.0, z_torso - z_max)
# height shaping around target
r_height = np.exp(-((z_torso - z_target) ** 2) / temp_height)
# violation penalty (0 when inside range; approaches -1 as violation grows)
p_unhealthy = -(1.0 - np.exp(-z_violation / 0.05))

# --------------------------
# 3) Upright torso (human-like running posture)
# --------------------------
# For a unit quaternion (x,y,z,w), upright corresponds to small x/y
    ↪ components.
# Penalize tilt magnitude = sqrt(x^2 + y^2).
tilt = float(np.sqrt(torso_quat[0] ** 2 + torso_quat[1] ** 2))
r_upright = np.exp(-(tilt ** 2) / temp_upright)

# --------------------------
# 4) Reduce wasted motion: lateral/vertical velocity and excessive spinning
# --------------------------
r_no_side = np.exp(-(v_lateral ** 2) / temp_side)
r_no_bounce = np.exp(-(v_vertical ** 2) / temp_vert)
r_stable_rot = np.exp(-(float(np.sum(w_torso ** 2))) / temp_angvel)

# --------------------------
# 5) Efficiency: penalize large actuator forces
# --------------------------
# Use L2 magnitude; shaped into (0..1] then converted to a penalty in [-1..0]
act_mag2 = float(np.mean(qfrc_act ** 2))
r_effort = np.exp(-act_mag2 / temp_effort)          # 1 is best
p_effort = -(1.0 - r_effort)                        # 0 best, down to ~-1
```

```
    # --------------------------
    # 6) Smooth contacts: penalize large external contact forces/torques
        ↪ (impacts)
    # --------------------------
    # Ignore worldbody row=0 (all zeros); focus on robot bodies (1..13).
    ext = cfrc_ext[1:, :]
    impact_mag2 = float(np.mean(ext ** 2))
    r_soft_contact = np.exp(-impact_mag2 / temp_impact)
    p_impact = -(1.0 - r_soft_contact)

    # --------------------------
    # 7) Simple "human-like gait" symmetry regularizers (soft)
    # --------------------------
    # Knee velocities (indices in velocity block):
    # right_knee vel at obs[22+12]=34, left_knee vel at obs[22+16]=38
    right_knee_vel = float(obs[34])
    left_knee_vel = float(obs[38])
    # Encourage alternating/symmetric magnitudes (not necessarily equal sign).
    r_gait_sym = np.exp(-((abs(right_knee_vel) - abs(left_knee_vel)) ** 2) /
        ↪ temp_gait)

    # Arm swing presence: encourage some elbow motion but not extreme.
    # right_elbow vel obs[22+19]=41, left_elbow vel obs[22+22]=44
    right_elbow_vel = float(obs[41])
    left_elbow_vel = float(obs[44])
    elbow_speed = 0.5 * (abs(right_elbow_vel) + abs(left_elbow_vel))
    # Reward moderate elbow speed: peak around ~1.5 rad/s
    elbow_target = 1.5
    r_arm_swing = np.exp(-((elbow_speed - elbow_target) ** 2) / temp_arm)

    # --------------------------
    # Combine (keep total roughly in a stable range)
    # --------------------------
    # Speed is the main driver; regularizers prevent falling/unnatural thrashing.
    reward = (
        2.5 * r_speed
        + 0.6 * r_height
        + 0.8 * r_upright
        + 0.3 * r_no_side
        + 0.3 * r_no_bounce
        + 0.3 * r_stable_rot
        + 0.2 * r_gait_sym
        + 0.2 * r_arm_swing
        + 0.6 * p_effort
        + 0.6 * p_impact
        + 2.0 * p_unhealthy
    )

    # Mild squashing to keep reward bounded ~(-1, 1) for convenience
    temp_total = 3.0
    reward = float(np.tanh(reward / temp_total))

    components: Dict[str, float] = {
        "r_speed": float(r_speed),
        "r_height": float(r_height),
        "p_unhealthy": float(p_unhealthy),
        "r_upright": float(r_upright),
        "r_no_side": float(r_no_side),
        "r_no_bounce": float(r_no_bounce),
        "r_stable_rot": float(r_stable_rot),
        "p_effort": float(p_effort),
        "p_impact": float(p_impact),
        "r_gait_sym": float(r_gait_sym),
        "r_arm_swing": float(r_arm_swing),
        "reward_tanh": float(reward),
```

```
    }
    return reward, components
```

Generation 1 (r_1)

```
def compute_reward(obs: np.ndarray) -> Tuple[float, Dict[str, float]]:
    # ----------------------------
    # Guaranteed indices from spec
    # ----------------------------
    torso_z = float(obs[0])          # torso z-position
    vx = float(obs[22])              # torso/root x-velocity
    vy = float(obs[23])              # torso/root y-velocity
    vz = float(obs[24])              # torso/root z-velocity
    ang = obs[25:28]                 # root angular velocity (wx, wy, wz)

    u = obs[269:292]                 # actuator forces (23,)
    cfrc = obs[292:376].reshape(14, 6)  # external contact forces per body (14,6)

    # Hands/arms: right_lower_arm=11, left_lower_arm=13 (per description table)
    hand_force_mag = float(
        np.linalg.norm(cfrc[11, 0:3]) + np.linalg.norm(cfrc[13, 0:3])
    )

    # ----------------------------
    # Temperatures (named constants)
    # ----------------------------
    temp_speed = 2.2          # speed saturation (~2-6 m/s in many humanoid
        ↪ tasks)
    temp_height = 0.18        # how tight we want height around target
    temp_lateral = 0.6        # sideways drift tolerance
    temp_vertical = 0.7       # vertical bounce tolerance
    temp_ang = 2.0            # torso spin tolerance
    temp_effort = 20000.0     # scale for mean(u^2) (keep non-saturated)
    temp_hands = 60.0         # discourage using arms/hands to support/propel

    # ----------------------------
    # 1) Forward speed (main)
    # ----------------------------
    vx_pos = max(0.0, vx)
    r_speed = 1.0 - np.exp(-vx_pos / temp_speed)  # in [0,1)

    # ----------------------------
    # 2) Upright / healthy height shaping
    # ----------------------------
    z_target = 1.45
    r_height = float(np.exp(-((torso_z - z_target) ** 2) / (2.0 * temp_height **
        ↪ 2)))

    unhealthy = float((torso_z <= 1.0) or (torso_z >= 2.0))
    p_unhealthy = -unhealthy  # 0 if healthy else -1

    # ----------------------------
    # 3) Smooth motion / straight running
    # ----------------------------
    r_lateral = float(np.exp(-abs(vy) / temp_lateral))
    r_vertical = float(np.exp(-abs(vz) / temp_vertical))

    ang_speed = float(np.linalg.norm(ang))
    r_stability = float(np.exp(-ang_speed / temp_ang))

    # ----------------------------
    # 4) Energy regularization (bounded, non-saturating)
    # ----------------------------
    effort_mse = float(np.mean(np.square(u)))
    r_effort = float(np.exp(-effort_mse / temp_effort))
```

```
    # ----------------------------
    # 5) Human-like contact: discourage hand/arm impacts
    # ----------------------------
    r_no_hands = float(np.exp(-hand_force_mag / temp_hands))

    # ----------------------------
    # Combine (bounded ~[-1, 1])
    # ----------------------------
    w_speed = 0.58
    w_height = 0.16
    w_lat = 0.07
    w_vert = 0.06
    w_stab = 0.07
    w_eff = 0.04
    w_no_hands = 0.02

    shaped = (
        w_speed * r_speed
        + w_height * r_height
        + w_lat * r_lateral
        + w_vert * r_vertical
        + w_stab * r_stability
        + w_eff * r_effort
        + w_no_hands * r_no_hands
    )

    reward = float(np.clip(shaped + 0.6 * p_unhealthy, -1.0, 1.0))

    info: Dict[str, float] = {
        "r_speed": float(r_speed),
        "r_height": float(r_height),
        "p_unhealthy": float(p_unhealthy),
        "r_lateral": float(r_lateral),
        "r_vertical": float(r_vertical),
        "r_stability": float(r_stability),
        "r_effort": float(r_effort),
        "r_no_hands": float(r_no_hands),
        "vx": float(vx),
        "vy": float(vy),
        "vz": float(vz),
        "torso_z": float(torso_z),
        "ang_speed": float(ang_speed),
        "effort_mse": float(effort_mse),
        "hand_force_mag": float(hand_force_mag),
    }
    return reward, info
```

Generation 2 (r_2)

```
def compute_reward(observation: np.ndarray) -> Tuple[float, Dict[str, float]]:
    obs = observation

    # -------------------------
    # Key signals from obs
    # -------------------------
    torso_z = float(obs[0])

    vx = float(obs[22])
    vy = float(obs[23])
    vz = float(obs[24])

    wx = float(obs[25])
    wy = float(obs[26])
    wz = float(obs[27])
```

```
    abdomen_z = float(obs[5])
    abdomen_y = float(obs[6])
    abdomen_x = float(obs[7])

    r_hip_x, r_hip_z, r_hip_y, r_knee = float(obs[8]), float(obs[9]),
        ↪ float(obs[10]), float(obs[11])
    l_hip_x, l_hip_z, l_hip_y, l_knee = float(obs[12]), float(obs[13]),
        ↪ float(obs[14]), float(obs[15])

    r_sh1, r_sh2, r_elb = float(obs[16]), float(obs[17]), float(obs[18])
    l_sh1, l_sh2, l_elb = float(obs[19]), float(obs[20]), float(obs[21])

    act = obs[269:292]
    act_l2 = float(np.sqrt(np.mean(np.square(act)) + 1e-8))

    cfrc = obs[292:376]
    cfrc_rms = float(np.sqrt(np.mean(np.square(cfrc)) + 1e-8))

    # -------------------------
    # Transformed components with explicit temperatures
    # -------------------------

    # 1) Forward speed
    t_speed = 2.5
    speed_reward = 1.0 - np.exp(-max(0.0, vx) / t_speed)  # [0,1)

    # 2) Height / health (MUTATED)
    # Old formulation assumed torso_z is in meters near ~1.5; your logs show
        ↪ torso_z is enormous,
    # making the old Gaussian effectively 0 always (or numerically meaningless
        ↪ depending on data issues).
    # New: a robust "healthy range gate" around [1,2] with a wide temperature so
        ↪ it remains shapeable
    # even under moderate scaling issues. Bounded in [0,1].
    healthy_min_z, healthy_max_z = 1.0, 2.0
    z_mid = 0.5 * (healthy_min_z + healthy_max_z)
    half_range = 0.5 * (healthy_max_z - healthy_min_z)

