DUAL-AGENT CO-DESIGN OF TERRAIN CURRICULA AND REWARD FUNCTIONS FOR MIXED-TERRAIN QUADRUPEDAL LOCOMOTION

Anonymous authors
Paper under double-blind review

ABSTRACT

Robust quadrupedal locomotion over mixed terrain requires the robot to preserve stable motion as contact, posture, and momentum demands change between terrain segments. These transitions couple two training decisions: the terrain curricula used for policy training and the reward functions used to guide policy optimization. Optimizing them separately can misalign reward incentives with the behaviors required by the training terrains. We introduce a policy-mediated dual-agent framework in which a Terrain Curriculum Agent and a Reward Hypothesis Agent co-design executable mixed-terrain training arenas and arena-informed reward programs. Transition-frontier feedback computed from rollouts of the induced policy identifies unresolved transitions and links them to local geometry and robot–terrain interactions, enabling both agents to address the same locomotion bottleneck. Candidate designs are grounded by proximal policy optimization in physics simulation and selected through hierarchical behavioral validation that combines reward screening with gait-constrained cross-arena evaluation. Experiments on an independently constructed mixed-terrain test set show gains over terrain-evolution, reward-search, hand-designed, and task-curriculum baselines in full-course success and traversal progress. The advantage is larger on closely spaced terrain compositions than on isolated components, and ablations show that terrain evolution and explicit designer feedback complement reward optimization, particularly on difficult courses. Additional evaluations on three public terrain suites support the transfer of the learned policies across terrain generators.

1 INTRODUCTION

Reinforcement learning has enabled quadrupedal robots to traverse individual terrains such as ramps, stairs, gaps, and hurdles (Zhuang et al., 2023; Cheng et al., 2024; Hoeller et al., 2024). Real-world environments, however, interleave these terrains within a single course, and competence on each terrain in isolation does not compose into reliable mixed-terrain traversal. The difficulty concentrates at the transitions between terrains, where the robot must adapt its footholds, contact timing, and posture within a few strides while carrying momentum shaped by the preceding segment. Training a single policy that reliably negotiates such transitions remains an open problem.

Training such a policy hinges on what the robot practices and what it is rewarded for: a terrain curriculum progressively exposes diverse obstacle combinations, while a reward function reinforces the behaviors those combinations demand. The two are tightly coupled, since added difficulty is productive only if the reward credits the behaviors it requires, and a reward shapes learning only on the terrains the curriculum presents. A stalled transition may therefore stem from either component or their mismatch, and updating one alone can leave the other misaligned, particularly near the current policy's competence frontier.

Existing approaches do not yet provide such coordination. Quadrupedal locomotion systems have traditionally hand-crafted both components (Zhuang et al., 2023; Cheng et al., 2024; Hoeller et al., 2024), relying on expert priors and costly tuning to produce designs that generalize poorly to new obstacle compositions. Recent work has shown that LLMs can automate this design, but existing methods optimize only one axis at a time: Eurekaverse and EnvGen evolve the training environment

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

under a fixed reward (Liang et al., 2024; Zala et al., 2024), while Eureka and Text2Reward evolve the reward in a fixed environment (Ma et al., 2024b; Xie et al., 2024). In either case, the fixed component constrains what optimizing the other can achieve. LLM-based joint design of multiple training components has shown promise in other domains (Ryu et al., 2025; Zhu et al., 2025; Wang et al., 2024; Faldor et al., 2025; Wu, 2025; Wang et al., 2026), yet it does not carry over directly to mixed-terrain locomotion, which poses challenges of its own. Co-evolving terrain and reward calls for a diagnostic precise enough to attribute failures to individual transitions, as well as a training and evaluation architecture that keeps a two-sided search tractable when every candidate must be assessed through policy learning.

We present a policy-mediated dual-agent framework that co-designs terrain curricula and reward functions. **A Terrain Curriculum Agent** generates training arenas with diverse course compositions, each instantiated at graded difficulty levels. **A Reward Hypothesis Agent** proposes arena-conditioned reward programs. The coupling is policy-mediated: terrain and reward programs jointly shape the learned policy, whose transition-frontier feedback coordinates their subsequent updates. The terrain agent responds to rewards through policy behavior, while the reward agent additionally conditions on the newly generated terrain. This feedback identifies each branch's competence frontier and blocking goal-to-goal transition, linking them to local geometry and interaction diagnostics that guide complete-course regeneration and reward synthesis around the same locomotion bottleneck. Candidate rewards are screened through short PPO runs; selected policies receive further training and undergo gait-constrained evaluation across all arenas of the generation. The best-transferring policy initializes the next generation and supplies feedback to both agents, closing the co-design loop.

On an independently constructed mixed-terrain test set excluded from policy training, our framework achieves 82.5% full-course success, with an absolute success-rate margin of 20.0% over the strongest baseline, Eureka. The margin increases to 22.3% on higher-difficulty courses. Evaluations of isolated and composed terrains show that our policy combines strong component competence with a larger advantage over the baselines when components are closely spaced. Ablations identify reward optimization as a central contributor and show that terrain evolution and explicit designer feedback further improve performance, with larger benefits on difficult courses. The same trained policies also achieve competitive performance on Extreme Parkour, Robot Parkour Ground, and ETH Confined-Space, providing complementary evidence of transfer across terrain generators.

Our main contributions are as follows:

- We propose a policy-mediated dual-agent framework that co-designs terrain curricula and reward functions for mixed-terrain quadrupedal locomotion.
- We develop transition-frontier feedback and hierarchical behavioral selection, combining local failure diagnosis, within-arena reward screening, and cross-arena policy selection to guide terrain–reward co-design.
- We demonstrate that our framework outperforms strong baselines in mixed-terrain traversal, with larger advantages on difficult courses and closely spaced terrain compositions. Ablations support the complementary contributions of terrain evolution, reward optimization, and designer feedback.

2 Related Work

LLM Environment Design. Automatic environment design predates LLMs: curriculum learning adapts task distributions to learner progress (Bengio et al., 2009; Florensa et al., 2018), and unsupervised environment design co-evolves environments with agents through novelty or regret (Wang et al., 2019; Jiang et al., 2021; Parker-Holder et al., 2022). LLMs extend this line from parameter search to executable task and world generation: EnvGen revises structured simulator configurations from policy feedback, and Eurekaverse evolves terrain programs according to quadruped performance (Zala et al., 2024; Liang et al., 2024). These systems expand or adapt the training environment distribution while leaving reward design outside that loop.

LLM Reward Design. Reward design classically relies on manual shaping, and misspecified objectives invite unintended optima, motivating inverse reward design and analyses of reward gaming

(Ng et al., 1999; Hadfield-Menell et al., 2017; Skalse et al., 2022). Eureka and Text2Reward generate executable reward code and revise it through policy training, execution feedback, or human correction (Ma et al., 2024b; Xie et al., 2024); DrEureka adds safety rewards and domain randomization for sim-to-real transfer (Ma et al., 2024a), and MAPL learns multi-objective locomotion preferences under a fixed terrain curriculum (Chen et al., 2026). These methods revise reward functions while the training environment distribution remains fixed, externally scheduled, or optimized in a separate stage.

Joint Environment and Reward Design. Several systems couple environment or curriculum construction with reward specification: RoboGen and OMNI-EPIC generate scenes or open-ended environment programs containing reward and success logic (Wang et al., 2024; Faldor et al., 2025), CurricuLLM jointly designs reward functions and goal distributions across curriculum stages (Ryu et al., 2025), AURA revises curricula together with rewards and training configurations (Zhu et al., 2025), MAESTRO regenerates auxiliary rewards as a traffic curriculum changes (Wu, 2025), RLAnything jointly updates tasks, policies, and a generative reward model (Wang et al., 2026), and LIMEN searches executable observation and reward programs under a fixed environment (Jaswal et al., 2026). A complementary line grounds design choices in policy behavior: PAIRED's regret-based environment generation, Enhanced POET's transfer across environment–agent archives, COvolve's cross-play payoff matrix, and Eureka's fitness-based reward ranking (Dennis et al., 2020; Wang et al., 2020; Sygkounas et al., 2026; Ma et al., 2024b). Unlike these systems, we localize failures to goal-to-goal transitions within mixed courses, use the same policy-grounded evidence to coordinate terrain and reward updates, and select the surviving policy by gait-constrained cross-arena transfer rather than own-pair fitness.

Quadrupedal Locomotion over Complex Terrain. Learning-based controllers have progressed from agile motor skills and blind rough-terrain locomotion to perceptive traversal in the wild, with massively parallel simulation accelerating training (Hwangbo et al., 2019; Lee et al., 2020; Miki et al., 2022; Rudin et al., 2022). Robot Parkour Learning combines soft-to-hard exploration with expert distillation, Extreme Parkour uses a unified depth-conditioned controller with an automatic terrain curriculum, and ANYmal Parkour integrates 3D perception, a locomotion skill library, and hierarchical navigation (Zhuang et al., 2023; Cheng et al., 2024; Hoeller et al., 2024). These systems automate progression or skill selection, but their terrain families and reward functions remain separately engineered; we use transition-level policy feedback to co-design terrain curricula and reward functions for a single mixed-terrain policy.

3 METHOD

We study goal-conditioned quadrupedal locomotion on mixed-terrain courses that combine unstructured continuous terrain with discrete obstacles. The robot traverses each course by following a sequence of G ordered navigation goals. Our design variables are a *training arena* $\mathcal{E}$ and an executable reward program r. An arena comprises C compositional course families, which we call *branches*. Each branch is represented by a terrain-generation program that produces a complete course as a function of difficulty $d \in [0, 1]$. Instantiating the programs at L ordered difficulty levels produces a $C \times L$ grid of training courses. The branch index c identifies a composition; the level index $\ell \in \{0, \ldots, L-1\}$ controls its difficulty. One reward program is shared across all courses in an arena. Across generations, the terrain agent generates new terrain programs conditioned on parent programs and policy feedback. Within each arena, a fixed curriculum rule adjusts each robot's difficulty level according to its goal progress during PPO training.

We search for arena–reward pairs through the traversal performance of the policies they induce under finite-budget PPO training (Schulman et al., 2017). At generation t, the framework generates B new training arenas (child arenas) and M candidate reward programs per arena, using the incumbent policy–reward pair $(\pi^\star_{t-1}, r^\star_{t-1})$. Short PPO runs evaluate the candidate rewards, and the selected policy for each arena receives additional training. A common cross-arena evaluation then selects the policy carried into the next generation. Rollout feedback from the selected policy guides the generation of new terrain and reward programs. The process starts from a pretrained walking policy and a bootstrap reward. Figure 1 summarizes the framework; Algorithm 1 gives the full procedure.

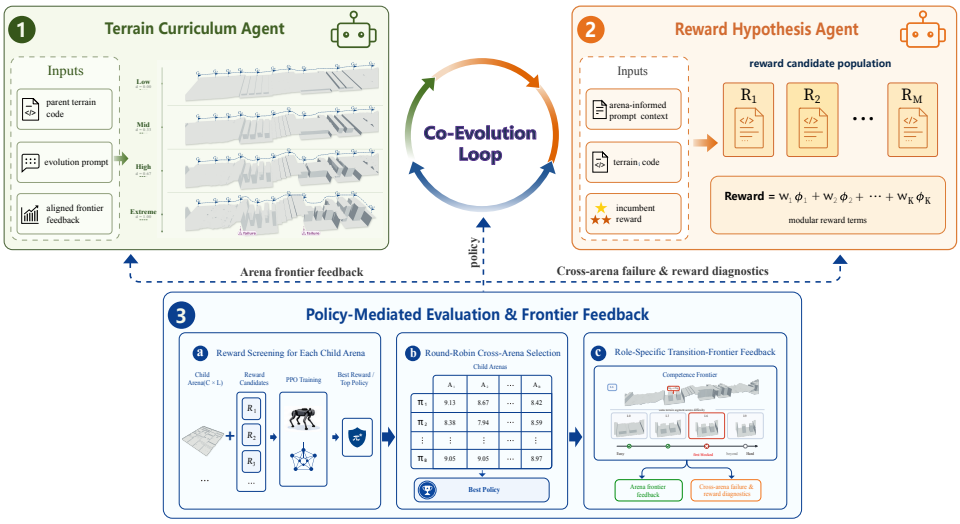


Figure 1: Framework overview. (1) The Terrain Curriculum Agent generates branch programs from parent code and transition-frontier feedback, with each branch instantiated across difficulty levels. (2) The Reward Hypothesis Agent proposes candidate reward programs conditioned on each child arena, the incumbent reward, and shared policy feedback. (3) Short PPO screening and continued training precede gait-constrained cross-arena selection. The selected policy provides transition-frontier feedback to both agents for the next generation, closing the terrain–reward co-design loop.

3.1 Transition-Frontier Feedback

An aggregate traversal score does not indicate where a policy becomes blocked or provide the local context needed for terrain and reward updates. We therefore construct transition-frontier feedback that combines the policy's competence boundary, blocking goal-to-goal intervals, and associated geometry and interaction statistics. We evaluate a frozen policy π with curriculum adaptation disabled and parallel environments distributed evenly across an arena's branch–level cells. For a sampled cell (c, ℓ) with $n_{c,\ell}$ recorded episodes, its normalized progress is $p_{c,\ell} = \frac{1}{n_{c,\ell}} \sum_e g_e/G$, where g_e is the number of goals reached in an episode from that cell. A cell is *verified* when $n_{c,\ell} \geq n_{\min}$ and $p_{c,\ell} \geq \tau$. The *verified depth* $D_c \in \{0, \ldots, L\}$ is the number of consecutive verified levels beginning at level zero. The *frontier* is the first unverified level, or the hardest level when the entire branch is verified; ρ_c denotes its mean normalized progress. We score the policy on the arena by

$$S(\pi, \mathcal{E}) = \frac{1}{C} \sum_{c=1}^{C} \big(D_c + \epsilon \rho_c\big). \tag{1}$$

For $0 < \epsilon < 1/C$, the ranking prioritizes mean verified depth and uses mean frontier progress to break ties. Because S uses goal attainment, it provides a common criterion across candidates with different reward programs.

Within each frontier cell, let $q_{c,k}$ be the fraction of episodes reaching at least k goals, with $q_{c,0} = 1$. When incomplete traversals occur, we identify the *blocking transition* as

$$k_c^\star = \arg\max_{1 \leq k \leq G} \big(q_{c,k-1} - q_{c,k}\big). \tag{2}$$

The difference is the fraction of episodes ending with exactly $k-1$ goals reached, so $k_c^\star$ identifies the goal-progress interval in which the largest fraction of episodes fail. The convention $q_{c,0} = 1$ includes failures before the first goal. An interval can span an obstacle traversal or a change of terrain type. If all episodes in a frontier cell reach all G goals, no blocking transition is assigned.

We denote the resulting feedback record by $F(\pi, r, \mathcal{E})$, where r is the reward program used to compute per-term reward contributions. For each branch, the record combines verified depth, frontier progress, and the blocking transition with a geometry description and episode-level statistics. We summarize the geometry of the identified interval from the branch's terrain-generation program at the frontier difficulty. The statistics include mean and maximum goals reached, episode duration,

foot–edge contacts, and per-term reward contributions aggregated over episodes in the frontier cell. Together, they provide a localized failure indicator and its physical and behavioral context for the design updates.

3.2 Policy-Mediated Terrain–Reward Co-Design

To keep terrain and reward updates focused on the same locomotion bottlenecks, both design agents use feedback from the incumbent policy evaluated on the retained parent arenas. The coupling is asymmetric: the Terrain Curriculum Agent responds to policy behavior induced by previous rewards, while the Reward Hypothesis Agent additionally conditions on the newly generated terrain. At generation t, the shared feedback from the retained parent arenas $\mathcal{P}_{t-1}$ is:

$$\mathcal{F}_{t-1} = \left\{ F(\pi^\star_{t-1}, r^\star_{t-1}, \mathcal{E}) \;\middle|\; \mathcal{E} \in \mathcal{P}_{t-1} \right\}. \tag{3}$$

At initialization, $\mathcal{P}_{-1} = \mathcal{F}_{-1} = \varnothing$, and initial arena and reward generation use the task specification and bootstrap reward.

In subsequent generations, each child arena $\mathcal{E}^b_t$ is assigned a retained parent arena in round-robin order. For each branch of that parent arena, the terrain agent receives its terrain-generation program, the parent's branch-level feedback table, and the branch's frontier record. Using the observed depth, progress, and failure geometry, the agent generates a new terrain program specifying the complete course's obstacle composition, ordering, and goal placement. The agent is instructed to adjust difficulty and vary course structure according to the observed policy competence, including simplifying crossings with inconsistent traversal success. Its generation instructions specify an easy-to-hard difficulty parameterization and obstacle-specific geometric bounds. Generated terrain programs must execute successfully and pass the terrain constructor's validity checks before training.

For each new arena, the reward agent synthesizes M candidate reward programs $\{r^{b,m}_t\}_{m=1}^{M}$. Each reward consists of named, weighted functions of training-time robot and terrain state, shared across the arena's branches and difficulty levels. The reward agent's generation context combines representative terrain-generation programs from the new arena, the incumbent reward $r^\star_{t-1}$, and compact summaries of frontier cases with low verified depth from $\mathcal{F}_{t-1}$, including per-term reward contributions. These terrain programs describe the new arena's course compositions, while the parent evaluations characterize the incumbent policy's behavior on the retained parent arenas. To promote diversity in the reward search, each candidate is generated with a different behavioral focus, such as contact quality, transition timing, or posture recovery. The incumbent reward serves as a reference; all M candidates are newly generated. The resulting arena–reward candidates undergo hierarchical selection (Section 3.3).

3.3 Hierarchical Behavioral Selection

A policy that performs well on its own training arena may still generalize poorly to other arenas. Selection operates at two levels: reward candidates are first screened within each arena, and the resulting policies are then compared across arenas. Within each child arena, reward screening trains a separate policy under each candidate reward program. All M policies are initialized from the same incumbent policy $\pi^\star_{t-1}$ and trained for H_{s} iterations using identical PPO settings. Parallel robots remain assigned to their compositional branches, with an automatic curriculum adjusting each robot's difficulty level according to goal progress. Experience from all branches updates one shared policy per reward candidate. Candidate policies are trained with $r_{\mathrm{train}} = r + r_{\mathrm{gait}}$, where the fixed gait regularization term r_{gait} is shared across candidates and discourages prolonged single-foot suspension during supported locomotion.

Let $\widetilde{\pi}^{b,m}_t$ denote the policy trained under reward $r^{b,m}_t$ during this screening stage. A binary gait gate $\gamma(\pi, \mathcal{E})$ checks whether each foot meets the prescribed contact-fraction and support-share thresholds (Appendix D). We continue the highest-S eligible policy for H_1 iterations to obtain $(\pi^b_t, \hat{r}^b_t)$; Algorithm 1 specifies bounded retries and a failure exit when no candidate is eligible. Each screening run starts with fresh optimizer state; continued training restores the selected policy's training state.

After continuation training, we evaluate each resulting policy on every child arena, yielding the matrix $A_{ij} = S(\pi^i_t, \mathcal{E}^j_t)$. Each policy is thus assessed on the same population of courses, including the arenas used to train its competitors. Let $\mathcal{I}_t = \{i : \gamma(\pi^i_t, \mathcal{E}^j_t) = 1 \text{ for all } j\}$ be the eligible policy

indices. For $\mathcal{I}_t \neq \varnothing$, the policy retained for the next generation is selected by

$$i^\star = \arg\max_{i \in \mathcal{I}_t} \frac{1}{B} \sum_{j=1}^{B} A_{ij}, \qquad (\pi_t^\star, r_t^\star) = (\pi_t^{i^\star}, \hat{r}_t^{i^\star}). \tag{4}$$

If $\mathcal{I}_t = \varnothing$, the run terminates with failure. We retain the N_p arenas with the highest scores in the selected row as $\mathcal{P}_t$ and construct $\mathcal{F}_t$ from that same policy's evaluations on those arenas. The selected policy initializes the next generation's training runs, and its paired reward becomes the reference for reward generation. After T generations, the co-design loop outputs the policy $\pi_{T-1}^\star$.