    # Temperature chosen to be lenient (prevents collapse if z is slightly
        ↪ noisy/misscaled),
    # yet still provides gradient toward the healthy band when values are
        ↪ near-scale.
    t_height_gate = 1.5
    # distance outside the band; 0 inside
    z_out = max(0.0, abs(torso_z - z_mid) - half_range)
    height_reward = float(np.exp(-(z_out ** 2) / (t_height_gate ** 2)))  # [0,1]

    # 3) Drift
    t_drift = 1.0
    drift_penalty = 1.0 - float(np.exp(-(vy * vy + vz * vz) / (t_drift ** 2)))  #
        ↪ [0,1)

    # 4) Angular velocity
    t_angvel = 4.0
    angvel_mag2 = wx * wx + wy * wy + wz * wz
    angvel_penalty = 1.0 - float(np.exp(-angvel_mag2 / (t_angvel ** 2)))  # [0,1)

    # 5) Symmetry
    t_sym = 0.8
    leg_diff2 = ((r_hip_x + l_hip_x) ** 2 +
                 (r_hip_y + l_hip_y) ** 2 +
                 (r_hip_z + l_hip_z) ** 2 +
                 (r_knee  - l_knee)  ** 2)
    arm_diff2 = ((r_sh1 + l_sh1) ** 2 +
```

```
            (r_sh2 + l_sh2) ** 2 +
            (r_elb - l_elb) ** 2)
    symmetry_penalty = 1.0 - float(np.exp(-(leg_diff2 + 0.5 * arm_diff2) /
        ↪  (t_sym ** 2)))  # [0,1)

    # 6) Posture
    t_posture = 0.7
    posture_mag2 = abdomen_x**2 + abdomen_y**2 + 0.5 * abdomen_z**2
    posture_penalty = 1.0 - float(np.exp(-posture_mag2 / (t_posture ** 2)))  #
        ↪  [0,1)

    # 7) Effort penalty (mutated previously)
    t_effort_log = 3.0
    act_log = float(np.log1p(act_l2))
    effort_penalty = 1.0 - float(np.exp(-((act_log ** 2) / (t_effort_log **
        ↪  2))))  # [0,1)

    # 8) Impact penalty
    t_impact = 15.0
    impact_penalty = 1.0 - float(np.exp(-(cfrc_rms ** 2) / (t_impact ** 2)))  #
        ↪  [0,1)

    # -------------------------
    # Combine
    # -------------------------
    w_speed = 1.30
    w_height = 0.40

    w_drift = 0.30
    w_angvel = 0.25
    w_sym = 0.20
    w_posture = 0.15
    w_effort = 0.30
    w_impact = 0.25

    reward = (
        w_speed * float(speed_reward)
        + w_height * float(height_reward)
        - w_drift * float(drift_penalty)
        - w_angvel * float(angvel_penalty)
        - w_sym * float(symmetry_penalty)
        - w_posture * float(posture_penalty)
        - w_effort * float(effort_penalty)
        - w_impact * float(impact_penalty)
    )

    components = {
        "speed_reward": float(speed_reward),
        "height_reward": float(height_reward),
        "drift_penalty": float(drift_penalty),
        "angvel_penalty": float(angvel_penalty),
        "symmetry_penalty": float(symmetry_penalty),
        "posture_penalty": float(posture_penalty),
        "effort_penalty": float(effort_penalty),
        "impact_penalty": float(impact_penalty),
        "vx": float(vx),
        "torso_z": float(torso_z),
        "act_l2": float(act_l2),
        "act_log": float(act_log),
        "cfrc_rms": float(cfrc_rms),
        "total_reward": float(reward),
    }

    return float(reward), components
```

Generation 3 (r_3)

```
def compute_reward(observation: np.ndarray) -> Tuple[float, Dict[str, float]]:
    obs = observation

    # -------------------------
    # Key signals from obs
    # -------------------------
    torso_z = float(obs[0])

    vx = float(obs[22])
    vy = float(obs[23])
    vz = float(obs[24])

    wx = float(obs[25])
    wy = float(obs[26])
    wz = float(obs[27])

    abdomen_z = float(obs[5])
    abdomen_y = float(obs[6])
    abdomen_x = float(obs[7])

    r_hip_x, r_hip_z, r_hip_y, r_knee = float(obs[8]), float(obs[9]),
        ↪ float(obs[10]), float(obs[11])
    l_hip_x, l_hip_z, l_hip_y, l_knee = float(obs[12]), float(obs[13]),
        ↪ float(obs[14]), float(obs[15])

    r_sh1, r_sh2, r_elb = float(obs[16]), float(obs[17]), float(obs[18])
    l_sh1, l_sh2, l_elb = float(obs[19]), float(obs[20]), float(obs[21])

    act = obs[269:292]
    act_l2 = float(np.sqrt(np.mean(np.square(act)) + 1e-8))

    cfrc = obs[292:376]
    cfrc_rms = float(np.sqrt(np.mean(np.square(cfrc)) + 1e-8))

    # -------------------------
    # Transformed components with explicit temperatures
    # -------------------------

    # 1) Forward speed (MUTATED): log-shaped, bounded, avoids early saturation
    # Keeps gradient for large vx (your logs show vx up to ~5000).
    t_speed_log = 50.0  # temperature controls how quickly log grows; larger =>
        ↪ less sensitive at low speeds
    v_ref = 300.0       # reference speed that maps close to 1.0 (not an input;
        ↪ fixed shaping constant)

    vpos = max(0.0, vx)
    speed_raw = np.log1p(vpos / t_speed_log)
    speed_norm = speed_raw / (np.log1p(v_ref / t_speed_log) + 1e-8)
    speed_reward = float(np.clip(speed_norm, 0.0, 1.0))  # [0,1]

    # 2) Height / health
    healthy_min_z, healthy_max_z = 1.0, 2.0
    z_mid = 0.5 * (healthy_min_z + healthy_max_z)
    half_range = 0.5 * (healthy_max_z - healthy_max_z + (healthy_max_z -
        ↪ healthy_min_z))  # keep identical behavior
    t_height_gate = 1.5
    z_out = max(0.0, abs(torso_z - z_mid) - half_range)
    height_reward = float(np.exp(-(z_out ** 2) / (t_height_gate ** 2)))  # [0,1]

    # 3) Drift
    t_drift = 1.0
    drift_penalty = 1.0 - float(np.exp(-(vy * vy + vz * vz) / (t_drift ** 2)))  #
        ↪ [0,1)
```

```
    # 4) Angular velocity
    t_angvel = 4.0
    angvel_mag2 = wx * wx + wy * wy + wz * wz
    angvel_penalty = 1.0 - float(np.exp(-angvel_mag2 / (t_angvel ** 2)))  # [0,1)

    # 5) Symmetry
    t_sym = 0.8
    leg_diff2 = ((r_hip_x + l_hip_x) ** 2 +
                 (r_hip_y + l_hip_y) ** 2 +
                 (r_hip_z + l_hip_z) ** 2 +
                 (r_knee  - l_knee)  ** 2)
    arm_diff2 = ((r_sh1 + l_sh1) ** 2 +
                 (r_sh2 + l_sh2) ** 2 +
                 (r_elb - l_elb) ** 2)
    symmetry_penalty = 1.0 - float(np.exp(-(leg_diff2 + 0.5 * arm_diff2) /
        ↪  (t_sym ** 2)))  # [0,1)

    # 6) Posture
    t_posture = 0.7
    posture_mag2 = abdomen_x**2 + abdomen_y**2 + 0.5 * abdomen_z**2
    posture_penalty = 1.0 - float(np.exp(-posture_mag2 / (t_posture ** 2)))  #
        ↪  [0,1)

    # 7) Effort penalty
    t_effort_log = 3.0
    act_log = float(np.log1p(act_l2))
    effort_penalty = 1.0 - float(np.exp(-((act_log ** 2) / (t_effort_log **
        ↪  2))))  # [0,1)

    # 8) Impact penalty
    t_impact = 15.0
    impact_penalty = 1.0 - float(np.exp(-(cfrc_rms ** 2) / (t_impact ** 2)))  #
        ↪  [0,1)

    # -------------------------
    # Combine
    # -------------------------
    w_speed = 1.30
    w_height = 0.40

    w_drift = 0.30
    w_angvel = 0.25
    w_sym = 0.20
    w_posture = 0.15
    w_effort = 0.30
    w_impact = 0.25

    reward = (
        w_speed * float(speed_reward)
        + w_height * float(height_reward)
        - w_drift * float(drift_penalty)
        - w_angvel * float(angvel_penalty)
        - w_sym * float(symmetry_penalty)
        - w_posture * float(posture_penalty)
        - w_effort * float(effort_penalty)
        - w_impact * float(impact_penalty)
    )

    components = {
        "speed_reward": float(speed_reward),
        "height_reward": float(height_reward),
        "drift_penalty": float(drift_penalty),
        "angvel_penalty": float(angvel_penalty),
        "symmetry_penalty": float(symmetry_penalty),
```

```
        "posture_penalty": float(posture_penalty),
        "effort_penalty": float(effort_penalty),
        "impact_penalty": float(impact_penalty),
        "vx": float(vx),
        "torso_z": float(torso_z),
        "act_l2": float(act_l2),
        "act_log": float(act_log),
        "cfrc_rms": float(cfrc_rms),
        "total_reward": float(reward),
    }

    return float(reward), components
```

Generation 4 (r_4)

```
def compute_reward(observation: np.ndarray) -> Tuple[float, Dict[str, float]]:
    obs: np.ndarray = observation

    # -------------------------
    # Key signals from obs
    # -------------------------
    torso_z: float = float(obs[0])

    vx: float = float(obs[22])
    vy: float = float(obs[23])
    vz: float = float(obs[24])

    wx: float = float(obs[25])
    wy: float = float(obs[26])
    wz: float = float(obs[27])

    abdomen_z: float = float(obs[5])
    abdomen_y: float = float(obs[6])
    abdomen_x: float = float(obs[7])

    r_hip_x, r_hip_z, r_hip_y, r_knee = float(obs[8]), float(obs[9]),
        ↪ float(obs[10]), float(obs[11])
    l_hip_x, l_hip_z, l_hip_y, l_knee = float(obs[12]), float(obs[13]),
        ↪ float(obs[14]), float(obs[15])

    r_sh1, r_sh2, r_elb = float(obs[16]), float(obs[17]), float(obs[18])
    l_sh1, l_sh2, l_elb = float(obs[19]), float(obs[20]), float(obs[21])

    act: np.ndarray = obs[269:292]
    act_l2: float = float(np.sqrt(np.mean(np.square(act)) + 1e-8))

    cfrc: np.ndarray = obs[292:376]
    cfrc_rms: float = float(np.sqrt(np.mean(np.square(cfrc)) + 1e-8))