4 Experiments

We evaluate the framework through four questions:

Q1 (Performance and evolution). Does policy-mediated terrain–reward co-design improve mixed-terrain traversal over the baselines under a common PPO budget, and how does its performance evolve across generations?

Q2 (Component competence). How does the learned policy compare with the baselines on isolated terrain components?

Q3 (Terrain composition). Can the learned policy reliably negotiate transitions between terrain components that it can traverse in isolation?

Q4 (Component ablations). How do terrain evolution, reward optimization, and explicit designer feedback contribute to mixed-terrain traversal?

4.1 Experimental Setup

Training setup and baselines. We train all Go1 policies in Isaac Gym (Makoviychuk et al., 2021) using PPO and the Extreme Parkour control stack (Cheng et al., 2024). All methods use 6,144 parallel environments, 24-step rollouts, and a common nominal allocation of 160,000 PPO iterations per seed, summed over candidate and branch training. The 1,500-iteration walking bootstrap is accounted for separately. We compare Ours with four baselines: Human-Designed, which fixes both the terrain generator and locomotion reward; Eureka (Ma et al., 2024b), which searches for a reward on the fixed generator; Eurekaverse (Liang et al., 2024), which evolves terrain while retaining its fixed reward and native selection procedure; and CurricuLLM (Ryu et al., 2025), which jointly designs reward functions and goal distributions across curriculum stages. Ours allocates the common budget over $T = 4$ generations with $B = 4$ arenas per generation. On each arena, we screen $M = 8$ reward candidates for $H_\mathrm{s} = 1{,}000$ PPO iterations each and continue the selected policy for $H_\mathrm{l} = 2{,}000$ additional iterations. Both design agents access GPT-5.5 through the API with reasoning effort set to medium.

Benchmark and evaluation. We construct an independent mixed-terrain test set by randomly combining and ordering segments from different terrain types into 16 courses. Each 24 m course contains $G = 11$ ordered navigation goals and is instantiated at difficulty levels 0–9, yielding 160 course–difficulty combinations. The test courses are held fixed across methods and kept separate from policy training and the terrain–reward design loop. All methods first complete their prescribed training budgets without using this benchmark. For each method and seed, we report the evaluated policy with the highest overall full-course success. We measure *full-course success*, the fraction of episodes reaching all goals, and *mean goals reached*, which captures partial traversal. Metrics are averaged within each course–difficulty combination and then equally across combinations. All tables in this section report mean $\pm$ standard deviation over three independent training seeds; differences are computed from the rounded means reported in the tables. We additionally report results on levels 5–9, the higher-difficulty half of the benchmark, to assess performance on challenging courses separately from the all-level average.

4.2 MIXED-TERRAIN TRAVERSAL

Table 1: Mixed-terrain performance of Ours and the baselines.

	All levels		Levels 5–9	
Method	Success (%) ↑	Goals ↑	Success (%) ↑	Goals ↑
Human-Designed	54.5 ± 6.3	7.99 ± 0.46	35.7 ± 3.2	6.65 ± 0.31
Eurekaverse	55.3 ± 21.0	8.24 ± 1.29	35.8 ± 26.9	7.04 ± 1.67
Eureka	62.5 ± 5.5	8.96 ± 0.34	49.8 ± 5.6	8.12 ± 0.39
CurricuLLM	44.2 ± 3.5	7.38 ± 0.42	29.4 ± 4.7	6.22 ± 0.58
Ours	$\mathbf{82.5 \pm 0.7}$	$\mathbf{9.88 \pm 0.12}$	$\mathbf{72.1 \pm 1.3}$	$\mathbf{9.19 \pm 0.12}$

Ours achieves 82.5% full-course success and reaches 9.88 goals on average (Table 1). It outperforms Eureka, the strongest baseline, with an absolute success-rate margin of 20.0% and 0.92 more goals. Ours also achieves the highest mean success rate and mean goal count at every difficulty level (Figure 2).

On levels 5–9, Ours achieves 72.1% success compared with 49.8% for Eureka, an absolute margin of 22.3%. Its mean goal count is 9.19 versus 8.12, increasing the advantage from 0.92 goals overall to 1.07 goals on harder courses.

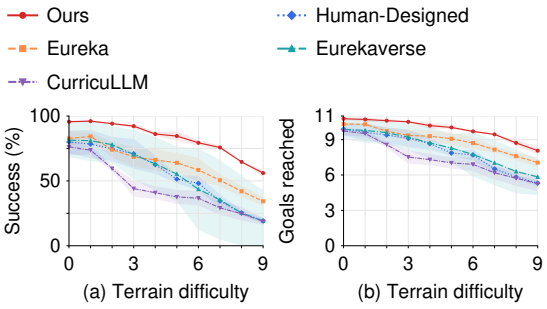


Figure 2: Baseline comparison by difficulty.

Evaluations of the same trained policies on three public terrain suites further support transfer across terrain generators and course structures (Appendix F.3).

Figure 3(a) shows sustained improvement across co-design generations in a single-seed comparison. Ours leads the baselines at each completed generation, increasing full-course success from 58.50% after the first generation to 82.92% after the fourth. Its final success rate exceeds that of Eurekaverse (74.37%), the strongest baseline in this trace, by an absolute margin of 8.55%.

Figure 3(b) traces feedback-guided terrain–reward co-design across generations. Each generation regenerates complete courses and synthesizes rewards for the resulting arenas; colored regions highlight representative feedback–edit correspondences.

In R1, evaluation identifies a weak transition across rough terrain with a transverse ridge. This feedback informs the introduction of broad support pads in R2, alongside stronger progress incentives and stable-landing rewards. The regenerated course also relocates the gap and support islands and increases the gap width, changing the sequence and geometry of the traversal challenges. Evaluation after R2 identifies residual weakness in the terminal approach, informing its reconstruction around a rising terminal ramp in R3, together with further changes to obstacle ordering and support geometry. Meanwhile, broader parent-arena feedback guides reward updates that emphasize safe support and penalize slipping and excessive contact forces. Evaluation after R3 highlights fragile traversal where a gap leads into a laterally tilted, rough section. In R4, the terrain agent removes lateral tilt, enlarges the flat goal region, and moves the gap downstream, while the reward agent introduces explicit shaping for transition direction, heading alignment, and touchdown. The resulting policy supplies feedback for subsequent design updates, closing the terrain–reward co-design loop.

4.3 COMPONENT COMPETENCE AND TRANSITION RELIABILITY

We compare the same policies on isolated terrain components and ordered A-to-B compositions to assess whether component competence extends to terrain transitions (Table 2). The evaluation uses 12 ordered terrain pairs and three fixed layouts, with composed trials at boundary-to-boundary clearances of 0.10 m, 0.35 m, and 1.50 m. Larger clearances provide more space for the robot to

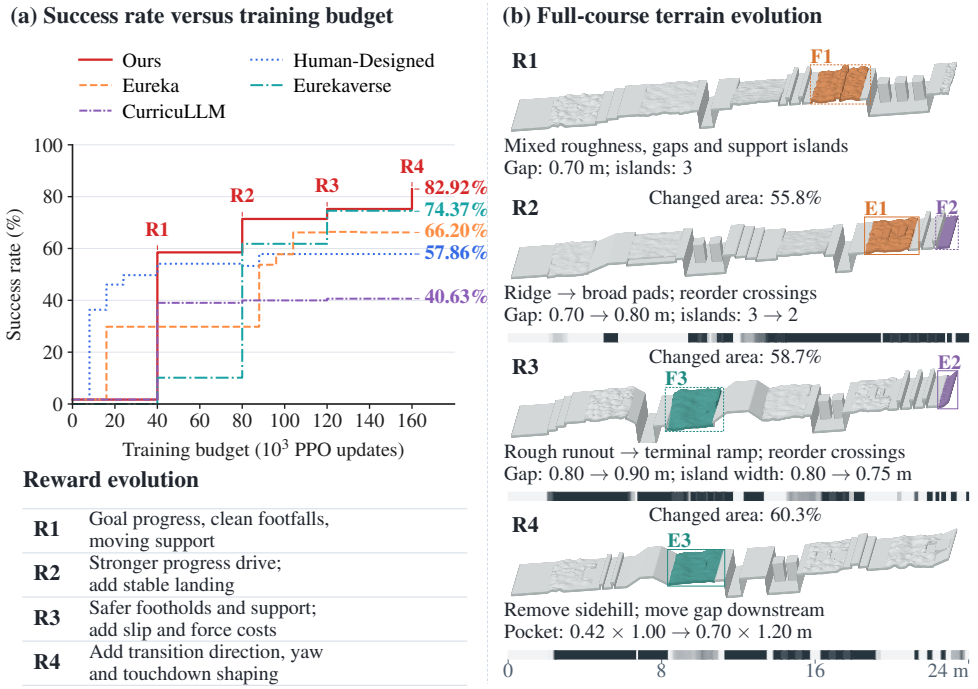


Figure 3: Training dynamics (single seed) and terrain evolution ($d = 1$). Complete courses are regenerated each round; colored regions highlight representative feedback–edit correspondences. Dashed F1–F3 mark feedback regions; solid E1–E3 mark matching next-round edits. Gray bands encode changed-width fractions (light: 0%; dark: 100%).

regain a stable posture before entering the next terrain component, while smaller clearances test its ability to negotiate successive components with limited recovery space.

Table 2: Success (%) on isolated components and A-to-B compositions. Distances indicate boundary-to-boundary clearance.

	Controls (0.10 m)		A-to-B composition		
Method	A-only	B-only	0.10 m	0.35 m	1.50 m
Human-Designed	93.4 ± 1.9	93.3 ± 1.1	77.6 ± 4.9	79.8 ± 4.7	87.2 ± 3.5
Eurekaverse	97.2 ± 3.3	97.4 ± 2.0	86.7 ± 4.6	91.3 ± 6.9	94.6 ± 6.0
Eureka	95.8 ± 2.7	96.9 ± 2.4	86.2 ± 5.6	89.0 ± 5.8	93.8 ± 5.1
CurricuLLM	94.6 ± 4.2	94.0 ± 4.6	77.6 ± 6.0	82.1 ± 9.7	89.3 ± 8.5
Ours	$\mathbf{97.9 \pm 1.5}$	$\mathbf{98.5 \pm 0.9}$	$\mathbf{92.2 \pm 3.4}$	$\mathbf{94.4 \pm 2.0}$	$\mathbf{97.2 \pm 0.7}$

Ours achieves 97.9% and 98.5% success on the A-only and B-only controls, respectively, compared with 97.2% and 97.4% for Eurekaverse, the strongest baseline on both controls. The performance gap widens under composition: at 0.10 m clearance, Ours reaches 92.2% success versus 86.7% for Eurekaverse, an absolute margin of 5.5%. The composed trials thus distinguish policies whose isolated-component success rates are closely matched.

Increasing clearance improves success for all methods. At 0.35 m and 1.50 m, Ours achieves 94.4% and 97.2%, retaining absolute margins of 3.1% and 2.6% over Eurekaverse. The largest margin at the smallest clearance highlights the framework's advantage in traversing closely spaced terrain sequences while retaining strong isolated-component performance. The same policies also achieve the highest mean success in each of twelve single-terrain families when averaged over difficulty levels (Appendix F.1). The complete component-to-composition evaluation protocol is provided in Appendix F.2.

4.4 Ablation Study

We compare the full framework with three ablated variants to assess the contributions of terrain evolution, reward optimization, and explicit designer feedback (Table 3; Figure 4).

Table 3: Component ablations on the mixed-terrain benchmark.

	All levels		Levels 5–9	
Variant	Success (%) ↑	Goals ↑	Success (%) ↑	Goals ↑
w/o Terrain Evolution	75.1 ± 5.3	9.41 ± 0.29	59.0 ± 8.6	8.35 ± 0.47
w/o Reward Optimization	31.7 ± 3.3	6.08 ± 0.48	5.1 ± 4.1	4.23 ± 0.47
w/o Explicit Feedback	72.8 ± 3.8	9.25 ± 0.20	56.1 ± 5.1	8.15 ± 0.29
Ours	$\mathbf{82.5 \pm 0.7}$	$\mathbf{9.88 \pm 0.12}$	$\mathbf{72.1 \pm 1.3}$	$\mathbf{9.19 \pm 0.12}$

w/o Terrain Evolution fixes the initial terrain population while retaining reward optimization. *w/o Reward Optimization* retains terrain evolution but trains with the fixed bootstrap reward. *w/o Explicit Feedback* retains both design agents but removes explicit rollout-derived feedback from their proposal prompts, including frontier summaries, failure descriptors, contact statistics, and reward-term contributions. Selected-code inheritance, the gait gate, reward-candidate screening, and cross-arena policy selection remain active.

The full framework achieves the highest success rates and mean goal counts in both difficulty ranges.

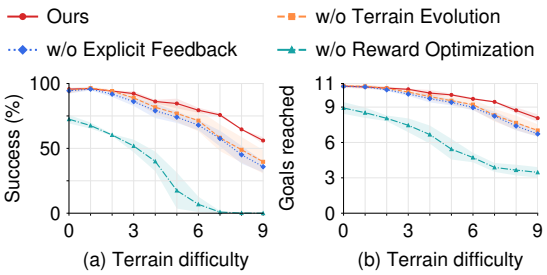


Figure 4: Component ablations by difficulty.

Terrain evolution. Fixing the terrain population while retaining reward optimization yields 75.1% success overall and 59.0% on levels 5–9, with absolute deficits of 7.4% and 13.1% relative to Ours, respectively. The gain shows the benefit of updating the training terrain population in addition to optimizing rewards.

Reward optimization. With the bootstrap reward held fixed, success decreases from 82.5% to 31.7% overall and from 72.1% to 5.1% on levels 5–9. This is the largest decrease among the three ablations, identifying reward optimization as a central contributor to learning effective traversal behavior.

Explicit designer feedback. The *w/o Explicit Feedback* variant reaches 72.8% and 56.1%, with corresponding absolute deficits of 9.7% and 16.0%. On levels 5–9, the mean number of goals reached also decreases from 9.19 to 8.15. Because both design agents, code inheritance, and policy selection remain active, this comparison shows the benefit of conditioning design updates on explicit behavioral evidence.

Across all three ablations, the larger success deficits on levels 5–9 show the value of the complete policy-mediated co-design loop for demanding mixed-terrain traversal.

5 Conclusion

We presented a policy-mediated dual-agent framework for co-designing terrain curricula and reward programs for mixed-terrain quadrupedal locomotion. Transition-frontier feedback connects policy competence and traversal bottlenecks to coordinated terrain–reward updates, closing the design loop. Experiments in simulation show improved mixed-terrain traversal and a larger advantage on closely spaced compositions than on isolated components. Ablations demonstrate that terrain evolution and explicit designer feedback complement reward optimization, highlighting the value of jointly adapting training experience and learning objectives to the policy's evolving capabilities.

AI USE STATEMENT

Large language models were used to polish the language and improve the readability of the manuscript. AI assistance also supported literature search, feedback on research methodology and experimental design, result interpretation, figure preparation, and LaTeX formatting. Within the proposed framework, LLMs generate terrain and reward programs, as described in Section 3. The authors are responsible for the final content, scientific claims, and reported results.

REFERENCES

Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, pp. 41–48, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605585161. doi: 10.1145/1553374.1553380. URL https://doi.org/10.1145/1553374.1553380.

Xiyue Chen, Muhan Lin, Shuyang Shi, and Joseph Campbell. MAPL: Multi-objective preference learning for robot locomotion. *arXiv preprint arXiv:2606.25398*, 2026.

Xuxin Cheng, Kexin Shi, Ananye Agarwal, and Deepak Pathak. Extreme parkour with legged robots. In *2024 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 11443–11450, 2024. doi: 10.1109/ICRA57147.2024.10610200.

Michael Dennis, Natasha Jaques, Eugene Vinitsky, Alexandre Bayen, Stuart Russell, Andrew Critch, and Sergey Levine. Emergent complexity and zero-shot transfer via unsupervised environment design. *Advances in neural information processing systems*, 33:13049–13061, 2020.

Maxence Faldor, Jenny Zhang, Antoine Cully, and Jeff Clune. OMNI-EPIC: Open-endedness via models of human notions of interestingness with environments programmed in code. In Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu (eds.), *International Conference on Learning Representations*, volume 2025, pp. 85260–85385, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/d40d7cbe7210f8a13ea0149eeae9c6de-Paper-Conference.pdf.

Carlos Florensa, David Held, Xinyang Geng, and Pieter Abbeel. Automatic goal generation for reinforcement learning agents. In Jennifer Dy and Andreas Krause (eds.), *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pp. 1515–1528. PMLR, 10–15 Jul 2018. URL https://proceedings.mlr.press/v80/florensa18a.html.

Dylan Hadfield-Menell, Smitha Milli, Pieter Abbeel, Stuart J Russell, and Anca Dragan. Inverse reward design. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (eds.), *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL https://proceedings.neurips.cc/paper_files/paper/2017/file/32fdab6559cdfa4f167f8c31b9199643-Paper.pdf.

David Hoeller, Nikita Rudin, Dhionis Sako, and Marco Hutter. Anymal parkour: Learning agile navigation for quadrupedal robots. *Science Robotics*, 9(88):eadi7566, 2024.

Jemin Hwangbo, Joonho Lee, Alexey Dosovitskiy, Dario Bellicoso, Vassilios Tsounis, Vladlen Koltun, and Marco Hutter. Learning agile and dynamic motor skills for legged robots. *Science robotics*, 4(26):eaau5872, 2019.

Akshat Singh Jaswal, Ashish Baghel, and Paras Chopra. Discovering reinforcement learning interfaces with large language models. *arXiv preprint arXiv:2605.03408*, 2026.

Minqi Jiang, Edward Grefenstette, and Tim Rocktäschel. Prioritized level replay. In Marina Meila and Tong Zhang (eds.), *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pp. 4940–4950. PMLR, 18–24 Jul 2021. URL https://proceedings.mlr.press/v139/jiang21b.html.

Joonho Lee, Jemin Hwangbo, Lorenz Wellhausen, Vladlen Koltun, and Marco Hutter. Learning quadrupedal locomotion over challenging terrain. *Science robotics*, 5(47):eabc5986, 2020.

William Liang, Sam Wang, Hung-Ju Wang, Osbert Bastani, Dinesh Jayaraman, and Yecheng Jason Ma. Eurekaverse: Environment curriculum generation via large language models. *arXiv preprint arXiv:2411.01775*, 2024.

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*. RSS, 2024a.

Yecheng Jason Ma, William Liang, Guanzhi Wang, De-An Huang, Osbert Bastani, Dinesh Jayaraman, Yuke Zhu, Jim Fan, et al. Eureka: Human-level reward design via coding large language models. In *International conference on learning Representations*, volume 2024, pp. 26516–26560, 2024b.

Viktor Makoviychuk, Lukasz Wawrzyniak, Yunrong Guo, Michelle Lu, Kier Storey, Miles Macklin, David Hoeller, Nikita Rudin, Arthur Allshire, Ankur Handa, et al. Isaac gym: High performance gpu-based physics simulation for robot learning. *arXiv preprint arXiv:2108.10470*, 2021.

Takahiro Miki, Joonho Lee, Jemin Hwangbo, Lorenz Wellhausen, Vladlen Koltun, and Marco Hutter. Learning robust perceptive locomotion for quadrupedal robots in the wild. *Science robotics*, 7(62):eabk2822, 2022.

Takahiro Miki, Joonho Lee, Lorenz Wellhausen, and Marco Hutter. Learning to walk in confined spaces using 3D representation. In *2024 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 8649–8656, 2024. doi: 10.1109/ICRA57147.2024.10610271.

Andrew Y Ng, Daishi Harada, and Stuart Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *Icml*, volume 99, pp. 278–287. Citeseer, 1999.

Jack Parker-Holder, Minqi Jiang, Michael Dennis, Mikayel Samvelyan, Jakob Foerster, Edward Grefenstette, and Tim Rocktäschel. Evolving curricula with regret-based environment design. In Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvari, Gang Niu, and Sivan Sabato (eds.), *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pp. 17473–17498. PMLR, 17–23 Jul 2022. URL https://proceedings.mlr.press/v162/parker-holder22a.html.

Nikita Rudin, David Hoeller, Philipp Reist, and Marco Hutter. Learning to walk in minutes using massively parallel deep reinforcement learning. In Aleksandra Faust, David Hsu, and Gerhard Neumann (eds.), *Proceedings of the 5th Conference on Robot Learning*, volume 164 of *Proceedings of Machine Learning Research*, pp. 91–100. PMLR, 08–11 Nov 2022. URL https://proceedings.mlr.press/v164/rudin22a.html.

Kanghyun Ryu, Qiayuan Liao, Zhongyu Li, Payam Delgosha, Koushil Sreenath, and Negar Mehr. CurricuLLM: Automatic task curricula design for learning complex robot skills using large language models. In *2025 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 4470–4477, 2025. doi: 10.1109/ICRA55743.2025.11128783.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017. URL https://arxiv.org/abs/1707.06347.

Joar Skalse, Nikolaus Howe, Dmitrii Krasheninnikov, and David Krueger. Defining and characterizing reward gaming. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (eds.), *Advances in Neural Information Processing Systems*, volume 35, pp. 9460–9471. Curran Associates, Inc., 2022. doi: 10.52202/068431-0687. URL https://proceedings.neurips.cc/paper_files/paper/2022/file/3d719fee332caa23d5038b8a90e81796-Paper-Conference.pdf.

Alkis Sygkounas, Rishi Hazra, Andreas Persson, Pedro Zuidberg dos Martires, and Amy Loutfi. COvolve: Adversarial co-evolution of large-language-model-generated policies and environments via two-player zero-sum game. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pp. 338–347, 2026.

Rui Wang, Joel Lehman, Jeff Clune, and Kenneth O. Stanley. POET: open-ended coevolution of environments and their optimized solutions. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '19, pp. 142–151, New York, NY, USA, 2019. Association

for Computing Machinery. ISBN 9781450361118. doi: 10.1145/3321707.3321799. URL https://doi.org/10.1145/3321707.3321799.

Rui Wang, Joel Lehman, Aditya Rawal, Jiale Zhi, Yulun Li, Jeffrey Clune, and Kenneth Stanley. Enhanced POET: Open-ended reinforcement learning through unbounded invention of learning challenges and their solutions. In Hal Daumé III and Aarti Singh (eds.), *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pp. 9940–9951. PMLR, 13–18 Jul 2020. URL https://proceedings.mlr.press/v119/wang20l.html.

Yinjie Wang, Tianbao Xie, Ke Shen, Mengdi Wang, and Ling Yang. RLAnything: Forge environment, policy, and reward model in completely dynamic RL system, 2026. URL https://arxiv.org/abs/2602.02488.

Yufei Wang, Zhou Xian, Feng Chen, Tsun-Hsuan Wang, Yian Wang, Katerina Fragkiadaki, Zackory Erickson, David Held, and Chuang Gan. RoboGen: Towards unleashing infinite data for automated robot learning via generative simulation, 2024. URL https://arxiv.org/abs/2311.01455.

Boyuan Wu. MAESTRO: Multi-agent environment shaping through task and reward optimization, 2025. URL https://arxiv.org/abs/2511.19253.

Tianbao Xie, Siheng Zhao, Chen Wu, Yitao Liu, Qian Luo, Victor Zhong, Yanchao Yang, and Tao Yu. Text2Reward: Reward shaping with language models for reinforcement learning. In B. Kim, Y. Yue, S. Chaudhuri, K. Fragkiadaki, M. Khan, and Y. Sun (eds.), *International Conference on Learning Representations*, volume 2024, pp. 35663–35699, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/file/9941833e8327910ef25daeb9005e4748-Paper-Conference.pdf.

Abhay Zala, Jaemin Cho, Han Lin, Jaehong Yoon, and Mohit Bansal. EnvGen: Generating and adapting environments via LLMs for training embodied agents, 2024. URL https://arxiv.org/abs/2403.12014.

Alvin Zhu, Yusuke Tanaka, Andrew Goldberg, and Dennis Hong. AURA: Autonomous upskilling with retrieval-augmented agents, 2025. URL https://arxiv.org/abs/2506.02507.

Ziwen Zhuang, Zipeng Fu, Jianren Wang, Christopher Atkeson, Soeren Schwertfeger, Chelsea Finn, and Hang Zhao. Robot parkour learning, 2023. URL https://arxiv.org/abs/2309.05665.

This document collects the complete co-design algorithm, implementation thresholds, validator rules, protocol constants, prompt templates, and qualitative terrain and reward examples referenced from the main paper. Notation follows the main paper: an arena has C branches at L difficulty levels, D_c and ρ_c denote the verified depth and frontier progress of branch c, τ is the competence threshold, and H_{s} and H_{l} are the short and additional long training budgets. The feedback record $F(\pi, r, \mathcal{E})$ includes contribution diagnostics for the reward paired with π; $\mathcal{F}_t$ collects these records over the retained parent arenas at generation t.

A Complete Co-Design Algorithm

Algorithm 1 Policy-Mediated Terrain–Reward Co-Design

Require: Bootstrap $(\pi^\star_{-1}, r^\star_{-1})$; generations T; arenas B; rewards per arena M; budgets $H_{\mathrm{s}}, H_{\mathrm{l}}$; retained parents N_{p}; additional batch retries $K \geq 0$
Ensure: Selected policy or FAILURE

$\mathcal{P}_{-1} \leftarrow \varnothing$; $\mathcal{F}_{-1} \leftarrow \varnothing$
for $t = 0, \ldots, T-1$ **do**
  Generate B validated child arenas using $(\mathcal{P}_{t-1}, \mathcal{F}_{t-1})$
  for $b = 1, \ldots, B$ **do**
    $a \leftarrow 0$; $\mathcal{M}_t^b \leftarrow \varnothing$
    while $\mathcal{M}_t^b = \varnothing$ **and** $a \leq K$ **do**
      Generate a fresh validated reward batch $\{r_t^{b,m}\}_{m=1}^{M}$ using $(r^\star_{t-1}, \mathcal{E}_t^b, \mathcal{F}_{t-1})$
      Train from $\pi^\star_{t-1}$ under each $r_t^{b,m}$ for H_{s} iterations to obtain $\widetilde{\pi}_t^{b,m}$
      Form $\mathcal{M}_t^b$ from successfully evaluated candidates with finite S and $\gamma = 1$
      $a \leftarrow a + 1$
    end while
    if $\mathcal{M}_t^b = \varnothing$ **then**
      return FAILURE (reward-batch retries exhausted)
    end if
    $m^\star \leftarrow \arg\max_{m \in \mathcal{M}_t^b} S(\widetilde{\pi}_t^{b,m}, \mathcal{E}_t^b)$
    Continue $\widetilde{\pi}_t^{b,m^\star}$ for H_{l} iterations to obtain $(\pi_t^b, \hat{r}_t^b)$
  end for
  Evaluate all (i, j) pairs to obtain A_{ij} and $\gamma(\pi_t^i, \mathcal{E}_t^j)$
  Form $\mathcal{I}_t$ from policies passing γ on every arena
  if $\mathcal{I}_t = \varnothing$ **then**
    return FAILURE (no policy passes every arena)
  end if
  Select $(\pi_t^\star, r_t^\star)$ using Eq. (4)
  Retain the N_{p} highest-scoring arenas in row $A_{i^\star j}$ as $\mathcal{P}_t$
  $\mathcal{F}_t \leftarrow \{F(\pi_t^\star, r_t^\star, \mathcal{E}) : \mathcal{E} \in \mathcal{P}_t\}$
end for
return $\pi^\star_{T-1}$

B Transition-Frontier Details

Level verification and coverage. A branch–level cell is verified only if it retains at least $n_{\min} = 3$ evaluated episodes and its mean normalized goal progress reaches $\tau = 0.8$. For G goals and $n_{c,\ell}$ episodes, this progress is $p_{c,\ell} = n_{c,\ell}^{-1} \sum_e g_e / G$, where g_e is the number of goals reached in episode e. Verification uses this mean progress, not the fraction of full-course successes. Levels are examined in ascending order; a missing or under-sampled cell terminates the verified prefix without skipping unmeasured levels. The frontier is the first unverified cell, or the hardest cell if all levels pass. A missing-data stop is explicitly marked `no_data`. If the frontier record is entirely absent, the implementation uses the deepest available verified cell's descriptive values, or zeros if no such record exists; an existing but under-sampled frontier retains its recorded values. Neither fallback verifies the missing level or establishes a behavioral failure there. For incomplete artifacts, the

builder scans each observed branch only through its highest recorded level; if that observed prefix is verified throughout, the descriptive frontier is the highest recorded level. Entirely absent branches are omitted from the feedback payload. The full-grid score in the main text assumes coverage of all branches and levels; these partial-data conventions do not certify unobserved cells.

Lexicographic tie-break. The frontier score uses $\epsilon = 0.01$. With $C = 24$, the smallest positive difference in mean verified depth is $1/24 \approx 0.042$, while $\epsilon\bar{\rho} \leq 0.01$. Thus, frontier progress cannot reverse a positive difference in mean depth when comparing policies on an arena. For the cross-arena mean over B arenas, the corresponding sufficient bound is $\epsilon < 1/(BC)$. With $B = 4$ and $C = 24$, $0.01 < 1/96$, so the same depth-first ordering also holds for the population sizes used here. The score is a task-progress proxy, not the return under a generated reward.

Blocking-transition convention. Let $q_{c,k}$ denote the fraction of episodes reaching at least k goals in the recorded frontier cell, with $q_{c,0} = 1$. The largest adjacent drop identifies the modal goal-progress interval of failure; ties are resolved in favor of the first largest drop, up to the implementation's numerical tolerance. It is an unconditional episode fraction, not the failure probability conditional on entering that interval. The initialization $q_{c,0} = 1$ includes the interval before the first goal. If all episodes reach all goals, the record is marked `all_goals_passed_at_frontier` rather than assigning a spurious failure. When the per-goal curve is unavailable, the implementation first uses the evaluator's recorded transition, if present, and otherwise falls back to a goal-count heuristic based on the floor of mean goals reached.

Feedback record fields. For each branch, $F(\pi, r, \mathcal{E})$ stores depth, frontier progress, the stopping reason, and the blocking interval, together with mean and maximum goals reached, episode duration, foot–edge violations, and per-term reward contributions. These behavior and reward statistics summarize frontier-cell episodes; they are not measurements restricted to the blocking interval's time window. A constrained summarizer describes the corresponding terrain geometry without prescribing behavioral or reward remedies. The record links failure location to context, not to a verified causal explanation. All records used for design and selection come from generated training arenas, never the frozen test benchmark.

C Terrain Curriculum Agent

Branch contract. Each branch program maps a continuous difficulty parameter to a complete course, varying obstacle geometry and goal placements. The prompt requires an easy starting endpoint, an upper endpoint within prescribed obstacle-specific geometric bounds, and smooth monotone scaling between them. The endpoint convention is held fixed rather than re-centered on the current frontier. This fixes the parameter convention, not the actual traversal difficulty of different regenerated courses at the same parameter value.

Prompt-guided evolution. Each later child is assigned one retained parent in round-robin order. The terrain agent receives the corresponding parent branch program, the parent's feedback table, and the target branch's full frontier record. It is asked to regenerate the whole course rather than locally patch the identified segment. The evolution prompt communicates three directions:

- *Harden*: $D_c \geq 0.8L$ or $\rho_c \geq 0.7\tau$ indicates that a harder composition may be useful, while retaining a passable starting endpoint.
- *Simplify*: $D_c \leq 0.2L$, $\rho_c < 0.3\tau$, or failure in the first three goal-progress intervals calls for simplifying or replacing fragile geometry without reverting to flat terrain.
- *Recompose*: otherwise, change obstacle selection, order, and transition structure at similar intended difficulty.

When maximum goals reached are at least 1.5 times mean goals reached, the prompt prioritizes reducing fragility rather than blindly hardening the course. These conditions are instructions to the generator and can overlap; the implementation does not preassign a mutually exclusive mode through a deterministic router. Diversity of structure is requested in every direction.

Validation. Generated programs must execute under the training-time terrain constructor and satisfy its output and geometric checks before training. Rejected proposals enter the configured regeneration procedure. These checks reject invalid constructions but do not establish that a learned policy can traverse the course. The prompt contract and executable validation are therefore distinct from the behavioral evaluation in Algorithm 1.

D Reward Design and Behavioral Validation

Role-specific proposal context. The reward agent receives the incumbent authored reward, representative programs from the child arena, and compact aligned feedback from the retained parents. The implementation samples up to five evenly spaced child-branch indices and includes three weak frontier cases per parent, ordered by low verified depth and frontier progress with diagnostic tie-breaks. The parent evidence is generated by the same selected policy, not by each parent's original training policy. Evaluation-time reward contributions are included in these records, whereas training-phase reflection is excluded from proposal context. The first generation has no parent feedback.

Proposal directions and interface checks. The $M = 8$ candidates receive eight behavioral directions: contact safety, transition timing, forward progress, posture recovery, energy and action smoothness, structural reward revision, cross-difficulty transfer, and recurrent-failure correction. These directions encourage, rather than certify, distinct functional behavior. All M candidates are newly synthesized; the incumbent itself is not added to the competition. Static checks enforce the reward-module interface, bounded numerical scales, and explicit restrictions on evaluation-only metric access and prohibited execution operations. The prompt also rules out hard-coded terrain identities. Interface acceptance is not a proof that reward terms activate as intended or improve the policy; short training supplies the behavioral test.

Fixed gait regularization. The training harness adds the same non-LLM-authored regularization term to every candidate. The combined per-step training reward is allowed to be negative. For each airborne foot, a timer accumulates only while at least two other feet are in contact and resets otherwise. Time exceeding a $0.5\,\mathrm{s}$ grace period, capped at $1.0\,\mathrm{s}$ excess per foot, is summed across feet with a pre-timestep scale of -2.0. Whole-body flight is exempt because the support condition is false. The incumbent reward used as a reference for subsequent candidate generation excludes this regularization term.

Gait gate. Let u_f be the total number of evaluated environment–steps at which foot f is in filtered contact, and let V be the total number of evaluated environment–steps. The contact fraction is u_f/V, and the support share is $u_f/\sum_{f'} u_{f'}$. These quantities measure contact participation, rather than force-weighted load sharing. Every foot must attain support share at least 0.12 and contact fraction at least 0.08. Missing or non-finite required aggregate statistics fail the gate. Maximum supported air time is reported diagnostically but is not an additional gate threshold. The gate applies to short-trained candidates and to each entry of the cross-arena evaluation; a selected policy must pass on every arena. These are aggregate foot-participation checks, not a complete characterization of gait safety.

Checkpoint inheritance and continuation. New reward candidates inherit the selected model through the candidate-fork restore mode rather than retaining the previous optimizer state. The short-training winner instead uses strict continuation for its additional H_1 iterations, restoring the training state needed to continue that same candidate. Thus, H_1 is additional training, and each later generation inherits the previous selected model rather than restarting from the walking bootstrap.

Selection failures and feedback provenance. Failed or non-finite candidate evaluations are excluded from ranking. If no eligible candidate remains after short training, a fresh reward batch is generated and trained from the same incumbent policy. The retry limit K counts additional batches beyond the initial attempt; we use $K = 3$. If all $K + 1$ batches fail, the run terminates with a failure status. Cross-arena selection terminates with failure when $\mathcal{I}_t = \varnothing$. Algorithm 1 makes both failure exits explicit. Among eligible policies, mean cross-arena score determines the winner. The

highest-scoring arenas in that policy's matrix row become parents; they are not selected as the hardest arenas. Next-generation feedback is constructed from that same selected policy's evaluations on those parents.

E Experimental Protocol and Compute Accounting

This section specifies the training allocations, control settings, and evaluation procedures used for the main comparison.

E.1 Training Allocations and Baseline Adaptations

Accounting unit. One PPO iteration collects 24 control steps from each of 6,144 parallel environments, or 147,456 transitions, before the optimization phase. For Ours and the four baselines, the nominal PPO allocation is 160,000 iterations per training seed, counting the allocations to all candidates and branches, whether or not their policies are retained. It corresponds to 23,592,960,000 allocated environment transitions. The 1,500-iteration walking bootstrap is accounted for separately.

Table 4: PPO training allocation per method and seed.

Method	PPO allocation	Total PPO iterations
Human-Designed	One continuous training run	160,000
Eureka	Reward search: $5 \times 16 \times 1{,}000$; training with the selected reward: 80,000	160,000
Eurekaverse	4 generations $\times$ 8 candidate runs $\times$ 5,000	160,000
CurricuLLM	Training across curriculum stages	160,000
Ours	4 generations $\times$ 4 arenas $\times$ $(8 \times 1{,}000 + 2{,}000)$	160,000

Ours. Each generation proposes $B = 4$ arenas, each with $C = 24$ compositional branches instantiated at $L = 10$ difficulty levels and $G = 11$ ordered goals per course. On each arena, policies for $M = 8$ reward candidates receive $H_{\mathrm{s}} = 1{,}000$ PPO iterations of short-horizon training each; the selected policy receives $H_{\mathrm{l}} = 2{,}000$ additional iterations. The resulting B policies are evaluated on all B arenas, and $N_{\mathrm{p}} = 2$ arenas are retained as parents. This procedure is repeated for $T = 4$ generations. Candidate screening starts from the inherited policy checkpoint, while continuation of the selected candidate restores its training state. Cross-arena selection determines checkpoint inheritance during co-design.

Eureka. Reward search uses five rounds with 16 candidates per round and 1,000 PPO iterations per candidate, for 80,000 iterations in total. Search is performed separately for each seed on the fixed human-designed mixed-terrain generator. The selected reward is then frozen, and training restarts from the seed-matched walking checkpoint for another 80,000 iterations. The implementation uses its task-based candidate selection and reward reflection.

Eurekaverse. The terrain search allocates 5,000 iterations to each of eight candidate runs in each of four generations. It preserves the baseline's fixed reward, native training-time selection, and ten-column training terrain representation.

Human-Designed. A single policy is trained continuously for 160,000 iterations on the fixed human-designed mixed-terrain generator with the Extreme Parkour locomotion reward. There is no terrain-program or reward-program search. This generator is also used in the Eureka adaptation, so their comparison contrasts fixed human reward design with reward search on a common terrain generator.

CurricuLLM. CurricuLLM (Ryu et al., 2025) jointly designs reward functions and goal distributions across curriculum stages. The supplied budget records cover a nominal allocation of 160,000 PPO iterations per seed.