    # -------------------------
    # Reward components (bounded via exp transforms; each has its own
        ↪ temperature)
    # -------------------------

    # (A) Forward speed (saturating, positive only)
    t_speed: float = 2.2
    vx_pos: float = float(max(0.0, vx))
    speed_reward: float = float(1.0 - np.exp(-vx_pos / t_speed))  # ~[0,1)

    # (B) Soft speed cap penalty (activate earlier than before to avoid vx
        ↪ explosions)
    vx_cap: float = 6.0
    t_vx_cap: float = 1.8
    over_vx: float = float(max(0.0, vx_pos - vx_cap))
```

```
    speed_cap_penalty: float = float(1.0 - np.exp(-(over_vx ** 2) / (t_vx_cap **
        ↪ 2)))  # [0,1)

    # (C) Height shaping (prefer mid of healthy band)
    z_target: float = 1.5
    t_height: float = 0.22
    height_reward: float = float(np.exp(-((torso_z - z_target) ** 2) / (2.0 *
        ↪ (t_height ** 2))))  # (0,1]

    # (D) Unhealthy penalty (distance outside [1,2])
    z_min: float = 1.0
    z_max: float = 2.0
    t_unhealthy: float = 0.12
    z_violation: float = float(max(0.0, z_min - torso_z) + max(0.0, torso_z -
        ↪ z_max))
    unhealthy_penalty: float = float(1.0 - np.exp(-(z_violation ** 2) /
        ↪ (t_unhealthy ** 2)))  # [0,1)

    # (E) Drift penalty (avoid sideways/up-down translation)
    t_drift: float = 0.9
    drift_penalty: float = float(1.0 - np.exp(-(vy * vy + vz * vz) / (t_drift **
        ↪ 2)))  # [0,1)

    # (F) Angular velocity penalty (torso stability)
    t_angvel: float = 4.2
    angvel_mag2: float = float(wx * wx + wy * wy + wz * wz)
    angvel_penalty: float = float(1.0 - np.exp(-angvel_mag2 / (t_angvel ** 2)))
        ↪ # [0,1)

    # (G) Symmetry penalty (human-like gait)
    t_sym: float = 0.85
    leg_diff2: float = float(
        (r_hip_x + l_hip_x) ** 2 +
        (r_hip_y + l_hip_y) ** 2 +
        (r_hip_z + l_hip_z) ** 2 +
        (r_knee  - l_knee)  ** 2
    )
    arm_diff2: float = float(
        (r_sh1 + l_sh1) ** 2 +
        (r_sh2 + l_sh2) ** 2 +
        (r_elb - l_elb) ** 2
    )
    symmetry_penalty: float = float(1.0 - np.exp(-(leg_diff2 + 0.45 * arm_diff2)
        ↪ / (t_sym ** 2)))  # [0,1)

    # (H) Posture penalty (avoid extreme abdomen bending/twisting)
    t_posture: float = 0.75
    posture_mag2: float = float(abdomen_x**2 + abdomen_y**2 + 0.5 * abdomen_z**2)
    posture_penalty: float = float(1.0 - np.exp(-posture_mag2 / (t_posture **
        ↪ 2)))  # [0,1)

    # (I) Effort penalty (log-compressed actuator magnitude; robust to huge
        ↪ scales)
    t_effort_log: float = 3.3
    act_log: float = float(np.log1p(act_l2))
    effort_penalty: float = float(1.0 - np.exp(-((act_log ** 2) / (t_effort_log
        ↪ ** 2))))  # [0,1)

    # (J) Impact penalty (log-compressed contacts; robust)
    t_impact_log: float = 3.3
    cfrc_log: float = float(np.log1p(cfrc_rms))
    impact_penalty: float = float(1.0 - np.exp(-((cfrc_log ** 2) / (t_impact_log
        ↪ ** 2))))  # [0,1)

    # -------------------------
```

```
    # Combine (bounded components => stable weighting)
    # -------------------------
    w_speed: float = 1.55
    w_height: float = 0.40

    w_speed_cap: float = 0.55
    w_unhealthy: float = 1.35

    w_drift: float = 0.28
    w_angvel: float = 0.22
    w_sym: float = 0.14
    w_posture: float = 0.12
    w_effort: float = 0.26
    w_impact: float = 0.24

    reward: float = float(
        w_speed * speed_reward
        + w_height * height_reward
        - w_speed_cap * speed_cap_penalty
        - w_unhealthy * unhealthy_penalty
        - w_drift * drift_penalty
        - w_angvel * angvel_penalty
        - w_sym * symmetry_penalty
        - w_posture * posture_penalty
        - w_effort * effort_penalty
        - w_impact * impact_penalty
    )

    components: Dict[str, float] = {
        "speed_reward": float(speed_reward),
        "height_reward": float(height_reward),
        "speed_cap_penalty": float(speed_cap_penalty),
        "unhealthy_penalty": float(unhealthy_penalty),
        "drift_penalty": float(drift_penalty),
        "angvel_penalty": float(angvel_penalty),
        "symmetry_penalty": float(symmetry_penalty),
        "posture_penalty": float(posture_penalty),
        "effort_penalty": float(effort_penalty),
        "impact_penalty": float(impact_penalty),
        # diagnostics
        "vx": float(vx),
        "vy": float(vy),
        "vz": float(vz),
        "torso_z": float(torso_z),
        "act_l2": float(act_l2),
        "act_log": float(act_log),
        "cfrc_rms": float(cfrc_rms),
        "cfrc_log": float(cfrc_log),
        "total_reward": float(reward),
    }

    return float(reward), components
```

Generation 5 (r_5)

```
def compute_reward(observation: np.ndarray) -> Tuple[float, Dict[str, float]]:
    obs = observation

    # -------------------------
    # Key signals from obs (Humanoid-v4 layout given)
    # -------------------------
    torso_z = float(obs[0])

    # torso orientation as quaternion (x,y,z,w) in the provided table ordering
    tqx = float(obs[1])
```

```
    tqy = float(obs[2])
    tqz = float(obs[3])
    tqw = float(obs[4])

    vx = float(obs[22])
    vy = float(obs[23])
    vz = float(obs[24])

    wx = float(obs[25])
    wy = float(obs[26])
    wz = float(obs[27])

    abdomen_z = float(obs[5])
    abdomen_y = float(obs[6])
    abdomen_x = float(obs[7])

    r_hip_x, r_hip_z, r_hip_y, r_knee = float(obs[8]), float(obs[9]),
        ↪ float(obs[10]), float(obs[11])
    l_hip_x, l_hip_z, l_hip_y, l_knee = float(obs[12]), float(obs[13]),
        ↪ float(obs[14]), float(obs[15])

    r_sh1, r_sh2, r_elb = float(obs[16]), float(obs[17]), float(obs[18])
    l_sh1, l_sh2, l_elb = float(obs[19]), float(obs[20]), float(obs[21])

    act = obs[269:292]
    act_l2 = float(np.sqrt(np.mean(np.square(act)) + 1e-8))

    cfrc = obs[292:376]
    cfrc_rms = float(np.sqrt(np.mean(np.square(cfrc)) + 1e-8))

    # -------------------------
    # Reward components (bounded where possible)
    # -------------------------

    # 1) Forward speed (primary)
    t_speed = 2.5
    speed_reward = 1.0 - float(np.exp(-max(0.0, vx) / t_speed))  # [0,1)

    # 1b) Backward motion penalty (from reward #1; mild but useful)
    t_back = 1.0
    backward_penalty = 1.0 - float(np.exp(-(max(0.0, -vx) ** 2) / (t_back **
        ↪ 2)))  # [0,1)

    # 2) Height / health gate around healthy band [1,2] (from reward #2)
    healthy_min_z, healthy_max_z = 1.0, 2.0
    z_mid = 0.5 * (healthy_min_z + healthy_max_z)
    half_range = 0.5 * (healthy_max_z - healthy_min_z)

    t_height_gate = 1.5
    z_out = max(0.0, abs(torso_z - z_mid) - half_range)  # 0 inside band
    height_reward = float(np.exp(-(z_out ** 2) / (t_height_gate ** 2)))  # [0,1]

    # 3) Penalize lateral/vertical drift velocities
    t_drift = 1.0
    drift_penalty = 1.0 - float(np.exp(-(vy * vy + vz * vz) / (t_drift ** 2)))  #
        ↪ [0,1)

    # 4) Penalize excessive torso angular velocity
    t_angvel = 4.0
    angvel_mag2 = wx * wx + wy * wy + wz * wz
    angvel_penalty = 1.0 - float(np.exp(-angvel_mag2 / (t_angvel ** 2)))  # [0,1)

    # 5) Symmetry (human-like gait)
    t_sym = 0.8
    leg_diff2 = (
```

```
        (r_hip_x + l_hip_x) ** 2 +
        (r_hip_y + l_hip_y) ** 2 +
        (r_hip_z + l_hip_z) ** 2 +
        (r_knee - l_knee) ** 2
    )
    arm_diff2 = (
        (r_sh1 + l_sh1) ** 2 +
        (r_sh2 + l_sh2) ** 2 +
        (r_elb - l_elb) ** 2
    )
    symmetry_penalty = 1.0 - float(np.exp(-(leg_diff2 + 0.5 * arm_diff2) /
        ↪  (t_sym ** 2)))  # [0,1)

    # 6) Posture (avoid extreme abdomen bending/twisting)
    t_posture = 0.7
    posture_mag2 = abdomen_x ** 2 + abdomen_y ** 2 + 0.5 * abdomen_z ** 2
    posture_penalty = 1.0 - float(np.exp(-posture_mag2 / (t_posture ** 2)))  #
        ↪  [0,1)

    # 7) Effort penalty (log-compressed actuator norm)
    t_effort_log = 3.0
    act_log = float(np.log1p(act_l2))
    effort_penalty = 1.0 - float(np.exp(-((act_log ** 2) / (t_effort_log **
        ↪  2))))  # [0,1)

    # 8) Impact penalty (soft contacts)
    t_impact = 15.0
    impact_penalty = 1.0 - float(np.exp(-(cfrc_rms ** 2) / (t_impact ** 2)))  #
        ↪  [0,1)