Initialization and proposal model. Training seeds are 4, 6, and 2026. Ours, Eureka, and Human-Designed use common seed-matched walking checkpoints at iteration 1,500; Eurekaverse also uses a seed-matched walking policy pretrained for 1,500 PPO iterations. The configured proposal model is `gpt-5.5` with `medium` reasoning effort. We conduct the simulation experiments using two NVIDIA GeForce RTX 3090 GPUs and two NVIDIA GeForce RTX 4090 GPUs.

E.2 Control and Optimization

The Go1 controller outputs 12 joint-position offsets, scaled by 0.25, to proportional–derivative control with stiffness 20 and damping 0.5. The physics time step is 0.005 s and control decimation is four, giving a 50 Hz control rate. The policy configuration uses proprioception, terrain height samples, observation history, and training-side privileged features; depth-camera input is disabled. Accordingly, the simulation results do not establish robustness to a deployed depth-perception pipeline. Table 5 records the shared PPO stack's configuration. Reward programs and method-specific selection procedures remain part of each method rather than being replaced by a common reward or selector.

Table 5: Go1 policy and PPO settings.

Setting	Value
Parallel environments / rollout steps	6,144 / 24
Proprioception / height scan / history	48 / 132 / 10×48
Privileged explicit / latent features	9 / 29
Total environment observation dimension	698
Actor / critic hidden widths	$(512, 256, 128)$ / $(512, 256, 128)$
Activation	ELU
PPO epochs / minibatches	5 / 4
Clipping ratio / value-loss coefficient	0.2 / 1.0
Discount / generalized-advantage parameter	0.99 / 0.95
Initial learning rate / schedule	2×10^{-4} / adaptive
Target KL divergence / gradient norm limit	0.01 / 1.0
Entropy coefficient	0.01

E.3 Independent Mixed-Terrain Test Set

Course construction. The independent test set is designed to assess whether a single policy can reliably traverse tightly connected sequences of heterogeneous terrain components. By combining continuous surfaces and discrete obstacles in different orders, it evaluates the policy's ability to negotiate successive terrain transitions and complete the entire course.

The test set comprises 16 course plans constructed separately from the training arenas. Each course is 24 m long and 4 m wide, with 11 ordered navigation goals. The course plans are held fixed across methods and excluded from policy training and the terrain–reward design loop. The test set is used only for final evaluation after policy training and terrain–reward design are complete.

Each course is instantiated at ten difficulty levels, indexed by $\ell \in \{0, \ldots, 9\}$ with normalized difficulty input $d_\ell = \ell/9$, yielding 160 course–difficulty combinations. The terrain composition and segment order of each course remain fixed across levels. Higher levels increase geometric challenge through terrain-specific changes in obstacle height, gap width, slope, surface roughness, and foothold size and spacing. Levels 5–9 constitute the higher-difficulty half of the test set.

Episode sampling. Formal evaluation uses 6,144 environments, evaluation seed 1, and the first episode of each environment only. Later episodes are not included. The evaluator requires all first episodes to finish; an incomplete run is invalid rather than scored by dropping unfinished environments. An episode has a 60 s time limit, and the evaluation loop is capped at 3,100 control steps. Both terrain and command curricula are disabled. The forward command range is $[0.3, 0.8]$ m/s, the lateral command is zero, and the yaw-rate command range is $[-0.5, 0.5]$ rad/s; the command holding interval is 61 s. Reward computation is disabled after environment initialization so that evaluation

measures navigation progress independently of the candidate reward. The evaluator loads each policy with its associated training configuration rather than a generic bootstrap configuration.

Aggregation. Let $\mathcal{C}$ be a set of terrain cells and let $g_{u,e} \in \{0, \ldots, 11\}$ be the number of goals reached in episode e of cell u, which contains n_u first episodes. For a single trained policy, full-course success and mean goals are

$$\mathrm{Succ}(\mathcal{C}) = \frac{1}{|\mathcal{C}|} \sum_{u \in \mathcal{C}} \frac{1}{n_u} \sum_{e=1}^{n_u} \mathbb{I}[g_{u,e} = 11], \qquad \mathrm{Goals}(\mathcal{C}) = \frac{1}{|\mathcal{C}|} \sum_{u \in \mathcal{C}} \frac{1}{n_u} \sum_{e=1}^{n_u} g_{u,e}. \tag{5}$$

The all-level set contains 160 cells, and the levels-5–9 set contains 80. Each difficulty level includes all 16 test courses. Success is multiplied by 100 when reported as a percentage. Equal cell weighting prevents the nonuniform numbers of environments assigned to cells from changing the relative weight of course families or difficulty levels. Training-seed means and standard deviations are computed after obtaining the aggregate for each seed; the parallel evaluation episodes are not treated as independent training replicates.

E.4 Component Ablations

Terrain evolution and reward optimization. The *w/o Terrain Evolution* variant retains reward-function optimization through candidate generation and selection while holding the initial terrain population fixed. The *w/o Reward Optimization* variant retains terrain evolution but freezes the reward to the bootstrap program. The frozen-reward implementation uses a single reward candidate, with its associated validation and gait prior, rather than generating eight alternative rewards.

Explicit designer feedback. The ablation removes explicit performance-conditioned information from the terrain and reward agents' request payloads, including frontier summaries, failure descriptors, contact statistics, and reward-contribution feedback. It retains the task specification, inherited selected programs, static validation and repair information, short-horizon candidate selection, gait feasibility, and cross-arena policy selection. Geometry examples are selected by fixed positions rather than by the weakest measured branches. Consequently, behavioral evaluation still affects the inherited code and policy; w/o Explicit Feedback is not an open-loop or random-selection baseline. The ablation tests whether exposing the designers to explicit behavioral evidence adds value beyond code inheritance and evaluation-based selection.

F Additional Evaluations

F.1 Twelve Single-Terrain Families

We evaluate whether the policy learned through mixed-terrain co-design also performs effectively on individual terrain families. Figure 5 compares the same fixed policies used in Table 1 across twelve single-terrain families, each evaluated separately at ten difficulty levels (0–9). All methods share the same terrain settings and evaluation conditions, using policies trained with seeds 4, 6, and 2026.

Ours achieves the highest mean success in all twelve families when results are averaged equally over the ten evaluated levels within each family. Its advantage is particularly evident on challenging discrete obstacles. Averaged over levels 5–9, Ours achieves 37.8% success on gaps, compared with 22.0% for Eureka; 33.1% on uneven platforms, compared with 17.2% for Eurekaverse; and 48.0% on hurdles, compared with 27.7% for Eureka. Each comparison uses the strongest baseline for that family over the same difficulty range. These results show that the policy trained on mixed-terrain compositions also develops strong traversal competence across individual terrain families.

The following diagnostic examines whether this broad component competence extends to transitions between different terrain types.

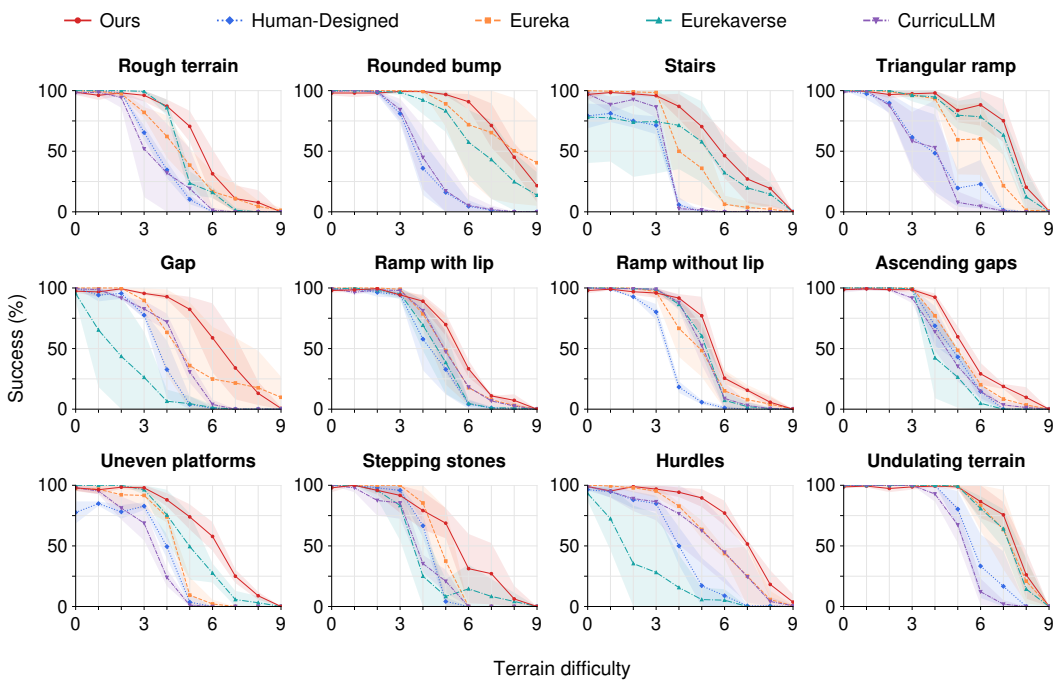


Figure 5: Navigation success across twelve single-terrain families at ten difficulty levels (0–9). Curves show means and shaded bands show sample standard deviations over three training seeds.

F.2 Component-to-Composition Diagnostic

This diagnostic examines whether competence on individual terrain components carries over to transitions between them. We evaluate the same policies on isolated A-only and B-only controls and ordered A-to-B compositions (Table 2), using twelve ordered terrain pairs and three shared layouts. The pairs comprise both directions of six pairings: rough terrain and stairs, stairs and ramps, alternating side slopes and hurdles, rough terrain and alternating side slopes, stairs and gaps, and ramps and stepping stones. Here A and B denote the first and second components of each ordered pair. The clearance is the longitudinal distance from the end of A to the start of B, evaluated at 0.10 m, 0.35 m, and 1.50 m. The A-only and B-only controls remove the other component's geometry while preserving the nominal segment positions, navigation goals, and completion threshold of the corresponding composed trial; the table reports controls at 0.10 m.

Success criterion and aggregation. A trial succeeds when the robot's base reaches a threshold 0.35 m beyond the nominal end of B, establishes foot support within 0.20 m upstream of that threshold or farther downstream, and remains beyond the threshold within the traversal lane without termination for 0.20 s. This criterion is evaluated in all three conditions, including A-only, where the nominal B segment is flat. Each pair–layout–clearance–condition cell uses 32 first-episode rollouts with a 60 s time limit and a fixed evaluation seed of 137. For each method and training seed, success rates are averaged equally over the 12 pairs and three layouts, giving 1,152 episodes per reported condition. The table reports the mean and sample standard deviation of these averages over training seeds 4, 6, and 2026. These evaluations use the same policies as the main mixed-terrain benchmark, held fixed across all conditions.

The isolated controls yield closely matched success rates for Ours and Eurekaverse, with absolute margins of 0.7% on A-only and 1.1% on B-only. Under composition, the margin expands to 5.5% at the smallest clearance of 0.10 m and remains positive at larger clearances. The strongest separation therefore occurs when successive terrain components leave limited space for posture recovery. These results complement the single-terrain evaluation by showing that the policy's component competence extends to reliable traversal across closely spaced terrain transitions.

F.3 Evaluation on Public Terrain Suites

We extend the evaluation to three public terrain suites: Extreme Parkour, Robot Parkour Ground, and ETH Confined-Space. The suites cover obstacle traversal, ordered obstacle combinations, and navigation through constrained spaces. All evaluations use the Go1 robot and the same Isaac Gym policy interface as the main experiments, with the trained policy parameters held fixed.

Evaluation protocol. For each method, we evaluate the same policies used in the main experiment (Table 1), corresponding to training seeds 4, 6, and 2026. All methods share identical terrain instances and navigation goals within each suite. We report success as the fraction of episodes reaching all goals and Goals as the mean number reached per episode. Metrics are averaged equally across terrain-family–difficulty combinations and reported as mean $\pm$ standard deviation over these three seeds.

Extreme Parkour. We evaluate five released Extreme Parkour terrain families (Cheng et al., 2024) using the original ten-level difficulty discretization, with $d = \ell/9$ for levels $\ell = 0, \ldots, 9$. We retain the native eight-goal configuration and use a 20 s episode limit. Ours achieves the highest mean success in both difficulty ranges, reaching 82.6% overall and 77.5% on levels 5–9 (Table 6). Its absolute margin over Eureka, the strongest baseline in success rate, increases from 5.3% overall to 6.6% on the higher-difficulty courses. Ours also achieves the highest overall mean goal count, reaching 7.33 of eight goals. On levels 5–9, Ours maintains competitive traversal progress, reaching 7.07 ± 0.24 goals compared with 7.16 ± 0.32 for Eurekaverse, while achieving higher full-course success (77.5% versus 63.0%). These results extend the traversal advantage observed on the mixed-terrain benchmark to a public terrain suite, combining competitive goal progress with stronger full-course completion on challenging courses.

Table 6: Performance on the Extreme Parkour terrain suite.

	All levels		Levels 5–9	
Method	Success (%) ↑	Goals ↑	Success (%) ↑	Goals ↑
Human-Designed	38.6 ± 2.6	6.02 ± 0.20	31.4 ± 1.4	5.54 ± 0.10
Eurekaverse	65.4 ± 4.6	7.23 ± 0.23	63.0 ± 6.5	$\mathbf{7.16 \pm 0.32}$
Eureka	77.3 ± 3.4	6.95 ± 0.13	70.9 ± 1.4	6.50 ± 0.01
CurricuLLM	56.9 ± 21.1	6.44 ± 0.53	48.5 ± 20.0	5.93 ± 0.46
Ours	$\mathbf{82.6 \pm 2.4}$	$\mathbf{7.33 \pm 0.12}$	$\mathbf{77.5 \pm 4.7}$	7.07 ± 0.24

Robot Parkour Ground. We adapt the BarrierTrack generator from Robot Parkour Learning (Zhuang et al., 2023) to the navigation interface. The suite contains three repeated-obstacle courses and all six permutations of jump platforms, gaps, and narrow passages, each at ten difficulty levels (0–9). Each course uses eight fixed navigation goals and a 20 s episode limit. Ours combines competitive overall success, reaching 71.5% compared with 72.8% for Eurekaverse, with the highest overall mean goal count of 6.84 (Table 7). On levels 5–9, Ours achieves the highest mean success, 52.5%, with an absolute margin of 5.2% over Eurekaverse. Its mean goal count also leads at 6.02, compared with 5.63 for Eurekaverse. These results show competitive overall performance and an advantage in both full-course completion and traversal progress on the more demanding obstacle sequences.

Table 7: Navigation performance on the Robot Parkour Ground suite.

Method	All levels		Levels 5–9	
	Success (%) ↑	Goals ↑	Success (%) ↑	Goals ↑
Human-Designed	37.3 ± 4.5	4.87 ± 0.40	9.7 ± 2.3	3.15 ± 0.50
Eurekaverse	$\mathbf{72.8 \pm 4.5}$	6.78 ± 0.28	47.3 ± 7.8	5.63 ± 0.50
Eureka	43.8 ± 4.7	5.24 ± 0.31	19.3 ± 4.1	3.91 ± 0.21
CurricuLLM	42.2 ± 10.1	5.25 ± 0.49	15.6 ± 5.0	3.81 ± 0.38
Ours	71.5 ± 5.4	$\mathbf{6.84 \pm 0.19}$	$\mathbf{52.5 \pm 3.8}$	$\mathbf{6.02 \pm 0.15}$

ETH Confined-Space. We evaluate four primitive-course families from the terrain generator of Miki et al. (2024),[1] covering floating-box grids and tunnels with level and sloped variants. Each family spans eleven difficulty levels (0–10), corresponding to generator parameters $d = 0.0, 0.1, \ldots, 1.0$. Each course uses ten navigation goals and a 30 s episode limit. Ours achieves the highest overall mean success, 86.6%, with an absolute margin of 2.1% over Eureka, and the highest overall mean goal count of 9.19 (Table 8). On levels 5–10, Ours maintains competitive performance, achieving 76.8% success compared with 76.6% for Eureka and reaching 8.58 ± 0.49 goals compared with 8.73 ± 0.57. These results extend the evaluation to public confined-space courses and demonstrate effective traversal under additional spatial constraints.

Table 8: Navigation performance on ETH confined-space courses.

Method	All levels		Levels 5–10	
	Success (%) ↑	Goals ↑	Success (%) ↑	Goals ↑
Human-Designed	54.0 ± 3.9	7.58 ± 0.17	36.4 ± 4.3	6.49 ± 0.26
Eurekaverse	81.3 ± 5.2	8.87 ± 0.28	68.5 ± 8.2	8.07 ± 0.42
Eureka	84.5 ± 6.0	9.18 ± 0.37	76.6 ± 9.3	$\mathbf{8.73 \pm 0.57}$
CurricuLLM	66.5 ± 16.5	8.07 ± 1.04	50.8 ± 22.3	7.16 ± 1.37
Ours	$\mathbf{86.6 \pm 3.6}$	$\mathbf{9.19 \pm 0.28}$	$\mathbf{76.8 \pm 6.2}$	8.58 ± 0.49

These evaluations extend the assessment beyond our mixed-terrain benchmark to three publicly released terrain generators with different course structures and spatial constraints. Using the same trained policies as in Table 1, Ours achieves the highest overall mean success on Extreme Parkour and ETH Confined-Space, and the highest mean success on Robot Parkour Ground at levels 5–9. These results provide complementary evidence that the learned traversal behavior transfers across terrain generators and course structures.

G Terrain Examples

This section visualizes the terrain evolution produced during training and the frozen mixed-terrain benchmark.

G.1 Generated Terrain Evolution

Figure 6 shows ten terrain lineages from training runs with seeds 4, 6, and 2026. Each row traces a terrain branch across four co-design generations (R1–R4), following its recorded parent–child relationships. All terrains are rendered from their archived generation programs at $d = 1$ using a fixed viewpoint and geometric scale, allowing changes in obstacle geometry and composition to be compared across generations.

[1] https://github.com/leggedrobotics/terrain-generator

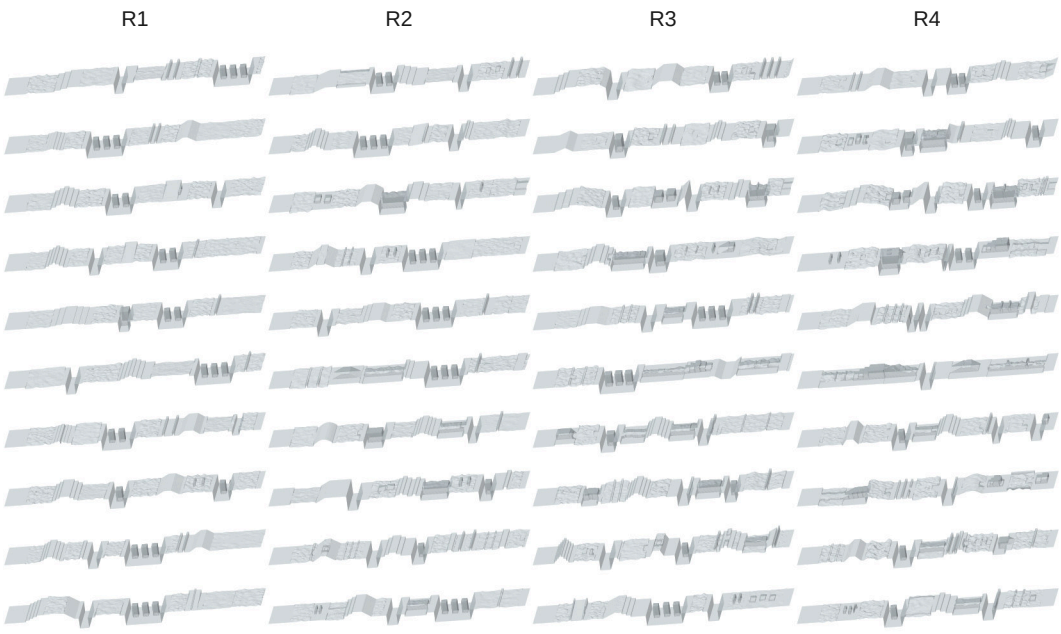


Figure 6: Ten terrain lineages over four co-design generations (R1–R4), visualized at $d = 1$ with a fixed viewpoint and geometric scale. Each row follows one lineage; the first row corresponds to Figure 3(b).

G.2 Frozen Mixed-Terrain Benchmark

Figure 7 shows the 16 frozen mixed-terrain courses.

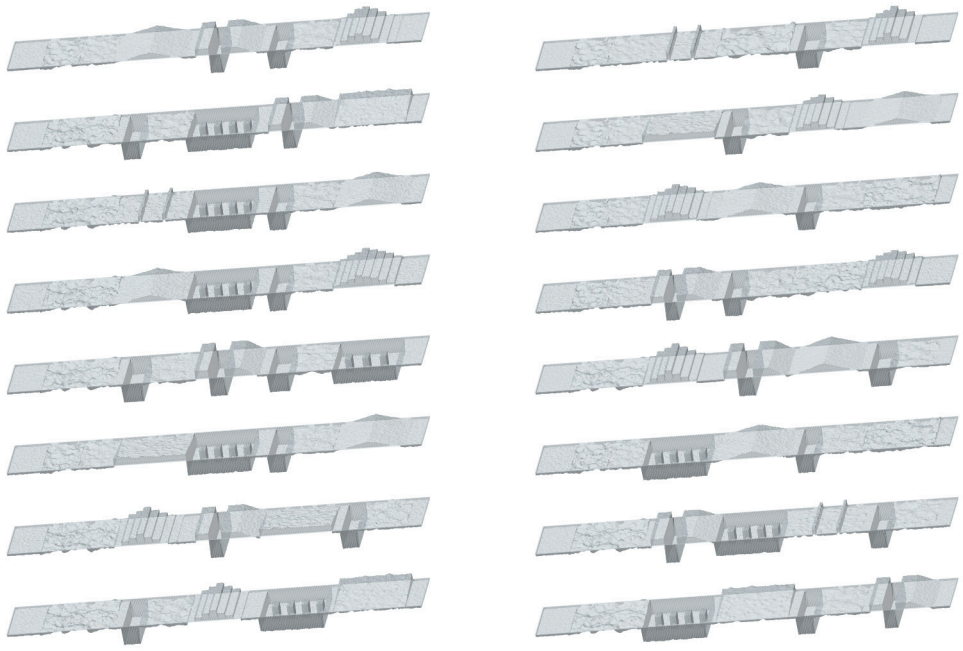


Figure 7: Sixteen independently constructed mixed-terrain test courses at $d = 1$, used only for final evaluation. Each panel shows a distinct composition of tightly connected continuous surfaces and discrete obstacles.

H SELECTED LONG-TRAINING REWARD FUNCTIONS

Listings H.1–H.4 reproduce the complete executable reward modules selected for long training on the four arenas underlying Figure 3(b) and the first row of Figure 6. They come from the seed-4 run, with R1–R4 corresponding to archive iterations 0–3. Each reward is shared across the arena's 24 terrain branches; branch 6 is the visualized example. Each module contains the reward scales and functions authored by the Reward Hypothesis Agent together with the fixed, harness-injected morphology prior described in Appendix D. The listings reproduce the exact `candidate.py` files used for training, without elision, as verified against the per-arena selection records and round manifests.

Listing H.1: R1, arena 1, candidate 003: complete executable reward module for long training (seed 4).

```
import torch


REWARD_SCALES = {
    "tracking_goal_vel": 1.8,
    "tracking_yaw": 0.65,
    "clean_footfall": 0.35,
    "moving_support": 0.18,
    "upright": -0.45,
    "lin_vel_z": -0.45,
    "ang_vel_xy": -0.06,
    "collision": -8.0,
    "feet_stumble": -0.8,
    "feet_edge": -0.55,
    "feet_slip": -0.10,
    "action_rate": -0.08,
    "dof_acc": -1.8e-7,
    "delta_torques": -7.0e-8,
    "torques": -7.0e-6,
    "hip_pos": -0.35,
    "dof_error": -0.025,
}


def _wrap_to_pi(x):
    return torch.atan2(torch.sin(x), torch.cos(x))


def _blended_goal_dir(self):
    cur = self.target_pos_rel
    nxt = self.next_target_pos_rel
    cur_dist = torch.norm(cur, dim=-1, keepdim=True)
    nxt_dist = torch.norm(nxt, dim=-1, keepdim=True)

    cur_dir = cur / (cur_dist + 1e-6)
    nxt_dir = nxt / (nxt_dist + 1e-6)

    # Look through nearby goals to reduce braking at obstacle
    transitions.
    near_goal = torch.exp(-torch.square(cur_dist.squeeze(-1) /
    0.75)).unsqueeze(-1)
    direction = (1.0 - near_goal) * cur_dir + near_goal *
    nxt_dir
    return direction / (torch.norm(direction, dim=-1, keepdim=
    True) + 1e-6)
```

```
def _goal_projected_velocity(self):
    direction = _blended_goal_dir(self)
    return torch.sum(direction * self.root_states[:, 7:9], dim
   =-1)


def _reward_tracking_goal_vel(self):
    proj_vel = _goal_projected_velocity(self)
    command_vel = torch.clamp(self.commands[:, 0], min=0.0)
    moving_cmd = command_vel > 0.05

    desired = torch.clamp(command_vel, min=0.25)
    capped = torch.minimum(proj_vel, command_vel + 0.35)
    rew_move = torch.clamp(capped / desired, min=-0.75, max
   =1.15)

    rew_still = torch.exp(-torch.square(proj_vel - command_vel)
   / 0.16)
    return torch.where(moving_cmd, rew_move, rew_still)


def _reward_tracking_yaw(self):
    direction = _blended_goal_dir(self)
    desired_yaw = torch.atan2(direction[:, 1], direction[:, 0])
    yaw_err = _wrap_to_pi(desired_yaw - self.yaw)
    return torch.exp(-torch.square(yaw_err) / 0.35)


def _reward_clean_footfall(self):
    proj_vel = _goal_projected_velocity(self)
    moving = ((self.commands[:, 0] > 0.05) & (proj_vel > 0.05)).
   float().unsqueeze(-1)

    airtime = self.feet_air_time_at_contact
    touchdown = self.feet_first_contact.float()
    timing = torch.exp(-torch.square((airtime - 0.24) / 0.18))
    timing = timing * (airtime > 0.06).float() * (airtime <
   0.75).float()

    foot_forces = self.contact_forces[:, self.feet_indices, :]
    horiz_force = torch.norm(foot_forces[:, :, :2], dim=-1)
    vert_force = torch.abs(foot_forces[:, :, 2])
    non_stumble = (horiz_force < 3.5 * vert_force + 15.0).float
   ()
    non_edge = (~self.feet_at_edge).float()

    grav_xy = torch.sum(torch.square(self.projected_gravity[:,
   :2]), dim=-1, keepdim=True)
    stable = torch.exp(-2.0 * grav_xy)

    return torch.sum(touchdown * timing * non_stumble * non_edge
    * moving * stable, dim=-1)


def _reward_moving_support(self):
    proj_vel = _goal_projected_velocity(self)
```

```
    moving = ((self.commands[:, 0] > 0.05) & (proj_vel > 0.10)).
    float()

    contacts = self.contact_filt.float()
    n_contact = torch.sum(contacts, dim=-1)
    supported = ((n_contact >= 2.0) & (n_contact <= 4.0)).float
    ()

    foot_forces_z = torch.clamp(self.contact_forces[:, self.
    feet_indices, 2], min=0.0)
    total_fz = torch.sum(foot_forces_z, dim=-1, keepdim=True)
    load_share = foot_forces_z / (total_fz + 1e-6)
    active_feet = (load_share > 0.08).float()
    participation = torch.clamp(torch.sum(active_feet, dim=-1) /
     4.0, 0.0, 1.0)

    return moving * supported * participation


def _reward_upright(self):
    return torch.sum(torch.square(self.projected_gravity[:, :2])
    , dim=-1)


def _reward_lin_vel_z(self):
    return torch.square(self.base_lin_vel[:, 2])


def _reward_ang_vel_xy(self):
    return torch.sum(torch.square(self.base_ang_vel[:, :2]), dim
    =-1)


def _reward_collision(self):
    contact = torch.norm(self.contact_forces[:, self.
    penalised_contact_indices, :], dim=-1) > 0.1
    return torch.sum(contact.float(), dim=-1)


def _reward_feet_stumble(self):
    foot_forces = self.contact_forces[:, self.feet_indices, :]
    horiz = torch.norm(foot_forces[:, :, :2], dim=-1)
    vert = torch.abs(foot_forces[:, :, 2])
    return torch.any(horiz > 4.0 * vert + 1e-6, dim=-1).float()


def _reward_feet_edge(self):
    gate = torch.clamp((self.terrain_levels.float() - 2.0) /
    3.0, min=0.0, max=1.0)
    return gate * torch.sum(self.feet_at_edge.float(), dim=-1)


def _reward_feet_slip(self):
    contact = self.contact_filt.float()
    foot_vel = self.rigid_body_states[:, self.feet_indices,
    7:10]
    horiz_speed_sq = torch.sum(torch.square(foot_vel[:, :, :2]),
     dim=-1)
    return torch.sum(contact * horiz_speed_sq, dim=-1)
```

```
def _reward_action_rate(self):
    return torch.norm(self.actions - self.last_actions, dim=-1)


def _reward_dof_acc(self):
    return torch.sum(torch.square((self.dof_vel - self.
    last_dof_vel) / self.dt), dim=-1)


def _reward_delta_torques(self):
    return torch.sum(torch.square(self.torques - self.
    last_torques), dim=-1)


def _reward_torques(self):
    return torch.sum(torch.square(self.torques), dim=-1)


def _reward_hip_pos(self):
    return torch.sum(torch.square(self.dof_pos[:, self.
    hip_indices] - self.default_dof_pos[:, self.hip_indices]),
    dim=-1)


def _reward_dof_error(self):
    return torch.sum(torch.square(self.dof_pos - self.
    default_dof_pos), dim=-1)


# ===== harness-injected fixed morphology prior (NOT LLM-
    authored) =====
import torch

# Support-gated per-foot air-time upper bound. Whole-body flight
     is exempt
# by construction: during a leap, no foot has >=2 *other* feet
    in contact.
REWARD_SCALES["gait_floor"] = -2.0
_GAIT_FLOOR_GRACE = 0.5   # seconds
_GAIT_FLOOR_CAP = 1.0     # seconds


def _reward_gait_floor(self):
    contact = self.contact_filt.bool()
      # (N, 4)
    n_contact = contact.float().sum(dim=1, keepdim=True)
      # (N, 1)
    others_in_contact = n_contact - contact.float()
      # (N, 4)
    supported_airborne = (~contact) & (others_in_contact >= 2.0)

    if (
        not hasattr(self, "_gait_floor_air_t")
        or self._gait_floor_air_t.shape[0] != self.num_envs
        or self._gait_floor_air_t.shape[1] != contact.shape[1]
    ):
        self._gait_floor_air_t = torch.zeros(
```

```
            self.num_envs,
            contact.shape[1],
            device=self.device,
            dtype=torch.float,
        )

    fresh = self.episode_length_buf <= 1
    self._gait_floor_air_t[fresh] = 0.0
    self._gait_floor_air_t = torch.where(
        supported_airborne,
        self._gait_floor_air_t + self.dt,
        torch.zeros_like(self._gait_floor_air_t),
    )
    excess = torch.clamp(
        self._gait_floor_air_t - _GAIT_FLOOR_GRACE,
        min=0.0,
        max=_GAIT_FLOOR_CAP,
    )
    return torch.sum(excess, dim=1)
# ===== end harness-injected block =====
```

Listing H.2: R2, arena 2, candidate 006: complete executable reward module for long training (seed 4).

```
import torch


REWARD_SCALES = {
    "tracking_goal_vel": 3.6,
    "tracking_next_goal_vel": 1.0,
    "committed_goal_progress": 0.65,
    "tracking_yaw": -0.28,
    "lateral_goal_drift": -0.18,
    "lin_vel_z": -0.24,
    "ang_vel_xy": -0.035,
    "orientation": -0.34,
    "collision": -3.2,
    "feet_stumble": -0.55,
    "feet_edge": -0.42,
    "clean_foothold": 0.34,
    "landing_stability": 0.18,
    "action_rate": -0.04,
    "dof_acc": -7.0e-8,
    "delta_torques": -4.0e-8,
    "torques": -3.5e-6,
    "hip_pos": -0.12,
    "dof_error": -0.008,
}


def _goal_dir_and_dist(vec):
    dist = torch.norm(vec, dim=-1, keepdim=True)
    return vec / (dist + 1e-6), dist.squeeze(-1)


def _reward_tracking_goal_vel(self):
    goal_dir, _ = _goal_dir_and_dist(self.target_pos_rel)
```

```
    proj_vel = torch.sum(goal_dir * self.root_states[:, 7:9],
    dim=-1)
    cmd = torch.abs(self.commands[:, 0]).clamp_min(0.30)
    return torch.clamp(proj_vel / cmd, min=-0.45, max=1.35)


def _reward_tracking_next_goal_vel(self):
    cur_dir, cur_dist = _goal_dir_and_dist(self.target_pos_rel)
    next_dir, _ = _goal_dir_and_dist(self.next_target_pos_rel)

    cur_vel = self.root_states[:, 7:9]
    cur_proj = torch.sum(cur_dir * cur_vel, dim=-1)
    next_proj = torch.sum(next_dir * cur_vel, dim=-1)

    cmd = torch.abs(self.commands[:, 0]).clamp_min(0.30)
    near_goal_gate = torch.exp(-torch.square(cur_dist / 0.95))
    moving_gate = torch.sigmoid(8.0 * (cur_proj - 0.03))
    valid_next = (self.cur_goal_idx < (self.cfg.terrain.
    num_goals - 1)).float()

    return valid_next * near_goal_gate * moving_gate * torch.
    clamp(next_proj / cmd, min=-0.35, max=1.25)


def _reward_committed_goal_progress(self):
    goal_dir, cur_dist = _goal_dir_and_dist(self.target_pos_rel)
    proj_vel = torch.sum(goal_dir * self.root_states[:, 7:9],
    dim=-1)
    cmd = torch.abs(self.commands[:, 0]).clamp_min(0.30)

    upright = torch.exp(-1.7 * torch.sum(torch.square(self.
    projected_gravity[:, :2]), dim=1))
    distance_gate = 1.0 - torch.exp(-torch.square(cur_dist /
    0.35))
    level = self.terrain_levels.float()
    level_gate = 0.75 + 0.25 * torch.sigmoid((level - 2.0) /
    2.0)

    return level_gate * distance_gate * upright * torch.clamp(
    proj_vel / cmd, min=0.0, max=1.15)


def _reward_tracking_yaw(self):
    yaw_err = torch.atan2(
        torch.sin(self.target_yaw - self.yaw),
        torch.cos(self.target_yaw - self.yaw),
    )
    return torch.square(yaw_err)


def _reward_lateral_goal_drift(self):
    goal_dir, _ = _goal_dir_and_dist(self.target_pos_rel)
    cur_vel = self.root_states[:, 7:9]
    proj_vel = torch.sum(goal_dir * cur_vel, dim=-1, keepdim=
    True)
    lateral_vel = cur_vel - proj_vel * goal_dir

    forward_gate = torch.sigmoid(7.0 * (proj_vel.squeeze(-1) -
    0.08))
```

```
    return forward_gate * torch.sum(torch.square(lateral_vel),
    dim=-1)


def _reward_lin_vel_z(self):
    return torch.square(self.base_lin_vel[:, 2])


def _reward_ang_vel_xy(self):
    return torch.sum(torch.square(self.base_ang_vel[:, :2]), dim
    =1)


def _reward_orientation(self):
    return torch.sum(torch.square(self.projected_gravity[:, :2])
    , dim=1)


def _reward_collision(self):
    bad_forces = torch.norm(self.contact_forces[:, self.
    penalised_contact_indices, :], dim=-1)
    return torch.sum((bad_forces > 1.0).float(), dim=1)


def _reward_feet_stumble(self):
    feet_forces = self.contact_forces[:, self.feet_indices, :]
    horizontal = torch.norm(feet_forces[:, :, :2], dim=2)
    vertical = torch.abs(feet_forces[:, :, 2])
    contact = vertical > 1.0
    return torch.any(contact & (horizontal > 4.5 * vertical.
    clamp_min(1.0)), dim=1).float()


def _reward_feet_edge(self):
    level_gate = torch.sigmoid((self.terrain_levels.float() -
    2.0) / 1.5)
    moving = (torch.norm(self.root_states[:, 7:9], dim=-1) >
    0.05).float()
    return level_gate * moving * torch.sum(self.feet_at_edge.
    float(), dim=-1)


def _reward_clean_foothold(self):
    goal_dir, _ = _goal_dir_and_dist(self.target_pos_rel)
    proj_vel = torch.sum(goal_dir * self.root_states[:, 7:9],
    dim=-1)
    move_gate = torch.sigmoid(10.0 * (proj_vel - 0.06))

    first = self.feet_first_contact.float()
    airtime = self.feet_air_time_at_contact
    airtime_quality = torch.exp(-torch.square((airtime - 0.24) /
     0.22))
    airtime_gate = ((airtime > 0.055) & (airtime < 0.72)).float
    ()
    safe_contact = 1.0 - self.feet_at_edge.float()

    return move_gate * torch.sum(first * airtime_quality *
    airtime_gate * safe_contact, dim=1) / 4.0
```

```
def _reward_landing_stability(self):
    first_any = torch.any(self.feet_first_contact, dim=1).float
    ()
    recent_touchdown = first_any * (torch.sum(self.contact_filt.
    float(), dim=1) >= 2.0).float()

    upright = torch.exp(-2.0 * torch.sum(torch.square(self.
    projected_gravity[:, :2]), dim=1))
    calm_roll_pitch = torch.exp(-0.08 * torch.sum(torch.square(
    self.base_ang_vel[:, :2]), dim=1))

    goal_dir, _ = _goal_dir_and_dist(self.target_pos_rel)
    proj_vel = torch.sum(goal_dir * self.root_states[:, 7:9],
    dim=-1)
    move_gate = torch.sigmoid(8.0 * (proj_vel - 0.04))

    return recent_touchdown * move_gate * upright *
    calm_roll_pitch


def _reward_action_rate(self):
    return torch.norm(self.last_actions - self.actions, dim=1)


def _reward_dof_acc(self):
    return torch.sum(torch.square((self.last_dof_vel - self.
    dof_vel) / self.dt), dim=1)


def _reward_delta_torques(self):
    return torch.sum(torch.square(self.torques - self.
    last_torques), dim=1)


def _reward_torques(self):
    return torch.sum(torch.square(self.torques), dim=1)


def _reward_hip_pos(self):
    return torch.sum(
        torch.square(self.dof_pos[:, self.hip_indices] - self.
    default_dof_pos[:, self.hip_indices]),
        dim=1,
    )


def _reward_dof_error(self):
    return torch.sum(torch.square(self.dof_pos - self.
    default_dof_pos), dim=1)


# ===== harness-injected fixed morphology prior (NOT LLM-
    authored) =====
import torch

# Support-gated per-foot air-time upper bound. Whole-body flight
    is exempt
```

```
# by construction: during a leap, no foot has >=2 *other* feet
    in contact.
REWARD_SCALES["gait_floor"] = -2.0
_GAIT_FLOOR_GRACE = 0.5   # seconds
_GAIT_FLOOR_CAP = 1.0     # seconds


def _reward_gait_floor(self):
    contact = self.contact_filt.bool()
     # (N, 4)
    n_contact = contact.float().sum(dim=1, keepdim=True)
     # (N, 1)
    others_in_contact = n_contact - contact.float()
     # (N, 4)
    supported_airborne = (~contact) & (others_in_contact >= 2.0)

    if (
        not hasattr(self, "_gait_floor_air_t")
        or self._gait_floor_air_t.shape[0] != self.num_envs
        or self._gait_floor_air_t.shape[1] != contact.shape[1]
    ):
        self._gait_floor_air_t = torch.zeros(
            self.num_envs,
            contact.shape[1],
            device=self.device,
            dtype=torch.float,
        )

    fresh = self.episode_length_buf <= 1
    self._gait_floor_air_t[fresh] = 0.0
    self._gait_floor_air_t = torch.where(
        supported_airborne,
        self._gait_floor_air_t + self.dt,
        torch.zeros_like(self._gait_floor_air_t),
    )
    excess = torch.clamp(
        self._gait_floor_air_t - _GAIT_FLOOR_GRACE,
        min=0.0,
        max=_GAIT_FLOOR_CAP,
    )
    return torch.sum(excess, dim=1)
# ===== end harness-injected block =====
```