    # 9) Upright / anti-flip penalty using torso orientation quaternion.
    # If upright corresponds to (0,0,0,1), penalize deviation from that.
    t_upright = 0.6
    quat_dev2 = tqx * tqx + tqy * tqy + tqz * tqz + (tqw - 1.0) * (tqw - 1.0)
    upright_penalty = 1.0 - float(np.exp(-quat_dev2 / (t_upright ** 2)))  # [0,1)

    # -------------------------
    # Combine (keep speed dominant; add mild backward + upright shaping)
    # -------------------------
    w_speed = 1.35
    w_height = 0.45

    w_drift = 0.30
    w_angvel = 0.25
    w_sym = 0.20
    w_posture = 0.15
    w_effort = 0.30
    w_impact = 0.25
    w_back = 0.15
    w_upright = 0.15

    reward = (
        w_speed * speed_reward
        + w_height * height_reward
        - w_drift * drift_penalty
        - w_angvel * angvel_penalty
        - w_sym * symmetry_penalty
        - w_posture * posture_penalty
        - w_effort * effort_penalty
        - w_impact * impact_penalty
        - w_back * backward_penalty
        - w_upright * upright_penalty
    )

    components: Dict[str, float] = {
```

```
        "speed_reward": float(speed_reward),
        "height_reward": float(height_reward),
        "drift_penalty": float(drift_penalty),
        "angvel_penalty": float(angvel_penalty),
        "symmetry_penalty": float(symmetry_penalty),
        "posture_penalty": float(posture_penalty),
        "effort_penalty": float(effort_penalty),
        "impact_penalty": float(impact_penalty),
        "backward_penalty": float(backward_penalty),
        "upright_penalty": float(upright_penalty),
        # diagnostics
        "vx": float(vx),
        "torso_z": float(torso_z),
        "act_l2": float(act_l2),
        "act_log": float(act_log),
        "cfrc_rms": float(cfrc_rms),
        "total_reward": float(reward),
    }

    return float(reward), components
```

K.2 Humanoid Standup

We show generations 0–5, covering the rising phase up to the peak; generation 6 onward begins the post-peak plasticity-loss decline characterized in Appendix I and is thus omitted here.

Reading the shaping trajectory. The Humanoid Standup trajectory shows failed mutations being absorbed rather than compounding: fitness oscillates through two regressions before a sharp single-step breakthrough at r_5. Each dip coincides with an over-aggressive design move; the breakthrough recombines elements that survived earlier generations rather than refining the immediate parent. Four design axes are active:

- **Orientation formulation try-and-revert.** r_0 uses `abs(qw)` (sign-invariant); r_3 switches to raw `qw` to penalize inversion; r_4 tries the full-angle $2\arccos(|q_w|)$; r_5 reverts to `qw` with the comment that it "appears more learnable than the full-angle version in practice."
- **Standing-hold bonus refinement.** r_1 introduces a multiplicative bonus (*tall*×*upright*×*stable*); r_4 reformulates it as $1-\exp$ saturation; r_5 rewraps it as $\exp(-(1-\text{core})/\tau)$ for bounded output.
- **Phase-gate architecture.** r_0's scalar phase weights become r_1's explicit smooth gates (`early_gate`, `upright_gate`, `standing_gate`), carried through r_2, then collapsed back into phase weights once the gates have shaped the components.
- **Temperature oscillation.** $r_1 \to r_2$ sharpens (`angvel_temp`: $2.0 \to 1.2$, `upright_temp`: $0.20 \to 0.18$) and fitness drops to 0.52; r_5 settles on moderate values (`upright_temp`=0.20, `stability_temp`=1.5).

The r_5 breakthrough reuses design elements that first appeared in non-adjacent predecessors rather than refining its immediate parent r_4.

Generation 0 (r_0)

```
def compute_reward(observation: np.ndarray) -> Tuple[float, Dict[str, float]]:
    """
    Reward for humanoid stand-up:
    - Always reward torso height to provide continuous progress signal.
    - When low to the ground, emphasize upward vertical velocity.
    - As height increases, shift emphasis toward upright torso orientation.
    - Once standing, emphasize stability (low torso angular velocity).
    - Do not reward forward motion.

    Observation usage:
    obs[0]      : torso height (z)
```

```
    obs[1:5]    : torso quaternion [w, x, y, z]
    obs[24]     : torso vertical velocity
    obs[25:28]  : torso angular velocity
    """

    obs = observation

    torso_height: float = float(obs[0])
    torso_quat: np.ndarray = obs[1:5]
    vertical_velocity: float = float(obs[24])
    torso_ang_vel: np.ndarray = obs[25:28]

    # -------- targets / temperatures --------
    stand_height_target = 1.4
    low_height_threshold = 0.5
    high_height_threshold = 0.8
    standing_threshold = 1.0

    height_temp = 0.35
    vel_temp = 0.75
    upright_temp = 0.25
    stability_temp = 1.5

    # -------- basic shaping terms --------
    # Continuous progress / alive bonus: 0 on ground-ish, approaches 1 near
        ↪ standing height.
    height_error = max(0.0, stand_height_target - torso_height)
    height_reward = float(np.exp(-height_error / height_temp))

    # Upward velocity only; avoid rewarding downward motion.
    upward_velocity = max(0.0, vertical_velocity)
    upward_vel_reward = float(1.0 - np.exp(-upward_velocity / vel_temp))

    # Upright orientation: torso quaternion close to [1,0,0,0].
    # Since q and -q represent same orientation, compare abs(w).
    qw = float(torso_quat[0])
    uprightness = np.clip(abs(qw), 0.0, 1.0)
    upright_error = 1.0 - uprightness
    upright_reward = float(np.exp(-upright_error / upright_temp))

    # Stability: low torso angular velocity is best, especially once standing.
    ang_speed = float(np.linalg.norm(torso_ang_vel))
    stability_reward = float(np.exp(-ang_speed / stability_temp))

    # -------- phase-aware weighting --------
    # progress_phase: 0 when low, 1 when high
    progress_phase = np.clip(
        (torso_height - low_height_threshold) / (high_height_threshold -
            ↪ low_height_threshold),
        0.0,
        1.0,
    )

    # stand_phase: 0 below standing threshold, increases above it
    stand_phase = np.clip((torso_height - standing_threshold) / 0.25, 0.0, 1.0)

    # Low height: prioritize getting up.
    # High height: prioritize uprightness.
    # Standing: stability is most important.
    w_height = 0.35
    w_upward = 0.40 * (1.0 - progress_phase)
    w_upright = 0.15 + 0.35 * progress_phase
    w_stability = 0.10 + 0.30 * stand_phase

    reward = (
```

```
        w_height * height_reward
        + w_upward * upward_vel_reward
        + w_upright * upright_reward
        + w_stability * stability_reward
    )

    # Small penalty for excessive horizontal drift to discourage
        ↪ locomotion/crawling,
    # without making it a primary objective.
    horizontal_speed = float(np.linalg.norm(obs[22:24]))
    drift_penalty = 0.05 * (1.0 - np.exp(-horizontal_speed / 1.0))
    reward -= drift_penalty

    # Keep reward non-negative and bounded.
    reward = float(np.clip(reward, 0.0, 1.0))

    components = {
        "height_reward": height_reward,
        "upward_vel_reward": upward_vel_reward,
        "upright_reward": upright_reward,
        "stability_reward": stability_reward,
        "drift_penalty": drift_penalty,
        "torso_height": torso_height,
        "vertical_velocity": vertical_velocity,
        "uprightness": uprightness,
        "angular_speed": ang_speed,
        "phase_progress": float(progress_phase),
        "phase_standing": float(stand_phase),
        "total_reward": reward,
    }

    return reward, components
```

Generation 1 (r_1)

```
def compute_reward(observation: np.ndarray) -> Tuple[float, Dict[str, float]]:
    """
    Hybrid standup reward for HumanoidStandupEnv.

    Design summary:
    - Keep a strong continuous height/progress reward at all times.
    - Reward positive upward velocity mainly while the humanoid is still low.
    - Gradually turn on upright-orientation reward as height increases.
    - Once near standing, prioritize stability strongly:
        low torso angular velocity + low vertical motion.
    - Apply only a small horizontal-motion penalty, mainly after the agent is
        ↪ already high,
      so we discourage crawling/sliding without suppressing useful transitional
          ↪ movement.

    Uses only documented observation entries:
      obs[0]    = torso height z
      obs[1:5]  = torso quaternion [w, x, y, z]
      obs[22:24]= torso horizontal velocity x,y
      obs[24]   = torso vertical velocity z
      obs[25:28]= torso angular velocity x,y,z
    """
    obs = observation

    # Key state
    height: float = float(obs[0])
    quat: np.ndarray = np.asarray(obs[1:5], dtype=np.float64)
    vx: float = float(obs[22])
    vy: float = float(obs[23])
    vz: float = float(obs[24])
```

```
    ang_vel: np.ndarray = np.asarray(obs[25:28], dtype=np.float64)

    # Reference heights
    lying_height: float = 0.2
    target_height: float = 1.4
    upright_start_height: float = 0.55
    standing_height: float = 1.0

    # Temperatures
    height_temp: float = 0.40
    rise_vel_temp: float = 0.90
    upright_temp: float = 0.20
    angvel_temp: float = 2.00
    standstill_vz_temp: float = 0.35
    horiz_temp: float = 2.50

    # -------------------------
    # Smooth phase gates
    # -------------------------
    height_progress: float = float(
        np.clip((height - lying_height) / (target_height - lying_height), 0.0,
            ↪ 1.0)
    )

    # Early rising gate: high when low, fades by ~0.85m
    early_gate: float = float(1.0 - np.clip((height - 0.45) / 0.40, 0.0, 1.0))

    # Upright gate: starts turning on after agent has made meaningful upward
        ↪ progress
    upright_gate: float = float(np.clip((height - upright_start_height) / 0.35,
        ↪ 0.0, 1.0))

    # Standing/stability gate: strongly active once close to standing
    standing_gate: float = float(np.clip((height - standing_height) / 0.20, 0.0,
        ↪ 1.0))

    # -------------------------
    # Reward components
    # -------------------------
    # 1) Height reward: main continuous progress signal
    # Monotonic toward standing height and saturates near the target.
    height_error: float = float(max(target_height - height, 0.0))
    height_reward: float = float(np.exp(-height_error / height_temp))

    # 2) Upward velocity reward: only positive z-velocity, emphasized early
    upward_speed: float = float(max(vz, 0.0))
    upward_velocity_raw: float = float(1.0 - np.exp(-upward_speed /
        ↪ rise_vel_temp))
    upward_velocity_reward: float = float(early_gate * upward_velocity_raw)