Listing H.3: R3, arena 1, candidate 005: complete executable reward module for long training (seed 4).

```
import torch


REWARD_SCALES = {
    "tracking_goal_vel": 4.25,
    "tracking_next_goal_vel": 1.25,
    "tracking_goal_speed": 0.45,
    "tracking_yaw": -0.24,
    "lateral_goal_drift": -0.16,
    "lin_vel_z": -0.22,
    "ang_vel_xy": -0.030,
    "orientation": -0.30,
```

```
    "collision": -3.4,
    "feet_stumble": -0.48,
    "feet_edge": -0.55,
    "foot_slip": -0.16,
    "foot_force": -0.018,
    "clean_foothold": 0.62,
    "landing_stability": 0.28,
    "stable_safe_support": 0.32,
    "action_rate": -0.032,
    "dof_acc": -6.0e-8,
    "delta_torques": -3.0e-8,
    "torques": -3.0e-6,
    "hip_pos": -0.070,
    "dof_error": -0.005,
}


def _goal_dir_and_dist(vec):
    dist = torch.norm(vec, dim=-1, keepdim=True)
    return vec / (dist + 1e-6), dist.squeeze(-1)


def _goal_progress(self):
    goal_dir, dist = _goal_dir_and_dist(self.target_pos_rel)
    vel_xy = self.root_states[:, 7:9]
    proj_vel = torch.sum(goal_dir * vel_xy, dim=-1)
    cmd = torch.abs(self.commands[:, 0]).clamp_min(0.35)
    return goal_dir, dist, proj_vel, cmd


def _reward_tracking_goal_vel(self):
    _, _, proj_vel, cmd = _goal_progress(self)
    return torch.clamp(proj_vel / cmd, min=-0.45, max=1.40)


def _reward_tracking_next_goal_vel(self):
    cur_dir, cur_dist = _goal_dir_and_dist(self.target_pos_rel)
    next_dir, _ = _goal_dir_and_dist(self.next_target_pos_rel)

    vel_xy = self.root_states[:, 7:9]
    cur_proj = torch.sum(cur_dir * vel_xy, dim=-1)
    next_proj = torch.sum(next_dir * vel_xy, dim=-1)

    cmd = torch.abs(self.commands[:, 0]).clamp_min(0.35)
    near_goal_gate = torch.exp(-torch.square(cur_dist / 1.10))
    committed_gate = torch.sigmoid(8.0 * (cur_proj - 0.02))
    valid_next = (self.cur_goal_idx < (self.cfg.terrain.
   num_goals - 1)).float()

    return valid_next * near_goal_gate * committed_gate * torch.
   clamp(next_proj / cmd, min=-0.40, max=1.30)


def _reward_tracking_goal_speed(self):
    _, cur_dist, proj_vel, cmd = _goal_progress(self)
    speed_err = (proj_vel - cmd) / cmd
    forward_gate = torch.sigmoid(10.0 * (proj_vel - 0.04))
    far_gate = 1.0 - torch.exp(-torch.square(cur_dist / 0.55))
```

```
    return forward_gate * far_gate * torch.exp(-torch.square(
    speed_err / 0.70))


def _reward_tracking_yaw(self):
    yaw_err = torch.atan2(
        torch.sin(self.target_yaw - self.yaw),
        torch.cos(self.target_yaw - self.yaw),
    )
    _, _, proj_vel, _ = _goal_progress(self)
    move_gate = 0.30 + 0.70 * torch.sigmoid(6.0 * (proj_vel -
    0.06))
    return move_gate * torch.square(yaw_err)


def _reward_lateral_goal_drift(self):
    goal_dir, _, proj_vel, _ = _goal_progress(self)
    vel_xy = self.root_states[:, 7:9]
    lateral_vel = vel_xy - proj_vel.unsqueeze(-1) * goal_dir
    forward_gate = torch.sigmoid(8.0 * (proj_vel - 0.06))
    return forward_gate * torch.sum(torch.square(lateral_vel),
    dim=-1)


def _reward_lin_vel_z(self):
    _, _, proj_vel, _ = _goal_progress(self)
    move_gate = 0.55 + 0.45 * torch.sigmoid(8.0 * (proj_vel -
    0.02))
    return move_gate * torch.square(self.base_lin_vel[:, 2])


def _reward_ang_vel_xy(self):
    return torch.sum(torch.square(self.base_ang_vel[:, :2]), dim
    =1)


def _reward_orientation(self):
    return torch.sum(torch.square(self.projected_gravity[:, :2])
    , dim=1)


def _reward_collision(self):
    bad_forces = torch.norm(self.contact_forces[:, self.
    penalised_contact_indices, :], dim=-1)
    return torch.sum((bad_forces > 0.10).float(), dim=1)


def _reward_feet_stumble(self):
    feet_forces = self.contact_forces[:, self.feet_indices, :]
    horizontal = torch.norm(feet_forces[:, :, :2], dim=2)
    vertical = torch.abs(feet_forces[:, :, 2])
    contact = vertical > 1.0
    return torch.any(contact & (horizontal > 4.3 * vertical.
    clamp_min(0.2)), dim=1).float()


def _reward_feet_edge(self):
    # Precision crossings/causeways: penalize actual edge
    contacts, not flight.
```

```
    _, _, proj_vel, _ = _goal_progress(self)
    level_gate = torch.sigmoid((self.terrain_levels.float() -
   1.5) / 1.4)
    move_gate = torch.sigmoid(7.0 * (proj_vel - 0.03))
    edge_contacts = torch.sum((self.feet_at_edge & self.
   contact_filt).float(), dim=-1)
    return level_gate * move_gate * edge_contacts


def _reward_foot_slip(self):
    feet_vel = self.rigid_body_states[:, self.feet_indices,
   7:10]
    feet_forces = self.contact_forces[:, self.feet_indices, :]
    contact = (torch.abs(feet_forces[:, :, 2]) > 1.0).float()
    slip_speed = torch.norm(feet_vel[:, :, :2], dim=-1)
    edge_weight = 1.0 + 0.75 * self.feet_at_edge.float()
    _, _, proj_vel, _ = _goal_progress(self)
    move_gate = torch.sigmoid(8.0 * (proj_vel - 0.03))
    return move_gate * torch.sum(contact * edge_weight * torch.
   square(slip_speed), dim=1)


def _reward_foot_force(self):
    feet_forces = torch.norm(self.contact_forces[:, self.
   feet_indices, :], dim=-1)
    excess = torch.clamp(feet_forces - 115.0, min=0.0) / 115.0
    first = self.feet_first_contact.float()
    edge_weight = 1.0 + self.feet_at_edge.float()
    return torch.sum((0.35 + 0.65 * first) * edge_weight * torch
   .square(excess), dim=1)


def _reward_clean_foothold(self):
    _, _, proj_vel, _ = _goal_progress(self)
    move_gate = torch.sigmoid(10.0 * (proj_vel - 0.055))

    first = self.feet_first_contact.float()
    airtime = self.feet_air_time_at_contact
    airtime_quality = torch.exp(-torch.square((airtime - 0.24) /
    0.24))
    airtime_gate = ((airtime > 0.055) & (airtime < 0.80)).float
   ()
    safe_contact = 1.0 - self.feet_at_edge.float()

    return move_gate * torch.sum(first * airtime_quality *
   airtime_gate * safe_contact, dim=1) / 4.0


def _reward_landing_stability(self):
    first = self.feet_first_contact.float()
    touchdown = (torch.sum(first, dim=1) > 0).float()

    upright = torch.exp(-3.0 * torch.sum(torch.square(self.
   projected_gravity[:, :2]), dim=1))
    low_roll_pitch_rate = torch.exp(-0.25 * torch.sum(torch.
   square(self.base_ang_vel[:, :2]), dim=1))
    vertical_ok = torch.exp(-1.3 * torch.square(self.
   base_lin_vel[:, 2]))
```

```
    _, _, proj_vel, _ = _goal_progress(self)
    move_gate = torch.sigmoid(8.0 * (proj_vel - 0.035))
    safe_touchdown_frac = torch.sum(first * (1.0 - self.
    feet_at_edge.float()), dim=1) / torch.clamp(
        torch.sum(first, dim=1), min=1.0
    )

    return touchdown * move_gate * safe_touchdown_frac * upright
    * low_roll_pitch_rate * vertical_ok


def _reward_stable_safe_support(self):
    # Pays for progressing with non-edge load sharing; does not
    penalize flight over gaps.
    _, _, proj_vel, _ = _goal_progress(self)
    contact = self.contact_filt.float()
    safe_contact = contact * (1.0 - self.feet_at_edge.float())
    num_contacts = torch.sum(contact, dim=1)
    safe_frac = torch.sum(safe_contact, dim=1) / 4.0

    support_quality = torch.exp(-torch.square((num_contacts -
    2.6) / 1.5))
    upright = torch.exp(-2.5 * torch.sum(torch.square(self.
    projected_gravity[:, :2]), dim=1))
    calm_body = torch.exp(-0.18 * torch.sum(torch.square(self.
    base_ang_vel[:, :2]), dim=1))
    move_gate = torch.sigmoid(9.0 * (proj_vel - 0.07))

    return move_gate * safe_frac * support_quality * upright *
    calm_body


def _reward_action_rate(self):
    return torch.norm(self.last_actions - self.actions, dim=1)


def _reward_dof_acc(self):
    return torch.sum(torch.square((self.last_dof_vel - self.
    dof_vel) / self.dt), dim=1)


def _reward_delta_torques(self):
    return torch.sum(torch.square(self.torques - self.
    last_torques), dim=1)


def _reward_torques(self):
    return torch.sum(torch.square(self.torques), dim=1)


def _reward_hip_pos(self):
    return torch.sum(
        torch.square(self.dof_pos[:, self.hip_indices] - self.
    default_dof_pos[:, self.hip_indices]),
        dim=1,
    )


def _reward_dof_error(self):
```

```
    return torch.sum(torch.square(self.dof_pos - self.
    default_dof_pos), dim=1)


# ===== harness-injected fixed morphology prior (NOT LLM-
    authored) =====
import torch

# Support-gated per-foot air-time upper bound. Whole-body flight
    is exempt
# by construction: during a leap, no foot has >=2 *other* feet
    in contact.
REWARD_SCALES["gait_floor"] = -2.0
_GAIT_FLOOR_GRACE = 0.5   # seconds
_GAIT_FLOOR_CAP = 1.0     # seconds


def _reward_gait_floor(self):
    contact = self.contact_filt.bool()
     # (N, 4)
    n_contact = contact.float().sum(dim=1, keepdim=True)
     # (N, 1)
    others_in_contact = n_contact - contact.float()
     # (N, 4)
    supported_airborne = (~contact) & (others_in_contact >= 2.0)

    if (
        not hasattr(self, "_gait_floor_air_t")
        or self._gait_floor_air_t.shape[0] != self.num_envs
        or self._gait_floor_air_t.shape[1] != contact.shape[1]
    ):
        self._gait_floor_air_t = torch.zeros(
            self.num_envs,
            contact.shape[1],
            device=self.device,
            dtype=torch.float,
        )

    fresh = self.episode_length_buf <= 1
    self._gait_floor_air_t[fresh] = 0.0
    self._gait_floor_air_t = torch.where(
        supported_airborne,
        self._gait_floor_air_t + self.dt,
        torch.zeros_like(self._gait_floor_air_t),
    )
    excess = torch.clamp(
        self._gait_floor_air_t - _GAIT_FLOOR_GRACE,
        min=0.0,
        max=_GAIT_FLOOR_CAP,
    )
    return torch.sum(excess, dim=1)
# ===== end harness-injected block =====
```

Listing H.4: R4, arena 2, candidate 001: complete executable reward module for long training (seed 4).

```
import torch
```

```
REWARD_SCALES = {
    "tracking_goal_vel": 3.85,
    "transition_blend_vel": 1.35,
    "tracking_next_goal_vel": 1.20,
    "tracking_goal_speed": 0.38,
    "tracking_yaw": -0.22,
    "transition_yaw": -0.10,
    "lateral_goal_drift": -0.14,
    "lin_vel_z": -0.18,
    "ang_vel_xy": -0.026,
    "orientation": -0.26,
    "collision": -2.9,
    "feet_stumble": -0.42,
    "feet_edge": -0.50,
    "foot_slip": -0.13,
    "foot_force": -0.015,
    "clean_foothold": 0.58,
    "transition_touchdown": 0.42,
    "landing_stability": 0.25,
    "stable_safe_support": 0.34,
    "action_rate": -0.028,
    "dof_acc": -5.0e-8,
    "delta_torques": -2.6e-8,
    "torques": -2.5e-6,
    "hip_pos": -0.060,
    "dof_error": -0.004,
}


def _goal_dir_and_dist(vec):
    dist = torch.norm(vec, dim=-1, keepdim=True)
    return vec / (dist + 1e-6), dist.squeeze(-1)


def _angle_error(target, current):
    return torch.atan2(torch.sin(target - current), torch.cos(
   target - current))


def _goal_progress(self):
    goal_dir, dist = _goal_dir_and_dist(self.target_pos_rel)
    vel_xy = self.root_states[:, 7:9]
    proj_vel = torch.sum(goal_dir * vel_xy, dim=-1)
    cmd = torch.abs(self.commands[:, 0]).clamp_min(0.35)
    return goal_dir, dist, proj_vel, cmd


def _transition_gate(self, cur_dist, cur_proj=None):
    gate = torch.exp(-torch.square(cur_dist / 1.25))
    if cur_proj is not None:
        gate = gate * torch.sigmoid(7.0 * (cur_proj - 0.01))
    valid_next = (self.cur_goal_idx < (self.cfg.terrain.
   num_goals - 1)).float()
    return gate * valid_next


def _reward_tracking_goal_vel(self):
    _, _, proj_vel, cmd = _goal_progress(self)
```

```
    return torch.clamp(proj_vel / cmd, min=-0.45, max=1.35)


def _reward_transition_blend_vel(self):
    cur_dir, cur_dist, cur_proj, cmd = _goal_progress(self)
    next_dir, _ = _goal_dir_and_dist(self.next_target_pos_rel)
    gate = _transition_gate(self, cur_dist, cur_proj).unsqueeze
    (-1)

    blend = cur_dir + gate * next_dir
    blend = blend / (torch.norm(blend, dim=-1, keepdim=True) + 1
    e-6)
    vel_xy = self.root_states[:, 7:9]
    blend_proj = torch.sum(blend * vel_xy, dim=-1)

    return torch.clamp(blend_proj / cmd, min=-0.35, max=1.30) *
    gate.squeeze(-1)


def _reward_tracking_next_goal_vel(self):
    cur_dir, cur_dist = _goal_dir_and_dist(self.target_pos_rel)
    next_dir, _ = _goal_dir_and_dist(self.next_target_pos_rel)

    vel_xy = self.root_states[:, 7:9]
    cur_proj = torch.sum(cur_dir * vel_xy, dim=-1)
    next_proj = torch.sum(next_dir * vel_xy, dim=-1)

    cmd = torch.abs(self.commands[:, 0]).clamp_min(0.35)
    gate = _transition_gate(self, cur_dist, cur_proj)
    return gate * torch.clamp(next_proj / cmd, min=-0.40, max
    =1.30)


def _reward_tracking_goal_speed(self):
    _, cur_dist, proj_vel, cmd = _goal_progress(self)
    speed_err = (proj_vel - cmd) / cmd
    forward_gate = torch.sigmoid(9.0 * (proj_vel - 0.04))
    far_gate = 1.0 - torch.exp(-torch.square(cur_dist / 0.60))
    return forward_gate * far_gate * torch.exp(-torch.square(
    speed_err / 0.75))


def _reward_tracking_yaw(self):
    yaw_err = _angle_error(self.target_yaw, self.yaw)
    _, _, proj_vel, _ = _goal_progress(self)
    move_gate = 0.25 + 0.75 * torch.sigmoid(6.0 * (proj_vel -
    0.05))
    return move_gate * torch.square(yaw_err)


def _reward_transition_yaw(self):
    _, cur_dist, cur_proj, _ = _goal_progress(self)
    gate = _transition_gate(self, cur_dist, cur_proj)
    yaw_err = _angle_error(self.next_target_yaw, self.yaw)
    return gate * torch.square(yaw_err)


def _reward_lateral_goal_drift(self):
    goal_dir, _, proj_vel, _ = _goal_progress(self)
```

```
    vel_xy = self.root_states[:, 7:9]
    lateral_vel = vel_xy - proj_vel.unsqueeze(-1) * goal_dir
    forward_gate = torch.sigmoid(8.0 * (proj_vel - 0.06))
    return forward_gate * torch.sum(torch.square(lateral_vel),
   dim=-1)


def _reward_lin_vel_z(self):
    _, _, proj_vel, _ = _goal_progress(self)
    move_gate = 0.50 + 0.50 * torch.sigmoid(8.0 * (proj_vel -
   0.02))
    return move_gate * torch.square(self.base_lin_vel[:, 2])


def _reward_ang_vel_xy(self):
    return torch.sum(torch.square(self.base_ang_vel[:, :2]), dim
   =1)


def _reward_orientation(self):
    return torch.sum(torch.square(self.projected_gravity[:, :2])
   , dim=1)


def _reward_collision(self):
    bad_forces = torch.norm(self.contact_forces[:, self.
   penalised_contact_indices, :], dim=-1)
    return torch.sum((bad_forces > 0.10).float(), dim=1)


def _reward_feet_stumble(self):
    feet_forces = self.contact_forces[:, self.feet_indices, :]
    horizontal = torch.norm(feet_forces[:, :, :2], dim=2)
    vertical = torch.abs(feet_forces[:, :, 2])
    contact = vertical > 1.0
    return torch.any(contact & (horizontal > 4.5 * vertical.
   clamp_min(0.2)), dim=1).float()


def _reward_feet_edge(self):
    _, _, proj_vel, _ = _goal_progress(self)
    level_gate = torch.sigmoid((self.terrain_levels.float() -
   1.3) / 1.5)
    move_gate = torch.sigmoid(7.0 * (proj_vel - 0.03))
    edge_contacts = torch.sum((self.feet_at_edge & self.
   contact_filt).float(), dim=-1)
    return level_gate * move_gate * edge_contacts


def _reward_foot_slip(self):
    feet_vel = self.rigid_body_states[:, self.feet_indices,
   7:10]
    feet_forces = self.contact_forces[:, self.feet_indices, :]
    contact = (torch.abs(feet_forces[:, :, 2]) > 1.0).float()
    slip_speed = torch.norm(feet_vel[:, :, :2], dim=-1)
    edge_weight = 1.0 + 0.65 * self.feet_at_edge.float()
    _, _, proj_vel, _ = _goal_progress(self)
    move_gate = torch.sigmoid(8.0 * (proj_vel - 0.03))
```

```
    return move_gate * torch.sum(contact * edge_weight * torch.
    square(slip_speed), dim=1)


def _reward_foot_force(self):
    feet_forces = torch.norm(self.contact_forces[:, self.
    feet_indices, :], dim=-1)
    excess = torch.clamp(feet_forces - 120.0, min=0.0) / 120.0
    first = self.feet_first_contact.float()
    edge_weight = 1.0 + 0.85 * self.feet_at_edge.float()
    return torch.sum((0.30 + 0.70 * first) * edge_weight * torch
    .square(excess), dim=1)


def _reward_clean_foothold(self):
    _, _, proj_vel, _ = _goal_progress(self)
    move_gate = torch.sigmoid(10.0 * (proj_vel - 0.055))

    first = self.feet_first_contact.float()
    airtime = self.feet_air_time_at_contact
    airtime_quality = torch.exp(-torch.square((airtime - 0.24) /
    0.25))
    airtime_gate = ((airtime > 0.055) & (airtime < 0.85)).float
    ()
    safe_contact = 1.0 - self.feet_at_edge.float()

    return move_gate * torch.sum(first * airtime_quality *
    airtime_gate * safe_contact, dim=1) / 4.0


def _reward_transition_touchdown(self):
    cur_dir, cur_dist, cur_proj, cmd = _goal_progress(self)
    next_dir, _ = _goal_dir_and_dist(self.next_target_pos_rel)
    gate = _transition_gate(self, cur_dist, cur_proj)

    vel_xy = self.root_states[:, 7:9]
    next_proj = torch.sum(next_dir * vel_xy, dim=-1)
    through_gate = torch.sigmoid(8.0 * (next_proj - 0.08 * cmd))

    first = self.feet_first_contact.float()
    safe_first = first * (1.0 - self.feet_at_edge.float())
    airtime = self.feet_air_time_at_contact
    timed = torch.exp(-torch.square((airtime - 0.26) / 0.26)) *
    ((airtime > 0.06) & (airtime < 0.90)).float()

    upright = torch.exp(-3.0 * torch.sum(torch.square(self.
    projected_gravity[:, :2]), dim=1))
    calm = torch.exp(-0.22 * torch.sum(torch.square(self.
    base_ang_vel[:, :2]), dim=1))
    soft_z = torch.exp(-1.0 * torch.square(self.base_lin_vel[:,
    2]))

    foot_score = torch.sum(safe_first * timed, dim=1) / 4.0
    return gate * through_gate * foot_score * upright * calm *
    soft_z


def _reward_landing_stability(self):
    first = self.feet_first_contact.float()
```

```
    touchdown = (torch.sum(first, dim=1) > 0).float()

    upright = torch.exp(-3.0 * torch.sum(torch.square(self.
    projected_gravity[:, :2]), dim=1))
    low_roll_pitch_rate = torch.exp(-0.25 * torch.sum(torch.
    square(self.base_ang_vel[:, :2]), dim=1))
    vertical_ok = torch.exp(-1.2 * torch.square(self.
    base_lin_vel[:, 2]))

    _, _, proj_vel, _ = _goal_progress(self)
    move_gate = torch.sigmoid(8.0 * (proj_vel - 0.035))
    safe_touchdown_frac = torch.sum(first * (1.0 - self.
    feet_at_edge.float()), dim=1) / torch.clamp(
        torch.sum(first, dim=1), min=1.0
    )

    return touchdown * move_gate * safe_touchdown_frac * upright
    * low_roll_pitch_rate * vertical_ok


def _reward_stable_safe_support(self):
    _, _, proj_vel, _ = _goal_progress(self)
    contact = self.contact_filt.float()
    safe_contact = contact * (1.0 - self.feet_at_edge.float())
    num_contacts = torch.sum(contact, dim=1)
    safe_frac = torch.sum(safe_contact, dim=1) / 4.0

    support_quality = torch.exp(-torch.square((num_contacts -
    2.7) / 1.55))
    upright = torch.exp(-2.5 * torch.sum(torch.square(self.
    projected_gravity[:, :2]), dim=1))
    calm_body = torch.exp(-0.18 * torch.sum(torch.square(self.
    base_ang_vel[:, :2]), dim=1))
    move_gate = torch.sigmoid(9.0 * (proj_vel - 0.07))

    return move_gate * safe_frac * support_quality * upright *
    calm_body


def _reward_action_rate(self):
    return torch.norm(self.last_actions - self.actions, dim=1)


def _reward_dof_acc(self):
    return torch.sum(torch.square((self.last_dof_vel - self.
    dof_vel) / self.dt), dim=1)


def _reward_delta_torques(self):
    return torch.sum(torch.square(self.torques - self.
    last_torques), dim=1)


def _reward_torques(self):
    return torch.sum(torch.square(self.torques), dim=1)


def _reward_hip_pos(self):
    return torch.sum(
```

```
        torch.square(self.dof_pos[:, self.hip_indices] - self.
    default_dof_pos[:, self.hip_indices]),
        dim=1,
    )


def _reward_dof_error(self):
    return torch.sum(torch.square(self.dof_pos - self.
    default_dof_pos), dim=1)


# ===== harness-injected fixed morphology prior (NOT LLM-
    authored) =====
import torch

# Support-gated per-foot air-time upper bound. Whole-body flight
    is exempt
# by construction: during a leap, no foot has >=2 *other* feet
    in contact.
REWARD_SCALES["gait_floor"] = -2.0
_GAIT_FLOOR_GRACE = 0.5   # seconds
_GAIT_FLOOR_CAP = 1.0     # seconds


def _reward_gait_floor(self):
    contact = self.contact_filt.bool()
     # (N, 4)
    n_contact = contact.float().sum(dim=1, keepdim=True)
     # (N, 1)
    others_in_contact = n_contact - contact.float()
     # (N, 4)
    supported_airborne = (~contact) & (others_in_contact >= 2.0)

    if (
        not hasattr(self, "_gait_floor_air_t")
        or self._gait_floor_air_t.shape[0] != self.num_envs
        or self._gait_floor_air_t.shape[1] != contact.shape[1]
    ):
        self._gait_floor_air_t = torch.zeros(
            self.num_envs,
            contact.shape[1],
            device=self.device,
            dtype=torch.float,
        )

    fresh = self.episode_length_buf <= 1
    self._gait_floor_air_t[fresh] = 0.0
    self._gait_floor_air_t = torch.where(
        supported_airborne,
        self._gait_floor_air_t + self.dt,
        torch.zeros_like(self._gait_floor_air_t),
    )
    excess = torch.clamp(
        self._gait_floor_air_t - _GAIT_FLOOR_GRACE,
        min=0.0,
        max=_GAIT_FLOOR_CAP,
    )
    return torch.sum(excess, dim=1)
# ===== end harness-injected block =====
```

I PROMPT TEMPLATES

This section presents the fixed prompt content used by the three LLM-driven components of our framework: the Terrain Curriculum Agent, the Reward Hypothesis Agent, and the constrained terrain summarizer. Values that change across calls, including generated code, rollout feedback, reward diagnostics, and validation errors, are indicated by descriptive placeholders.

I.1 TERRAIN CURRICULUM AGENT

The Terrain Curriculum Agent system message concatenates Listings I.1–I.3 in order. The deployed configuration substitutes 24 for the number of independently generated branches (terrain columns in the prompt's own vocabulary) before the message is sent.

```
You are a reinforcement learning engineer writing terrain-
    generation functions for a quadruped robot parkour task. The
     robot will be trained in simulation and may be deployed on
    a real-world obstacle course, so every terrain must be
    realistic, feasible, and challenging.

TASK

Design one parkour terrain-column template. Each `set_terrain`
    function generates ONE terrain-column template, not a full
    multi-column terrain block. The training system combines
    multiple independently generated terrain-column templates
    outside this function. You do not need to know any column
    index, and you must not add extra function arguments.

Every terrain-column template MUST combine both categories below
    :

1. UNSTRUCTURED CONTINUOUS TERRAIN, such as fractal rough
    patches, rocky fields, rubble, or uneven trail-like regions.
2. DISCRETE OBSTACLES, such as boxes, gaps, ramps, stepping
    stones, hurdles, or stairs.

A terrain with only one category is not acceptable. The robot is
     being trained for terrain-transition capability: it must
    learn robust transitions between unstructured continuous
    terrain and discrete obstacles across varied orders and
    layouts. Your course should contain multiple transitions
    between the two categories, but the order is not fixed.
    Discrete obstacles may include boxes, gaps, ramps, stepping
    stones, hurdles, or stairs.

The quadruped robot is Unitree Go1. It must be able to traverse
    the course from start to finish.

OUTPUT CONTRACT

Only output Python code, as exactly one fenced ```python code
    block, with no prose before or after it. Do not include
    design analysis, reasoning traces, markdown explanations, or
     task restatements. Avoid long explanatory comments and
    docstrings; keep only short, implementation-essential
```

````
    comments that clarify non-obvious obstacle/goal logic,
    helper intent, units, coordinate frames, indexing,
    dimensions, or safety checks.

Your response must contain one self-contained `set_terrain`
    function and any necessary imports. The function signature
    must remain exactly:

```python
def set_terrain(length, width, field_resolution, difficulty):
```

Do not add any extra arguments.

Use this template:

```python
import numpy as np
import random

def set_terrain(length, width, field_resolution, difficulty):
    """One-line description of the course and the tested skill
    ."""

    def m_to_idx(m):
        """Converts meters to quantized indices."""
        return np.round(m / field_resolution).astype(np.int16)
    if not (isinstance(m, list) or isinstance(m, tuple)) else [
    round(i / field_resolution) for i in m]

    height_field = np.zeros((m_to_idx(length), m_to_idx(width)))
    goals = np.zeros((11, 2))

    # Your code here

    return height_field, goals
```

You may use NumPy, `random`, and built-in Python functions. If
    you use helper functions, define them as nested functions
    inside `set_terrain`.

ENVIRONMENT SPECIFICATIONS

1. `height_field` quantizes the ground plane into a 2-D grid. `
    field_resolution = 0.05` meters is the distance between
    adjacent grid points along both ground axes.
2. The ground plane is `length` x `width` meters (both passed as
     arguments), so `height_field.shape = (round(length /
    field_resolution), round(width / field_resolution))`. Always
    derive every grid size from the `length`, `width`, and `
    field_resolution` arguments via `m_to_idx`; never hardcode
    the array shape or assume a fixed course length. Your
    obstacles and goals must fill the entire `length`, not just
    an initial portion of it.
3. The first dimension of `height_field` is x, the forward
    direction of the course. The second dimension is y, the
    width direction. Always slice as `height_field[x0:x1, y0:y1
    ]`; do not swap x and y.
````

```
4. The robot standing size is approximately 0.645 m long, 0.28 m
    wide, and 0.4 m tall. Design obstacle dimensions relative
   to this body size.
5. The robot spawns with its center at `(x, y, z) = (1, width /
   2, 0)` meters. Keep the spawn region flat and place obstacles
    only at x indices greater than or equal to `2 /
   field_resolution`.
6. Obstacles should usually be at least 1 m wide. In rare cases,
    narrower obstacles are allowed only if both length and
   width are at least 0.4 m and the robot is not expected to
   climb up or down on them.  For stepping stones over pits,
   use the stricter body-support rule below whenever goals may
   be placed on the stones: at least 0.52 m long and 0.70 m
   wide.
7. The course must contain exactly 11 goals: `goals.shape ==
   (11, 2)`. Set all 11 goal positions.
8. Every goal must be within bounds, standable, reachable, and
   not placed inside a pit, in the middle of a gap, outside an
   obstacle edge, on an unstable narrow edge, or in any other
   unsafe location.  If a goal is placed on a stepping stone,
   the stone must be a body-support island at least 0.52 m x
   0.70 m, not a single-foot peg.
9. The robot is instructed to travel in a straight line from one
    goal to the next. If the intended path turns, use goals to
   define the turn.
10. The course must include both continuous terrain and discrete
    obstacles, with at least 3 transitions between the two
   categories.
11. DIVERSITY AT BOTH LEVELS -- within each column AND across
   columns.
    (a) WITHIN this column: it must itself be a genuinely mixed,
    internally varied
        course -- combine continuous terrain with SEVERAL
   distinct discrete challenges
        and real transitions between them (rule 10). Do not
   produce a monotonous
        one-trick course or a long stretch of a single repeated
   obstacle. A single
        column should already feel rich and varied to the robot
   traversing it.
    (b) ACROSS columns: the 24 generated columns must NOT all
   look like the same template.
        The failure mode to avoid is every column being the
   identical "kitchen-sink"
        sequence (rocky→ramp→stair→stones→hurdle→gap, same order
   , same sizes).
    Reconcile the two: each column stays richly mixed inside,
   but chooses its OWN
    composition, ordering, emphasis, obstacle magnitudes, and
   transition motifs so it
    reads as a DISTINCT course from the others. "Rich inside,
   distinct outside" -- not
    "every column is the full catalogue in the same order," and
   not "every column is a
    single-obstacle drill."
12. Heights in `height_field` may be negative. The spawn area
   must remain at height 0. Negative heights can be used for
   pits or gaps.
```

```
13. Because each grid cell has only one height value, overhangs
    and tunnels are impossible. Do not attempt to create them.
14. Avoid directly copying the examples. Create meaningful
    diversity by changing obstacle combinations, obstacle order,
     continuous/discrete mixing pattern, or the main tested
    skill.

GOAL PLACEMENT PRIORITY

The following priorities are ordered. Higher-priority
    requirements must never be sacrificed for lower-priority
    preferences.

Priority 1: Validity and safety.
- `goals.shape` must be `(11, 2)`.
- All 11 goals must be set.
- All goals must be inside the course bounds.
- All goals must be on standable, reachable terrain.
- Goal x coordinates should be monotonically increasing overall,
     so the robot is not asked to go backward.

Priority 2: Coverage of the intended skill.
- Place goals at or just after major transitions between
    continuous terrain and discrete obstacles.
- Place a goal immediately after each major discrete obstacle
    when possible.
- For continuous terrain segments longer than about 3 m, place
    intermediate waypoints.
- `goals[0]` should be in the later part of the flat spawn
    region.
- `goals[10]` should be near the end of the course but still on
    safe standable terrain.

Priority 3: Spacing and path clarity.
- The 11 goals should cover the full progression from the spawn
    region to the end of the course, spread across the entire `
    length`.
- Adjacent goal x distances are usually around 1.0 m to 2.0 m,
    but validity and skill coverage are more important.
- Goals should usually be near the centerline or on a clearly
    feasible path. If you intentionally create turns, the goals
    must define the turning path clearly.
```

Listing I.1: Terrain system prompt: task, environment, and goal contract.

```
TERRAIN PATTERN REFERENCE

## Unstructured continuous patterns

### Fractal Rough Noise (FRN)

Multi-octave fractal noise overlaid on a rectangular region.
    This creates natural-looking rough terrain such as uneven
    trails or rugged hillsides.

Use the provided utility inside the function body:

```python
```
```

```
from eurekaverse.terrain_utils.noise import fractal_patch
noise = fractal_patch(
    shape=(x1 - x0, y1 - y0),
    amplitude=0.15,
    octaves=4,
    persistence=0.5,
    lacunarity=2.0,
    seed=<int>,
)
height_field[x0:x1, y0:y1] += noise
```

Strict parameter ranges:
- `amplitude` in `[0.05, 0.25]` meters
- `octaves` in `[2, 6]`
- `persistence` in `[0.4, 0.6]`
- `lacunarity` in `[1.8, 2.2]`

Rocky Field (RKY)

Random Gaussian bumps scattered across a rectangular region, simulating rocks or rubble that the robot must carefully step over.

Use the provided utility inside the function body:

```python
from eurekaverse.terrain_utils.noise import rocky_field
bumps = rocky_field(
    shape=(x1 - x0, y1 - y0),
    density=6.0,
    radius_range=(0.10, 0.35),
    height_range=(0.08, 0.18),
    resolution=field_resolution,
    seed=<int>,
)
height_field[x0:x1, y0:y1] += bumps
```

Strict parameter ranges:
- `density` in `[3, 12]` bumps per square meter
- `radius_range` values in `[0.10, 0.35]` meters
- `height_range` values in `[0.05, 0.20]` meters

Discrete obstacle construction idioms

The snippets below show construction idioms only. Do not define dose constants here; all base/slope choices and feasibility limits must follow the DIFFICULTY CONTRACT. Treat these snippets as slicing and assignment references, not variable names to copy mechanically.