    # 3) Upright reward:
    # Better than raw quat distance here to use abs(qw), since q and -q are
        ↪ equivalent.
    quat_norm: float = float(np.linalg.norm(quat) + 1e-8)
    qw: float = float(quat[0] / quat_norm)
    upright_error: float = float(1.0 - abs(qw))
    upright_raw: float = float(np.exp(-upright_error / upright_temp))
    upright_reward: float = float(upright_gate * upright_raw)

    # 4) Stability reward from low torso angular velocity, especially important
        ↪ when standing
    ang_speed: float = float(np.linalg.norm(ang_vel))
    stability_raw: float = float(np.exp(-ang_speed / angvel_temp))
    stability_reward: float = float(standing_gate * stability_raw)
```

```
    # 5) Standstill vertical reward: once high, prefer keeping vz small over
        ↪ more rising speed
    standstill_vertical_raw: float = float(np.exp(-abs(vz) / standstill_vz_temp))
    standstill_vertical_reward: float = float(standing_gate *
        ↪ standstill_vertical_raw)

    # 6) Small horizontal-motion penalty, mostly after standing/high posture is
        ↪ reached
    horiz_speed: float = float(np.sqrt(vx * vx + vy * vy))
    horizontal_penalty: float = float(upright_gate * (1.0 - np.exp(-horiz_speed /
        ↪ horiz_temp)))

    # 7) Bonus for actually good standing posture: tall + upright + stable + low
        ↪ vertical motion
    standing_quality_bonus: float = float(
        standing_gate * upright_raw * stability_raw * standstill_vertical_raw
    )

    # -------------------------
    # Weighted total
    # -------------------------
    reward: float = float(
        0.40 * height_reward +
        0.22 * upward_velocity_reward +
        0.18 * upright_reward +
        0.28 * stability_reward +
        0.18 * standstill_vertical_reward +
        0.16 * standing_quality_bonus -
        0.04 * horizontal_penalty
    )

    reward = float(np.clip(reward, 0.0, 1.0))

    info: Dict[str, float] = {
        "reward_total": reward,
        "height_reward": height_reward,
        "upward_velocity_reward": upward_velocity_reward,
        "upright_reward": upright_reward,
        "stability_reward": stability_reward,
        "standstill_vertical_reward": standstill_vertical_reward,
        "standing_quality_bonus": standing_quality_bonus,
        "horizontal_penalty": horizontal_penalty,
        "height": height,
        "height_progress": height_progress,
        "vertical_velocity": vz,
        "angular_speed": ang_speed,
        "horizontal_speed": horiz_speed,
        "early_gate": early_gate,
        "upright_gate": upright_gate,
        "standing_gate": standing_gate,
        "upright_raw": upright_raw,
        "stability_raw": stability_raw,
        "standstill_vertical_raw": standstill_vertical_raw,
    }
    return reward, info
```

Generation 2 (r_2)

```
def compute_reward(observation: np.ndarray) -> Tuple[float, Dict[str, float]]:
    """
    Crossover reward for HumanoidStandupEnv.

    Rationale for combining components:
```

```
- Keep the strong continuous height reward from both candidates since it gives
  dense progress signal from lying to standing.
- Keep early-phase positive vertical-velocity reward, but only while low, so the
  agent is encouraged to rise without being pushed to keep bouncing once upright.
- Use the better uprightness formulation based on abs(qw), because q and -q are
  equivalent quaternions. This avoids an unnecessary representation artifact.
- Make stability dominant only after the humanoid is meaningfully high / near standing.
  The prior rewards' stability terms were often effectively inactive, so we use a
  smoother gating and a less brittle exponential on angular speed.
- Add a standstill term once standing: low vertical speed is desirable after getting up.
- Keep only a small horizontal-motion penalty and gate it late, so it discourages
  crawling/sliding without suppressing transitional movement.
- Add a multiplicative standing bonus for truly good states: tall + upright + stable + still.

Uses only documented observation entries:
  obs[0]    torso height
  obs[1:5]  torso quaternion [w, x, y, z]
  obs[22:24] torso horizontal velocity
  obs[24]   torso vertical velocity
  obs[25:28] torso angular velocity
"""
obs = observation

# Key state
height: float = float(obs[0])
quat: np.ndarray = np.asarray(obs[1:5], dtype=np.float64)
vx: float = float(obs[22])
vy: float = float(obs[23])
vz: float = float(obs[24])
ang_vel: np.ndarray = np.asarray(obs[25:28], dtype=np.float64)

# Reference heights
lying_height: float = 0.20
target_height: float = 1.40
low_height: float = 0.50
upright_start_height: float = 0.75
standing_height: float = 1.00

# Temperatures
height_temp: float = 0.35
rise_vel_temp: float = 0.80
upright_temp: float = 0.18
angvel_temp: float = 1.20
standstill_vz_temp: float = 0.30
horiz_temp: float = 1.50

# Basic derived quantities
horiz_speed: float = float(np.sqrt(vx * vx + vy * vy))
ang_speed: float = float(np.linalg.norm(ang_vel))
quat_norm: float = float(np.linalg.norm(quat) + 1e-8)
qw: float = float(quat[0] / quat_norm)

# -------------------------
# Smooth phase gates
# -------------------------
height_progress: float = float(
```

```
        np.clip((height - lying_height) / (target_height - lying_height), 0.0,
            ↪ 1.0)
    )

    # Early gate: emphasize rising when still low
    early_gate: float = float(
        1.0 - np.clip((height - low_height) / 0.35, 0.0, 1.0)
    )

    # Upright gate: turn on orientation guidance only after meaningful elevation
    upright_gate: float = float(
        np.clip((height - upright_start_height) / 0.25, 0.0, 1.0)
    )

    # Standing gate: stability should matter most once near/above standing
    standing_gate: float = float(
        np.clip((height - standing_height) / 0.20, 0.0, 1.0)
    )

    # -------------------------
    # Reward components
    # -------------------------
    # 1) Dense height/progress reward
    height_error: float = float(max(target_height - height, 0.0))
    height_reward: float = float(np.exp(-height_error / height_temp))

    # 2) Early upward velocity reward
    upward_speed: float = float(max(vz, 0.0))
    upward_velocity_raw: float = float(1.0 - np.exp(-upward_speed /
        ↪ rise_vel_temp))
    upward_velocity_reward: float = float(early_gate * upward_velocity_raw)

    # 3) Uprightness using abs(qw) for quaternion sign invariance
    upright_error: float = float(1.0 - abs(qw))
    upright_raw: float = float(np.exp(-upright_error / upright_temp))
    upright_reward: float = float(upright_gate * upright_raw)

    # 4) Stability from low torso angular velocity
    # Use squared speed inside exp for stronger preference near stillness,
    # but only after standing gate activates.
    stability_raw: float = float(np.exp(-(ang_speed ** 2) / angvel_temp))
    stability_reward: float = float(standing_gate * stability_raw)

    # 5) Standstill vertical reward once standing: prefer low |vz|
    standstill_vertical_raw: float = float(np.exp(-abs(vz) / standstill_vz_temp))
    standstill_vertical_reward: float = float(standing_gate *
        ↪ standstill_vertical_raw)

    # 6) Weak late horizontal penalty to discourage crawling/sliding
    horizontal_penalty: float = float(
        standing_gate * (1.0 - np.exp(-(horiz_speed ** 2) / horiz_temp))
    )

    # 7) High-quality standing bonus
    standing_quality_bonus: float = float(
        standing_gate * height_progress * upright_raw * stability_raw *
            ↪ standstill_vertical_raw
    )

    # -------------------------
    # Total reward
    # -------------------------
    reward: float = float(
        0.42 * height_reward +
        0.24 * upward_velocity_reward +
```

```
        0.20 * upright_reward +
        0.30 * stability_reward +
        0.18 * standstill_vertical_reward +
        0.18 * standing_quality_bonus -
        0.04 * horizontal_penalty
    )

    reward = float(np.clip(reward, 0.0, 1.0))

    info: Dict[str, float] = {
        "reward_total": reward,
        "height_reward": height_reward,
        "upward_velocity_reward": upward_velocity_reward,
        "upright_reward": upright_reward,
        "stability_reward": stability_reward,
        "standstill_vertical_reward": standstill_vertical_reward,
        "standing_quality_bonus": standing_quality_bonus,
        "horizontal_penalty": horizontal_penalty,
        "height": height,
        "height_progress": height_progress,
        "vertical_velocity": vz,
        "angular_speed": ang_speed,
        "horizontal_speed": horiz_speed,
        "early_gate": early_gate,
        "upright_gate": upright_gate,
        "standing_gate": standing_gate,
        "upright_raw": upright_raw,
        "stability_raw": stability_raw,
        "standstill_vertical_raw": standstill_vertical_raw,
    }
    return reward, info
```

Generation 3 (r_3)

```
def compute_reward(observation: np.ndarray) -> Tuple[float, Dict[str, float]]:
    """
    Reward for humanoid stand-up:
    - Always reward torso height to provide continuous progress signal.
    - When low to the ground, emphasize upward vertical velocity.
    - As height increases, shift emphasis toward upright torso orientation.
    - Once standing, emphasize stability (low torso angular velocity).
    - Do not reward forward motion.

    Observation usage:
    obs[0]      : torso height (z)
    obs[1:5]    : torso quaternion [w, x, y, z]
    obs[24]     : torso vertical velocity
    obs[25:28]  : torso angular velocity
    """

    obs = observation

    torso_height: float = float(obs[0])
    torso_quat: np.ndarray = obs[1:5]
    vertical_velocity: float = float(obs[24])
    torso_ang_vel: np.ndarray = obs[25:28]

    # -------- targets / temperatures --------
    stand_height_target = 1.4
    low_height_threshold = 0.5
    high_height_threshold = 0.8
    standing_threshold = 1.0

    height_temp = 0.35
    vel_temp = 0.75
```

```
    upright_temp = 0.20
    stability_temp = 1.5

    # -------- basic shaping terms --------
    # Continuous progress / alive bonus: 0 on ground-ish, approaches 1 near
        ↪ standing height.
    height_error = max(0.0, stand_height_target - torso_height)
    height_reward = float(np.exp(-height_error / height_temp))

    # Upward velocity only; avoid rewarding downward motion.
    upward_velocity = max(0.0, vertical_velocity)
    upward_vel_reward = float(1.0 - np.exp(-upward_velocity / vel_temp))

    # -------- mutated upright component --------
    # Upright torso orientation targets quaternion [1, 0, 0, 0].
    # Using qw directly (not abs(qw)) better distinguishes true upright from
        ↪ other orientations.
    qw = float(torso_quat[0])
    upright_alignment = float(np.clip(qw, -1.0, 1.0))
    upright_error = 1.0 - upright_alignment
    upright_reward = float(np.exp(-upright_error / upright_temp))

    # Stability: low torso angular velocity is best, especially once standing.
    ang_speed = float(np.linalg.norm(torso_ang_vel))
    stability_reward = float(np.exp(-ang_speed / stability_temp))

    # -------- phase-aware weighting --------
    progress_phase = np.clip(
        (torso_height - low_height_threshold) / (high_height_threshold -
            ↪ low_height_threshold),
        0.0,
        1.0,
    )

    stand_phase = np.clip((torso_height - standing_threshold) / 0.25, 0.0, 1.0)

    w_height = 0.35
    w_upward = 0.40 * (1.0 - progress_phase)
    w_upright = 0.15 + 0.35 * progress_phase
    w_stability = 0.10 + 0.30 * stand_phase

    reward = (
        w_height * height_reward
        + w_upward * upward_vel_reward
        + w_upright * upright_reward
        + w_stability * stability_reward
    )

    # Small penalty for excessive horizontal drift to discourage
        ↪ locomotion/crawling.
    horizontal_speed = float(np.linalg.norm(obs[22:24]))
    drift_penalty = 0.05 * (1.0 - np.exp(-horizontal_speed / 1.0))
    reward -= drift_penalty

    reward = float(np.clip(reward, 0.0, 1.0))

    components = {
        "height_reward": height_reward,
        "upward_vel_reward": upward_vel_reward,
        "upright_reward": upright_reward,
        "stability_reward": stability_reward,
        "drift_penalty": drift_penalty,
        "torso_height": torso_height,
        "vertical_velocity": vertical_velocity,
        "upright_alignment": upright_alignment,
```

```
        "angular_speed": ang_speed,
        "phase_progress": float(progress_phase),
        "phase_standing": float(stand_phase),
        "total_reward": reward,
    }

    return reward, components
```

Generation 4 (r_4)

```
def compute_reward(observation: np.ndarray) -> Tuple[float, Dict[str, float]]:
    """
    Crossover reward for humanoid stand-up.

    Design rationale:
    - Keep the strong continuous height shaping from both candidates.
    - Keep the useful upward-velocity reward early in the motion.
    - Use the better-performing full-quaternion upright term from the
        ↪ higher-fitness reward
      (it distinguishes lying-on-back from truly upright better than abs(qw)
          ↪ alone).
    - Improve stability shaping so it is weak during the get-up transition but
        ↪ becomes dominant
      once the agent is near/at standing height.
    - Add a standing hold bonus that explicitly rewards being simultaneously
        ↪ tall, upright,
      and rotationally stable, which should better exploit the conditional fall
          ↪ termination.
    - Keep only a small horizontal drift penalty to discourage crawling/running
        ↪ without
      turning the task into anti-locomotion optimization.

    Observation usage:
    obs[0]      : torso height (z)
    obs[1:5]    : torso quaternion [w, x, y, z]
    obs[22:24]  : torso horizontal velocity (x, y)
    obs[24]     : torso vertical velocity
    obs[25:28]  : torso angular velocity
    """
    obs = observation

    torso_height: float = float(obs[0])
    torso_quat: np.ndarray = obs[1:5]
    horizontal_vel: np.ndarray = obs[22:24]
    vertical_velocity: float = float(obs[24])
    torso_ang_vel: np.ndarray = obs[25:28]

    # -------- targets / thresholds --------
    stand_height_target: float = 1.4
    low_height_threshold: float = 0.5
    high_height_threshold: float = 0.8
    standing_threshold: float = 1.0

    # -------- temperatures --------
    height_temp: float = 0.35
    vel_temp: float = 0.75
    upright_angle_temp: float = 0.45
    stability_temp: float = 2.5
    hold_temp: float = 0.20
    drift_temp: float = 1.0

    # -------- height / progress --------
    # Strong dense shaping for upward progress; 1 near standing height.
    height_error: float = max(0.0, stand_height_target - torso_height)
```

```
    height_reward: float = float(np.exp(-height_error / height_temp))

    # Extra linearized progress signal from initial lying height (~0.2) to
        ↪ standing (~1.4).
    progress_reward: float = float(np.clip((torso_height - 0.2) /
        ↪ (stand_height_target - 0.2), 0.0, 1.0))

    # -------- upward motion --------
    upward_velocity: float = max(0.0, vertical_velocity)
    upward_vel_reward: float = float(1.0 - np.exp(-upward_velocity / vel_temp))

    # -------- upright orientation --------
    # Full quaternion-based upright angle from target [1,0,0,0].
    quat_norm: float = float(np.linalg.norm(torso_quat))
    if quat_norm > 1e-8:
        q = torso_quat / quat_norm
    else:
        q = np.array([1.0, 0.0, 0.0, 0.0], dtype=np.float64)

    qw: float = float(np.clip(abs(q[0]), 0.0, 1.0))
    upright_angle: float = float(2.0 * np.arccos(qw))
    upright_reward: float = float(np.exp(-upright_angle / upright_angle_temp))

    # -------- stability --------
    ang_speed: float = float(np.linalg.norm(torso_ang_vel))
    stability_reward: float = float(np.exp(-ang_speed / stability_temp))

    # -------- phase scheduling --------
    # Transition from "get off the ground" to "be upright" to "hold stable
        ↪ stand".
    progress_phase: float = float(np.clip(
        (torso_height - low_height_threshold) / (high_height_threshold -
            ↪ low_height_threshold),
        0.0, 1.0
    ))
    stand_phase: float = float(np.clip(
        (torso_height - standing_threshold) / 0.25,
        0.0, 1.0
    ))

    # -------- standing hold bonus --------
    # Encourages the final desired conjunction: tall + upright + stable.
    # This is especially helpful because losing height after standing causes
        ↪ termination.
    stand_quality: float = height_reward * upright_reward * stability_reward
    standing_hold_reward: float = float(1.0 - np.exp(-stand_quality / hold_temp))

    # -------- weights --------
    # Early: prioritize progress and rising.
    # Mid: shift toward uprightness.
    # Standing: prioritize stability and standing hold over speed.
    w_height: float = 0.22
    w_progress: float = 0.18
    w_upward: float = 0.32 * (1.0 - progress_phase)
    w_upright: float = 0.10 + 0.25 * progress_phase
    w_stability: float = 0.05 + 0.28 * stand_phase
    w_hold: float = 0.35 * stand_phase

    reward: float = (
        w_height * height_reward
        + w_progress * progress_reward
        + w_upward * upward_vel_reward
        + w_upright * upright_reward
        + w_stability * stability_reward
        + w_hold * standing_hold_reward
```

```
    )

    # Small drift penalty to reduce crawling/running incentives.
    horizontal_speed: float = float(np.linalg.norm(horizontal_vel))
    drift_penalty: float = float(0.03 * (1.0 - np.exp(-horizontal_speed /
        ↪ drift_temp)))
    reward -= drift_penalty

    # Bound reward to stable range.
    reward = float(np.clip(reward, 0.0, 1.0))

    components: Dict[str, float] = {
        "height_reward": height_reward,
        "progress_reward": progress_reward,
        "upward_vel_reward": upward_vel_reward,
        "upright_reward": upright_reward,
        "stability_reward": stability_reward,
        "standing_hold_reward": standing_hold_reward,
        "drift_penalty": drift_penalty,
        "torso_height": torso_height,
        "vertical_velocity": vertical_velocity,
        "upright_angle": upright_angle,
        "angular_speed": ang_speed,
        "horizontal_speed": horizontal_speed,
        "phase_progress": progress_phase,
        "phase_standing": stand_phase,
        "total_reward": reward,
    }

    return reward, components
```

Generation 5 (r_5)

```
def compute_reward(observation: np.ndarray) -> Tuple[float, Dict[str, float]]:
    """
    Crossover reward for humanoid stand-up.

    Design rationale:
    - Keep the strong continuous height shaping from the better-performing
        ↪ reward.
    - Keep the simple quaternion-w based upright term from the better-performing
        ↪ reward,
      since it appears more learnable than the full-angle version in practice.
    - Preserve phase-aware weighting:
        * low height -> reward upward velocity most
        * mid height -> shift toward uprightness
        * standing -> prioritize stability heavily
    - Add a standing hold bonus gated by height and uprightness so the agent is
        ↪ rewarded
      for remaining in a stable standing posture, not just reaching it briefly.
    - Keep horizontal drift penalty small so locomotion/crawling is discouraged
        ↪ but not dominant.
    """

    obs = observation

    torso_height: float = float(obs[0])
    torso_quat: np.ndarray = obs[1:5]
    vertical_velocity: float = float(obs[24])
    torso_ang_vel: np.ndarray = obs[25:28]
    horizontal_vel: np.ndarray = obs[22:24]

    # ---------------- temperatures / targets ----------------
    stand_height_target: float = 1.4
    low_height_threshold: float = 0.5
```

```
    high_height_threshold: float = 0.8
    standing_threshold: float = 1.0

    height_temp: float = 0.35
    vel_temp: float = 0.75
    upright_temp: float = 0.20
    stability_temp: float = 1.5
    stand_height_temp: float = 0.18
    hold_temp: float = 0.25
    drift_temp: float = 1.0

    # ---------------- core signals ----------------
    # Continuous upward progress signal
    height_error: float = max(0.0, stand_height_target - torso_height)
    height_reward: float = float(np.exp(-height_error / height_temp))

    # Early phase: reward only positive vertical motion
    upward_velocity: float = max(0.0, vertical_velocity)
    upward_vel_reward: float = float(1.0 - np.exp(-upward_velocity / vel_temp))

    # Uprightness: use normalized quaternion w-component toward [1,0,0,0]
    quat_norm: float = float(np.linalg.norm(torso_quat))
    if quat_norm > 1e-8:
        q = torso_quat / quat_norm
    else:
        q = np.array([1.0, 0.0, 0.0, 0.0], dtype=np.float64)

    qw: float = float(np.clip(q[0], -1.0, 1.0))
    upright_alignment: float = max(0.0, qw)
    upright_error: float = 1.0 - upright_alignment
    upright_reward: float = float(np.exp(-upright_error / upright_temp))

    # Stability: low torso angular velocity
    ang_speed: float = float(np.linalg.norm(torso_ang_vel))
    stability_reward: float = float(np.exp(-ang_speed / stability_temp))

    # Extra standing-height sharpness near final goal
    standing_height_error: float = max(0.0, stand_height_target - torso_height)
    standing_height_reward: float = float(np.exp(-standing_height_error /
        ↪ stand_height_temp))

    # ---------------- phase estimation ----------------
    progress_phase: float = float(np.clip(
        (torso_height - low_height_threshold) / (high_height_threshold -
            ↪ low_height_threshold),
        0.0,
        1.0,
    ))
    stand_phase: float = float(np.clip(
        (torso_height - standing_threshold) / 0.25,
        0.0,
        1.0,
    ))

    # ---------------- standing hold bonus ----------------
    # This term becomes large only when the agent is both tall, upright, and
        ↪ stable.
    # It helps exploit the environment's conditional termination by encouraging
        ↪ maintenance.
    hold_core: float = standing_height_reward * upright_reward * stability_reward
    stand_hold_reward: float = float(np.exp(-(1.0 - hold_core) / hold_temp))

    # ---------------- weighted combination ----------------
    # Better-performing structure, but with more emphasis on final stability
        ↪ after standing.
```

```
    w_height: float = 0.30
    w_upward: float = 0.40 * (1.0 - progress_phase)
    w_upright: float = 0.10 + 0.35 * progress_phase
    w_stability: float = 0.05 + 0.35 * stand_phase
    w_hold: float = 0.35 * stand_phase

    reward: float = (
        w_height * height_reward
        + w_upward * upward_vel_reward
        + w_upright * upright_reward
        + w_stability * stability_reward
        + w_hold * stand_hold_reward
    )

    # Small penalty for horizontal drift to discourage crawling/running.
    horizontal_speed: float = float(np.linalg.norm(horizontal_vel))
    drift_penalty: float = 0.04 * float(1.0 - np.exp(-horizontal_speed /
        ↪  drift_temp))
    reward -= drift_penalty

    # Keep reward bounded and non-negative
    reward = float(np.clip(reward, 0.0, 1.0))

    components: Dict[str, float] = {
        "height_reward": height_reward,
        "upward_vel_reward": upward_vel_reward,
        "upright_reward": upright_reward,
        "stability_reward": stability_reward,
        "standing_height_reward": standing_height_reward,
        "stand_hold_reward": stand_hold_reward,
        "drift_penalty": drift_penalty,
        "torso_height": torso_height,
        "vertical_velocity": vertical_velocity,
        "upright_alignment": upright_alignment,
        "angular_speed": ang_speed,
        "horizontal_speed": horizontal_speed,
        "phase_progress": progress_phase,
        "phase_standing": stand_phase,
        "total_reward": reward,
    }

    return reward, components
```

K.3 Adroit Hand

We show only generations 0–3, covering the rising phase up to convergence; later generations begin the post-peak plasticity-loss decline characterized in the extended-run analysis (Appendix I) and are omitted here.

Reading the shaping trajectory. The Adroit Hand sequence converges in four generations through small adjustments to a shared template, with no single decisive formulation change. The task is solved by r_3 at fitness 0.78; the four moves below account for the full gain:

- **Temperatures widen** (e.g., `door_temp`: $0.2 \rightarrow 0.3$) to prevent the exponential reward explosion that r_3's source comments flag explicitly.
- **Success bonus shrinks** ($100 \rightarrow 25$), balancing terminal success against intermediate progress.
- **Velocity-smoothness penalty** appears in r_2 and is kept in r_3.
- **Formulation try-and-revert.** r_1 switches the latch reward to an error-based form; r_2 reverts to progress-based but carries over the softened temperatures.

Generation 0 (r_0)

```
def compute_reward(
    hand_qpos: np.ndarray,
    latch_pos: float,
    door_pos: float,
    door_hinge: float,
    palm_pos: np.ndarray,
    handle_pos: np.ndarray,
    palm_handle_delta: np.ndarray,
    door_open: float,
    action: np.ndarray,
    joint_velocities: np.ndarray,
    joint_forces: np.ndarray
) -> Tuple[float, Dict[str, float]]:

    # Temperature parameters for reward components
    proximity_temp = 0.05
    latch_temp = 0.1
    door_temp = 0.2
    action_temp = 0.01

    # 1. Reward for getting palm close to handle
    palm_to_handle_distance = np.linalg.norm(palm_handle_delta)
    proximity_reward = np.exp(-palm_to_handle_distance / proximity_temp)

    # 2. Reward for rotating the latch (overcoming friction)
    # Target latch position is around -1.5 radians (fully rotated
        ↪ counter-clockwise)
    latch_progress = np.clip(-latch_pos / 1.5, 0, 1)
    latch_reward = np.exp(latch_progress / latch_temp)

    # 3. Reward for opening the door once latch is released
    # Door open when door_hinge >= 1.35 rad
    door_progress = np.clip(door_hinge / 1.35, 0, 1)
    door_reward = np.exp(door_progress / door_temp)

    # 4. Penalty for excessive actions (smoothness)
    action_cost = np.sum(np.square(action))
    action_penalty = -action_cost * action_temp

    # 5. Success bonus
    success_bonus = 100.0 if door_open == 1.0 else 0.0

    # Combine all rewards
    total_reward = (
        0.3 * proximity_reward +
        0.3 * latch_reward +
        0.3 * door_reward +
        0.1 * action_penalty +
        success_bonus
    )

    # Normalize total reward to [0, 1] range approximately
    normalized_reward = np.tanh(total_reward / 10.0)

    # Return detailed reward components
    reward_components = {
        'proximity_reward': proximity_reward,
        'latch_reward': latch_reward,
        'door_reward': door_reward,
        'action_penalty': action_penalty,
        'success_bonus': success_bonus
    }
```

```
    return normalized_reward, reward_components
```

Generation 1 (r_1)

```
def compute_reward(
    hand_qpos: np.ndarray,
    latch_pos: float,
    door_pos: float,
    door_hinge: float,
    palm_pos: np.ndarray,
    handle_pos: np.ndarray,
    palm_handle_delta: np.ndarray,
    door_open: float,
    action: np.ndarray,
    joint_velocities: np.ndarray,
    joint_forces: np.ndarray
) -> Tuple[float, Dict[str, float]]:

    # Temperature parameters for reward components
    proximity_temp = 0.05
    latch_temp = 1.0  # Increased from 0.1 to 1.0 to prevent extreme decay
    door_temp = 0.2
    effort_temp = 0.01

    # Reward for getting the palm close to the handle
    handle_distance = np.linalg.norm(palm_handle_delta)
    proximity_reward = np.exp(-handle_distance / proximity_temp)

    # Reward for opening the latch (latch needs to be pulled down/up)
    # Assuming the initial latch position is around 0 and target is around -1.5
        ↪ or similar negative value
    latch_target = -1.5
    latch_error = np.abs(latch_pos - latch_target)
    latch_reward = np.exp(-latch_error / latch_temp)

    # Reward for opening the door (hinge rotation)
    # Door is considered open at 1.35 radians
    door_progress = min(door_hinge / 1.35, 1.0)
    door_reward = np.exp(door_progress / door_temp) - 1.0  # Range: [0,
        ↪ e^(1/0.2) - 1] ~= [0, 147.4]

    # Penalty for excessive effort/action
    effort_penalty = np.sum(np.square(action)) * effort_temp

    # Bonus for completing the task
    completion_bonus = 0.0
    if door_open > 0:  # Door is open
        completion_bonus = 100.0

    # Total reward
    total_reward = (
        0.3 * proximity_reward +
        0.3 * latch_reward +
        0.3 * door_reward +
        completion_bonus -
        effort_penalty
    )

    # Normalize the reward to [-1, 1] range approximately
    normalized_reward = np.tanh(total_reward / 50.0)

    # Return detailed reward components
    reward_components = {
        "proximity_reward": proximity_reward,
        "latch_reward": latch_reward,
```

```
        "door_reward": door_reward,
        "effort_penalty": -effort_penalty,
        "completion_bonus": completion_bonus,
        "normalized_total_reward": normalized_reward
    }

    return normalized_reward, reward_components
```

Generation 2 (r_2)

```
def compute_reward(
    hand_qpos: np.ndarray,
    latch_pos: float,
    door_pos: float,
    door_hinge: float,
    palm_pos: np.ndarray,
    handle_pos: np.ndarray,
    palm_handle_delta: np.ndarray,
    door_open: float,
    action: np.ndarray,
    joint_velocities: np.ndarray,
    joint_forces: np.ndarray
) -> Tuple[float, Dict[str, float]]:

    # Temperature parameters for reward components
    proximity_temp = 0.03  # Lower temp for sharper focus
    latch_temp = 0.08      # Adjusted for better gradient
    door_temp = 0.15       # Balanced temperature
    action_temp = 0.005    # Reduced action penalty weight
    velocity_temp = 0.001  # Very light velocity penalty

    # 1. Reward for getting palm close to handle (improved version)
    palm_to_handle_distance = np.linalg.norm(palm_handle_delta)
    proximity_reward = np.exp(-palm_to_handle_distance / proximity_temp)

    # 2. Reward for rotating the latch (refined target)
    # Target latch position should be negative to release the door
    # Assuming initial position is around 0, target is around -1.5
    latch_target = -1.5
    latch_progress = np.clip((-latch_pos) / abs(latch_target), 0, 1)
    latch_reward = np.exp(latch_progress / latch_temp)

    # 3. Reward for opening the door
    # Door open when door_hinge >= 1.35 rad
    door_target = 1.35
    door_progress = np.clip(door_hinge / door_target, 0, 1)
    door_reward = np.exp(door_progress / door_temp)

    # 4. Penalty for excessive actions (smoothness)
    action_cost = np.sum(np.square(action))
    action_penalty = -action_cost * action_temp

    # 5. Penalty for excessive joint velocities (smoothness)
    velocity_cost = np.sum(np.square(joint_velocities))
    velocity_penalty = -velocity_cost * velocity_temp

    # 6. Success bonus
    success_bonus = 50.0 if door_open == 1.0 else 0.0

    # Combine all rewards with balanced weights
    total_reward = (
        0.25 * proximity_reward +
        0.25 * latch_reward +
        0.35 * door_reward +
        0.1 * action_penalty +
```

```
        0.05 * velocity_penalty +
        success_bonus
    )

    # Normalize total reward to [0, 1] range approximately
    normalized_reward = np.tanh(total_reward / 15.0)

    # Return detailed reward components
    reward_components = {
        'proximity_reward': proximity_reward,
        'latch_reward': latch_reward,
        'door_reward': door_reward,
        'action_penalty': action_penalty,
        'velocity_penalty': velocity_penalty,
        'success_bonus': success_bonus,
        'normalized_reward': normalized_reward
    }

    return normalized_reward, reward_components
```

Generation 3 (r_3)

```
def compute_reward(
    hand_qpos: np.ndarray,
    latch_pos: float,
    door_pos: float,
    door_hinge: float,
    palm_pos: np.ndarray,
    handle_pos: np.ndarray,
    palm_handle_delta: np.ndarray,
    door_open: float,
    action: np.ndarray,
    joint_velocities: np.ndarray,
    joint_forces: np.ndarray
) -> Tuple[float, Dict[str, float]]:

    # Temperature parameters for reward components - tuned for better scaling
    proximity_temp = 0.1  # Increased from Function 1 to prevent extreme values
    latch_temp = 0.2      # Increased to moderate the exponential growth
    door_temp = 0.3       # Increased to prevent explosion
    action_temp = 0.005   # Kept low to encourage smoothness
    velocity_temp = 0.001 # Kept low for smoothness

    # 1. Reward for getting palm close to handle - modified for better scaling
    palm_to_handle_distance = np.linalg.norm(palm_handle_delta)
    proximity_reward = np.exp(-palm_to_handle_distance / proximity_temp)

    # 2. Reward for rotating the latch - improved target and scaling
    # Target latch position should be negative to release the door
    latch_target = -1.5
    latch_progress = np.clip((-latch_pos) / abs(latch_target), 0, 1)
    latch_reward = np.exp(latch_progress / latch_temp)

    # 3. Reward for opening the door - improved scaling
    # Door open when door_hinge >= 1.35 rad
    door_target = 1.35
    door_progress = np.clip(door_hinge / door_target, 0, 1)
    door_reward = np.exp(door_progress / door_temp)

    # 4. Penalty for excessive actions (smoothness)
    action_cost = np.sum(np.square(action))
    action_penalty = -action_cost * action_temp

    # 5. Penalty for excessive joint velocities (smoothness)
    velocity_cost = np.sum(np.square(joint_velocities))
```

```
    velocity_penalty = -velocity_cost * velocity_temp

    # 6. Success bonus - balanced value
    success_bonus = 25.0 if door_open == 1.0 else 0.0

    # Combine all rewards with balanced weights to prevent any single component
        ↪ dominating
    total_reward = (
        0.3 * proximity_reward +
        0.2 * latch_reward +
        0.3 * door_reward +
        0.1 * action_penalty +
        0.05 * velocity_penalty +
        success_bonus
    )

    # Apply tanh normalization to keep rewards bounded and well-scaled
    normalized_reward = np.tanh(total_reward / 20.0)

    # Return detailed reward components
    reward_components = {
        'proximity_reward': proximity_reward,
        'latch_reward': latch_reward,
        'door_reward': door_reward,
        'action_penalty': action_penalty,
        'velocity_penalty': velocity_penalty,
        'success_bonus': success_bonus,
        'total_reward': total_reward,
        'normalized_reward': normalized_reward
    }

    return normalized_reward, reward_components
```

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The three contributions enumerated in Section 4.5 (reusable prior, expert demonstrations restoring missing capability, trajectory-level fitness) are each supported by a dedicated finding in Section 4.5 with the corresponding tables and figures (Table 2, Figure 2, Figure 3, Figure 5). Scope is limited to three MuJoCo continuous-control tasks and a single embodiment per task, as stated in the abstract and Section 6 (Limitations).

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: Section 6 contains an explicit "Limitations" paragraph listing: (i) the fixed-embodiment assumption and the requirement that pretrained ϕ already cover task-relevant states, (ii) the task-dependent rescue/harm pattern of expert injection and the inability to diagnose the binding constraint a priori, and (iii) higher run-to-run variance from compounded LLM-reward stochasticity and inherited-policy drift. Post-peak fitness degradation in extended runs is characterized in Appendix I.

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best

judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [N/A]

   Justification: The paper makes no formal theoretical claims; the closed-form skill-inference equation (Section 3.2) is a standard least-squares projection whose derivation follows directly from the successor-feature framework cited (Barreto et al. [2017], Park et al. [2024]). All other claims are empirical.

   Guidelines:

   - The answer [N/A] means that the paper does not include theoretical results.
   - All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
   - All assumptions should be clearly stated or referenced in the statement of any theorems.
   - The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
   - Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
   - Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: Algorithm 1 gives the full procedure. Section 3 specifies the skill-inference and fine-tuning steps. Appendix E describes all three main experimental configurations and ablations; Appendix B lists observation/action spaces, termination conditions, and SAC training hyperparameters per environment; Appendix D gives exact fitness-function formulas. The skill-prior pretraining hyperparameters (skill dim, learning rates, mix ratio, etc.) are enumerated in Appendix B.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
   - While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example

(a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
(b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
(c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
(d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

Answer: [No]

Justification: To preserve double-blind anonymity, code and pretrained checkpoints are not linked from the submission. An anonymized code release containing training scripts, evolution loop, skill-prior pretraining, and exact configurations for all reported experiments will accompany the camera-ready version. All datasets used (MuJoCo environments, D4RL `door-expert-v1`, `door-human-v1`) are publicly available. The algorithm and all hyperparameters are fully specified in Algorithm 1 and Appendix B, sufficient to reimplement the system.

Guidelines:

- The answer [N/A] means that paper does not include experiments requiring code.
- Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
- The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
- The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
- At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

Answer: [Yes]

Justification: Section 4.3 specifies the evolutionary horizon (7 generations $\times$ 16 individuals $\times$ 5 seeds). Appendix B lists SAC hyperparameters (MlpPolicy 2×256, lr 3×10^{-4}, batch size 256, $\gamma = 0.99$, $\tau = 0.005$), per-individual step budgets (500K for FORGE, 5M for the REvolve baseline), and skill-prior pretraining hyperparameters (skill dim $d = 50$ matching ϕ dimension by construction, hidden dim 1024, ϕ lr 5×10^{-4}, batch size 1024,

offline/online mix ratio 0.5, skill resampling every 300 steps). The five training and ablation configurations (REvolve-auto, FORGE island, FORGE elite, inheritance-only, skill-prior-only) are described in Appendix E; the three pretraining buffer variants (RND, expert-only, RND + expert) are described in Section 4.2.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
- The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

Answer: [Yes]

Justification: Main quantitative results report mean $\pm$ 1 standard deviation over 5 independent seeds. Table 2 uses the form "mean $\pm$ std"; Figures 2, 3, 4, and 5 show shaded ± 1 std bands computed from the same 5 seeds, while appendix figures (6, 7) use 3 seeds and Figure 8 uses 2 seeds (noted in their captions). The variability source is inter-seed variation (different RNG for SAC initialization, exploration, evolutionary sampling, and LLM mutation order). For the elite-inheritance condition (FORGE (elite) curves in Figure 2), the wide band reflects bimodal seed behavior in which a few seeds reach high fitness early while others stagnate.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

Answer: [Yes]

Justification: Total environment-step budgets are reported in Section 4.5 (560M for REvolve-auto vs 57M for FORGE) and decomposed in Table 4 in Appendix F into per-individual cost (500K steps for FORGE, 5M for the baseline) and one-time pretraining cost (1M steps). The Cost Analysis paragraphs in Appendix F also note that skill inference is effectively instantaneous compared to training, and that the pretrained feature space

amortizes across downstream tasks on the same embodiment. Section 4.3 states that a single FORGE generation, parallelized over 16 candidates on 8 NVIDIA GeForce RTX 4090 GPUs, takes approximately 6 hours per environment; Appendix B additionally notes that pretraining was performed on a single NVIDIA GPU with 32 GB VRAM. SAC for baselines is implemented via Stable-Baselines3 [Raffin et al., 2021].

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
- The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
- The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

Answer: [Yes]

Justification: The work uses only publicly available simulated environments (MuJoCo, AdroitHand) and publicly released offline datasets (D4RL). No human-subjects data, scraped web data, or personally identifiable information is involved. The authors have reviewed the NeurIPS Code of Ethics and confirm compliance.

Guidelines:

- The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
- If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
- The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

Answer: [N/A]

Justification: The contribution is foundational methodology for LLM-driven reward search in simulated continuous-control tasks (locomotion, standup, door-opening). It does not introduce a deployable system, generative model, or dataset that has a direct path to positive or negative societal impact beyond the general considerations applicable to all reinforcement-learning research. The reduction in human-feedback cost (Appendix F) is a generic efficiency improvement rather than a domain-specific application.

Guidelines:

- The answer [N/A] means that there is no societal impact of the work performed.
- If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.

- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

    Answer: [N/A]

    Justification: The paper does not release any high-risk artifacts. The pretrained skill-prior checkpoints are task-agnostic locomotion/manipulation policies for simulated MuJoCo embodiments, which carry no plausible misuse risk.

    Guidelines:

    - The answer [N/A] means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: All third-party assets are cited with their original publications: MuJoCo [Todorov et al., 2012], AdroitHand [Rajeswaran et al., 2018], D4RL datasets including `door-expert-v1` and `door-human-v1` [Fu et al., 2020], Stable-Baselines3 [Raffin et al., 2021], SAC [Haarnoja et al., 2018], HILP [Park et al., 2024], RND [Burda et al., 2019], and the REvolve evolutionary framework [Hazra et al., 2025]. All listed assets are released under permissive open-source licenses (Apache 2.0 or MIT) and are used within their stated terms for non-commercial research.

    Guidelines:

    - The answer [N/A] means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
    - If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
    - For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.

- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [N/A]

    Justification: The paper does not release new datasets or large pretrained models as primary contributions. The proposed system (FORGE) is described in full in Algorithm 1 and the appendices; an anonymized code release planned for the camera-ready version will include README and configuration documentation alongside the implementation.

    Guidelines:

    - The answer [N/A] means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [N/A]

    Justification: The paper does not involve crowdsourcing or any human-subjects research. REvolve's human-in-the-loop peak fitness (~ 0.875 on Adroit Hand) cited as a contextual reference in Section 4.4 is reported directly from the original publication [Hazra et al., 2025]; we conducted no new human-feedback collection.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [N/A]

    Justification: The paper does not involve human subjects, so no IRB approval was required.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.

- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

    Answer: [Yes]

    Justification: An LLM is a core methodological component: it serves as the reward designer G in Algorithm 1, generating and mutating reward-function code each generation. The role, prompting structure, and integration of the LLM into the evolutionary loop are described in Sections 2–3 and Algorithm 1. The reward designer is Qwen3-Coder (480B-A35B-Instruct) [Qwen Team, 2025], as stated in Section 4.1. To isolate the LLM's contribution from the proposed co-evolution mechanism, the REvolve-auto baseline (Section 4.4) is re-run with the same LLM as FORGE, so any reported gap is attributable to the method rather than to language-model quality.

    Guidelines:

    - The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
    - Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.