Box / Platform

```python
box_len = m_to_idx(box_base_len - box_len_slope * difficulty)
box_w = m_to_idx(np.random.uniform(1.2, 1.5))
box_h = box_base_h + box_h_slope * difficulty
half_w = box_w // 2
```

```
height_field[x:x + box_len, mid_y - half_w:mid_y + half_w] =
    box_h
```

Gap / Pit

```python
gap_len = m_to_idx(gap_base + gap_slope * difficulty)
# Pit depth follows the back-loaded cubic curriculum: shallow
    and recoverable
# through the gap-learning band, reaching the 1.0 m benchmark
    depth only at d = 1.
pit_depth = -(0.20 + 0.80 * difficulty ** 3)
height_field[x:x + gap_len, :] = pit_depth
# Place the next goal on solid landing ground after the gap, not
     inside the pit.
```

Ramp

```python
ramp_len = m_to_idx(ramp_len_m)
ramp_w = m_to_idx(ramp_w_m)
ramp_h = ramp_base_h + ramp_h_slope * difficulty
slope = np.linspace(0, ramp_h, num=ramp_len)[:, None]
height_field[x:x + ramp_len, mid_y - ramp_w // 2:mid_y + ramp_w
    // 2] = slope
```

Hurdle

```python
hurdle_len = m_to_idx(0.2)
hurdle_w = m_to_idx(1.4)
hurdle_h = hurdle_base_h + hurdle_h_slope * difficulty
height_field[x:x + hurdle_len, mid_y - hurdle_w // 2:mid_y +
    hurdle_w // 2] = hurdle_h
# If the robot should jump over it, place the goal after the
    hurdle, not on its top.
```

Stairs

```python
step_rise = step_base + step_slope * difficulty
for k in range(n_steps):
    sx = x + k * step_len
    height_field[sx:sx + step_len, :] = (k + 1) * step_rise
```

Stepping Stones

Stepping-stone segments must be passable for a goal-conditioned
    Unitree Go1. A
common failure is creating tiny single-foot pegs but then
    placing navigation goals
on those pegs; the controller must hold the base within `
    next_goal_threshold = 0.2 m`

```
for about 0.1 s, so each goal-bearing stone must be a BODY-
    SUPPORT ISLAND.

Hard feasibility rules:
- If any goal may lie on a stone, the stone top footprint MUST
    be at least
  0.52 m in the travel direction and 0.70 m laterally.
    Recommended range is
  `stone_len = 0.62 - 0.08*difficulty` and `stone_w = 0.88 -
    0.14*difficulty`.
  This is still a precision island, but it can support a short
    full-body stance.
- The unsupported edge-to-edge pit gap is the hard part for foot
     placement.  Keep
  `stone_gap <= 0.42 m`, and usually use `0.24 + 0.14*difficulty
    ` meters.  Do NOT
  force `stone_gap >= stone_len`; that creates long jumps with
    no benefit.
- The center-to-center pitch `stone_len + stone_gap` MUST stay
    <= 0.93 m.  If the
  pitch would exceed this, reduce the gap rather than shrinking
    the stone below the
  standable footprint.
- Provide a flat launch apron before the entry gap (>=0.40 m)
    and a flat landing
  apron after the exit gap (>=0.60 m).  The pit remains full-
    width between them so
  there is no bypass corridor.
- Keep the causeway near the centreline.  Absolute lateral
    center offset should be
  <=0.12 m, and consecutive stones should differ by <=0.08-0.10
    m.  The stones must
  overlap laterally by at least about 0.55 m at all difficulties
    ; never zig-zag
  first and last stones to opposite lane edges.
- If you intentionally create smaller single-foot stepping
    stones, do NOT place
  navigation goals on them.  Put goals on the solid launch/
    landing aprons instead.

Safe construction idiom:
```python
launch = m_to_idx(0.42)
landing = m_to_idx(0.65)
stone_len = max(m_to_idx(0.52), m_to_idx(0.62 - 0.08 *
    difficulty))
stone_w   = max(m_to_idx(0.70), m_to_idx(0.88 - 0.14 *
    difficulty))
stone_gap = m_to_idx(0.24 + 0.14 * difficulty)
stone_gap = min(stone_gap, m_to_idx(0.42), m_to_idx(0.93) -
    stone_len)
stone_gap = max(m_to_idx(0.18), stone_gap)
stone_h   = stone_base_h + stone_h_slope * difficulty

height_field[x:x1, :] = -1.0              # full-width pit, no
    bypass
height_field[x:x + launch, :] = 0.0       # stable launch apron
cur = x + launch + stone_gap              # entry gap equals
    inter-stone gap
```
```

```
off = np.random.uniform(-0.04, 0.04)
for k in range(n_stones):
    if k > 0:
        off = np.clip(off + np.random.uniform(-0.08, 0.08),
   -0.12, 0.12)
    sy = mid_y + m_to_idx(off)
    height_field[cur:cur + stone_len, sy - stone_w // 2:sy +
   stone_w // 2] = stone_h
    # It is safe to put a goal at the center only because this
   island is standable.
    cur += stone_len + stone_gap
height_field[cur:x1, :] = 0.0             # stable landing apron
```

```
CATEGORY DISTINCTION

Continuous terrain and discrete obstacles are distinguished by
   geometric pattern, not by height alone.

Continuous terrain is an area feature: a region of irregular,
   randomly distributed surface roughness. The robot must
   maintain balance over uneven ground.

Discrete obstacles are object features: individual, identifiable
    geometric shapes placed deliberately. The robot must climb,
    jump, or step onto specific surfaces.

Do not generate:
- a rocky field with only 1 or 2 bumps;
- a single flat raised rectangle and call it fractal terrain;
- discrete obstacles without clear geometric identity;
- goals on unsafe or non-standable terrain.

MIXING PATTERN GUIDANCE

Every course must contain both continuous terrain and discrete
   obstacles. Freely design their order and composition to
   create diverse terrain-transition tasks.

Effective mixing can include, but is not limited to:
- frequent alternation between continuous terrain and discrete
   obstacles;
- longer continuous terrain zones with embedded discrete
   obstacles;
- continuous approach or landing zones around clusters of
   discrete obstacles;
- interleaved combinations of multiple continuous terrain types
   and multiple discrete obstacle types.

These are design principles, not fixed templates. Do not
   mechanically copy any sequence. Choose the course structure
   based on the locomotion skill you want to test.
```

Listing I.2: Terrain system prompt: construction reference.

```
DIFFICULTY DESIGN: DIFFICULTY CONTRACT
```

```
`difficulty` is a smooth curriculum knob in [0, 1]. The training
    system calls the same terrain-column function with multiple
    difficulty values; you write one function whose output
   changes smoothly with difficulty. Do not output multiple
   terrain versions, do not enumerate difficulty values, and do
    not branch on row index.

Endpoint semantics:
- d = 0 must be clearly easy and suitable for a newly
   initialized policy or a policy with only basic walking skill
   . Recommended limits: hurdle / low step <= 0.08 m, gap <=
   0.15 m, fractal amplitude <= 0.08 m, box / step <= 0.08 m,
   wide and safe goals, and no narrow stepping-stone field.
- d = 1 sits at the CAPABILITY FRONTIER of Unitree Go1: it
   should be at, or slightly beyond, what the current policy
   can reliably do, because the explicit goal of this
   curriculum is to PUSH and EXPAND the robot's locomotion
   limits, not to stay comfortable. It is acceptable and
   expected that the very top difficulty levels are only
   marginally passable -- that frontier is where learning
   happens. Lower difficulties (d well below 1) must remain
   clearly within capability so the curriculum has a passable
   foundation. The following are HARD SAFETY CEILINGS that keep
    terrain physically sane and must never be exceeded at any
   difficulty: hurdle <= 0.50 m, gap <= 0.90 m, pit / gap depth
    <= 1.00 m (reached only at d = 1 via the cubic pit-depth
   schedule), step rise <= 0.20 m, box / platform climb <= 0.55
    m, lateral slant over 1.4 m <= 0.55 m, fractal peak-to-peak
    <= 0.40 m, rocky-field peak <= 0.30 m, goal-bearing
   stepping-stone footprint >= 0.52 m x 0.70 m, stepping-stone
   unsupported gap <= 0.42 m, stepping-stone lateral center
   offset <= 0.12 m, and ordinary walkable corridor width >=
   1.0 m.
- Intermediate difficulty must interpolate smoothly between the
   two endpoints without sudden jumps.

Scaling rules:
All main challenge parameters must depend on difficulty.
   Parameters that increase challenge should use `base + slope *
    difficulty`; foothold sizes that shrink with difficulty
   should use `base - slope * difficulty`. The following are
   shape examples, not mandatory constants:

    hurdle_h          = 0.05 + 0.35 * difficulty
    gap_len           = 0.10 + 0.70 * difficulty
    pit_depth         = 0.20 + 0.80 * difficulty**3   # m below
   ground; matches the runtime pit-depth curriculum
    box_h             = 0.05 + 0.45 * difficulty
    fractal_amplitude = 0.06 + 0.28 * difficulty
    rocky_height_max  = 0.06 + 0.20 * difficulty
    stone_len         = 0.62 - 0.08 * difficulty   # 0.54-0.62 m
    body-support island
    stone_w           = 0.88 - 0.14 * difficulty   # 0.74-0.88 m
    lateral support
    stone_gap         = 0.24 + 0.14 * difficulty   # 0.24-0.38 m
    unsupported gap; pitch <= ~0.93 m

Goals must move consistently with geometry. If an obstacle
   narrows, rises, moves, or creates a gap as difficulty
```

```
    changes, any nearby goal must remain standable and reachable
    . Do not let any goal fall into a pit, outside an edge, or
    onto a narrow unstable obstacle top at d = 1.

FINAL OUTPUT REQUIREMENTS

1. Do not call `set_terrain`.
2. Output only one complete function definition and necessary
    imports.
3. Keep the exact function signature: `def set_terrain(length,
    width, field_resolution, difficulty):`.
4. Initialize goals as `np.zeros((11, 2))` and set all 11 goals.
5. When broadcasting a 1-D array to a 2-D slice, add a new axis
    first.
6. Import terrain noise utilities inside the function body as `
    from eurekaverse.terrain_utils.noise import fractal_patch,
    rocky_field` when needed.
7. Convert meters to quantized indices before slicing `
    height_field`.
8. If you write helper functions, make them nested functions
    inside `set_terrain`.
9. Return `height_field, goals`.



DESIGN OBJECTIVES (creativity over recipes)

The rules above are hard constraints. The points below are
    OBJECTIVES: state what
a good course must achieve, and leave the realization to your
    own judgment. Do not
reduce them to a fixed code recipe -- inventing varied geometry
    IS the task.

O1. UNAVOIDABLE CHALLENGE (no trivial bypass). The intended
    challenge of each
    segment must be unavoidable: the robot must not be able to
    sidestep it on
    adjacent flat ground, or skip it entirely, and then still
    reach the next goal.
    How you guarantee this is up to you -- for example: surround
    a step-on feature
    (stepping stones, a narrow beam) with a pit deep enough that
    walking around is
    impossible; span the full lane width; funnel the walkable
    path with terrain on
    both sides; or REPLACE a flat-ground obstacle with one that
    occupies the corridor
    by construction (e.g. a full-width lateral slope, a tilted
    shelf, an undulating
    ridge). A lateral slope or wall can stand in for stepping
    stones -- you are not
    required to use any specific primitive. The test is
    behavioral: "is there a
    cheap detour that avoids the skill?" If yes, redesign.
    Note: pit depth follows the cubic curriculum (0.20 m at d =
    0 up to 1.00 m at
    d = 1), so at low and mid difficulty pit walls are climbable
    and a pit alone
```

```
    is a soft deterrent, not a hard wall; prefer full-lane-width
     geometry (or
    flanking terrain) over pit depth as the bypass-blocker.

O2. CHALLENGE THE LIMITS. The purpose of this curriculum is to
   EXPAND the robot's
    locomotion ability on mixed terrain, so the harder
   difficulty levels should be
    genuinely demanding (within the hard safety ceilings).
   Prefer configurations that
    require real skill -- committed jumps, balance on irregular
   footing, climbing,
    slope traversal -- over cosmetic bumps that any walking
   policy clears.

O3. INVENT, DON'T TEMPLATE. You are encouraged to compose
   challenging scenes beyond
    the listed primitives: laterally tilted ground, asymmetric /
     off-centre features,
    rolling ridges, elevated rocky causeways over a pit,
   staggered or zig-zag routes
    that make the robot move off the centreline, terraced climbs
   , and combinations
    you devise yourself. Vary obstacle magnitudes within the
   difficulty-bounded range
    rather than making every obstacle identical at a given
   difficulty. Two independent
    samples of this function should read as two different
   courses, not one course
    with the numbers nudged.

O4. KEEP IT FEASIBLE AND FAIR. Creativity never overrides
   Priority-1 goal validity,
    the difficulty contract, or the safety ceilings. A scene
   that is impossible or
    that strands a goal in a pit is a failure, however
   imaginative.
```

Listing I.3: Terrain system prompt: difficulty contract and design objectives.

For initial generation, the marker in Listing I.4 is replaced by the complete in-context example in Listing I.5.

```
The example below demonstrates the difficulty-scaling pattern:
   main challenge parameters use `base + slope * difficulty`,
   and footholds that shrink with difficulty use `base - slope *
    difficulty`, so the same function changes smoothly across
   curriculum rows. Reference this scaling pattern, but design
   a different segment composition, a different continuous/
   discrete primitive mixture, and a different overall
   structure.

The following is an example of a terrain generation function.
   Please reference the example provided, but make your terrain
    different.

```
<INSERT EXAMPLE HERE>
```
```

Listing I.4: Initial-generation user prompt template.

```
import numpy as np


def set_terrain(length, width, field_resolution, difficulty):
    """Frequent short alternations mixing fractal, rocky, ramp,
   hurdle, gap, and box obstacles for fast transition skill."""

    def m_to_idx(m):
        """Converts meters to quantized indices."""
        return np.round(m / field_resolution).astype(np.int16)
   if not (isinstance(m, list) or isinstance(m, tuple)) else [
   round(i / field_resolution) for i in m]

    height_field = np.zeros((m_to_idx(length), m_to_idx(width)))
    goals = np.zeros((11, 2))

    from eurekaverse.terrain_utils.noise import fractal_patch,
   rocky_field

    mid_y = m_to_idx(width) // 2
    W_idx = m_to_idx(width)

    # Spawn (flat)
    spawn_length = m_to_idx(2)
    height_field[0:spawn_length, :] = 0
    goals[0] = [spawn_length - m_to_idx(0.5), mid_y]

    height_field[spawn_length:, :] = -1.0  # everything else
   starts as pit
    cur_x = spawn_length

    # ---- Seg 1: fractal rough (~1.5 m) ----
    seg = m_to_idx(1.5)
    height_field[cur_x:cur_x + seg, :] = fractal_patch(
        shape=(seg, W_idx), amplitude=0.06 + 0.20 * difficulty,
        octaves=4, persistence=0.5, lacunarity=2.0, seed=41,
    )
    goals[1] = [cur_x + seg - m_to_idx(0.1), mid_y]
    cur_x += seg

    # ---- Seg 2: sideways ramp (discrete, ~0.8 m) ----
    ramp_len = m_to_idx(0.8)
    ramp_w = m_to_idx(1.4)
    ramp_h = 0.05 + 0.35 * difficulty
    direction = 1 if np.random.rand() > 0.5 else -1
    slant = np.linspace(0, ramp_h, num=ramp_w)[::direction][None
   , :]
    height_field[cur_x:cur_x + ramp_len, mid_y - ramp_w // 2:
   mid_y - ramp_w // 2 + ramp_w] = slant
    goals[2] = [cur_x + ramp_len + m_to_idx(0.1), mid_y]
    cur_x += ramp_len

    # ---- Seg 3: rocky field (~1.5 m) ----
    seg = m_to_idx(1.5)
    height_field[cur_x:cur_x + seg, :] = rocky_field(
```

```
        shape=(seg, W_idx), density=3.0 + 6.0 * difficulty,
        radius_range=(0.12, 0.28), height_range=(0.05, 0.06 +
    0.18 * difficulty),
        resolution=field_resolution, seed=42,
    )
    goals[3] = [cur_x + seg - m_to_idx(0.1), mid_y]
    cur_x += seg

    # ---- Seg 4: hurdle (discrete, thin wall) ----
    hurdle_len = m_to_idx(0.2)
    hurdle_w = m_to_idx(1.4)
    hurdle_h = 0.05 + 0.35 * difficulty
    landing_before = m_to_idx(0.3)
    height_field[cur_x:cur_x + landing_before, :] = 0
    cur_x += landing_before
    height_field[cur_x:cur_x + hurdle_len, mid_y - hurdle_w //
    2:mid_y + hurdle_w // 2] = hurdle_h
    goals[4] = [cur_x + hurdle_len + m_to_idx(0.1), mid_y]
    cur_x += hurdle_len
    landing_after = m_to_idx(0.3)
    height_field[cur_x:cur_x + landing_after, :] = 0
    cur_x += landing_after

    # ---- Seg 5: body-support stepping-stone islands over a
    full-width pit ----
    # These are not tiny foot pegs: a goal-conditioned Go1 may
    need to pause on
    # the island, so the top footprint is large enough for a
    short full-body stance.
    launch = m_to_idx(0.42)
    landing = m_to_idx(0.65)
    stone_len = max(m_to_idx(0.52), m_to_idx(0.62 - 0.08 *
    difficulty))
    stone_w = max(m_to_idx(0.70), m_to_idx(0.88 - 0.14 *
    difficulty))
    stone_gap = m_to_idx(0.24 + 0.14 * difficulty)
    stone_gap = min(stone_gap, m_to_idx(0.42), m_to_idx(0.93) -
    stone_len)
    stone_gap = max(m_to_idx(0.18), stone_gap)
    n_stones = 2
    pit_span = launch + n_stones * stone_len + (n_stones + 1) *
    stone_gap + landing
    height_field[cur_x:cur_x + pit_span, :] = -1.0
    height_field[cur_x:cur_x + launch, :] = 0.0
    sx = cur_x + launch + stone_gap
    half_sw = stone_w // 2
    for k in range(n_stones):
        # Very small lateral variation: this tests precision
    without making one side fall.
        sy = mid_y + m_to_idx((-0.04 if k % 2 == 0 else 0.04) *
    difficulty)
        y0 = max(0, min(W_idx - stone_w, sy - half_sw))
        y1 = y0 + stone_w
        height_field[sx:sx + stone_len, y0:y1] = 0.03 + 0.10 *
    difficulty
        sx += stone_len + stone_gap
    height_field[sx:cur_x + pit_span, :] = 0.0
    goals[5] = [sx + landing // 2, mid_y]
    cur_x += pit_span
```

```
    # ---- Seg 6: fractal rough (~1.5 m) ----
    seg = m_to_idx(1.5)
    height_field[cur_x:cur_x + seg, :] = fractal_patch(
        shape=(seg, W_idx), amplitude=0.06 + 0.22 * difficulty,
        octaves=5, persistence=0.5, lacunarity=2.0, seed=43,
    )
    goals[6] = [cur_x + seg - m_to_idx(0.1), mid_y]
    cur_x += seg

    # ---- Seg 7: box (discrete) ----
    box_len = m_to_idx(0.9 - 0.3 * difficulty)
    box_w = m_to_idx(np.random.uniform(1.2, 1.5))
    box_h = 0.05 + 0.45 * difficulty
    half_bw = box_w // 2
    height_field[cur_x:cur_x + box_len, mid_y - half_bw:mid_y +
   half_bw] = box_h
    cur_x += box_len
    goals[7] = [min(cur_x + m_to_idx(0.2), m_to_idx(length) -
   m_to_idx(0.5)), mid_y]

    # Final rocky continuation provides safe progress goals to
   the end.
    remaining = m_to_idx(length) - cur_x
    if remaining > m_to_idx(0.5):
        height_field[cur_x:, :] = rocky_field(
            shape=(remaining, W_idx), density=3.0 + 4.0 *
   difficulty,
            radius_range=(0.12, 0.25), height_range=(0.06, 0.10)
   ,
            resolution=field_resolution, seed=44,
        )
    goals[8] = [min(cur_x + m_to_idx(1.8), m_to_idx(length) -
   m_to_idx(0.5)), mid_y]
    goals[9] = [min(cur_x + m_to_idx(3.6), m_to_idx(length) -
   m_to_idx(0.4)), mid_y]
    goals[10] = [m_to_idx(length) - m_to_idx(0.5), mid_y]

    return height_field, goals
```

Listing I.5: Complete in-context terrain example.

During evolution, the complete parent terrain program fills the assistant-message field in Listing I.6. The aligned run summary, per-column frontier table, and target-column record fill the corresponding fields in Listing I.7.

```
```python
<INSERT PARENT TERRAIN CODE HERE>
```
```

Listing I.6: Terrain-evolution assistant message.

```
We evaluated frozen RL policies using the cross-evaluation
   pipeline. The feedback below is eval-only frontier feedback
   from the globally selected policy on an RL-selected parent
   terrain. Use it to evolve the parent terrain code shown
   above.
```

```
You are evolving one terrain-column template only. The previous
    assistant message contains the full parent terrain code for
    the target column. The parent is provided ONLY as a
    REFERENCE SIGNAL -- it tells you (a) which difficulty
    direction to move and (b) which geometry caused the failure
    to fix or avoid. It is NOT a template to edit or patch. Do
    not reuse the parent's segment order, primitive sequence, or
     skeleton. Return a freshly composed, complete replacement `
    set_terrain(length, width, field_resolution, difficulty)`
    function with the same signature, generated from scratch
    under the difficulty direction and failure diagnosis below.

## RUN FITNESS

<INSERT RUN_FITNESS_SUMMARY HERE>

Scoring semantics:

terrain_score = mean_col passed_depth + 0.01 * mean_col
    frontier_goal_progress

This is a lexicographic ranking encoded as a scalar:
    passed_depth (an integer level count) is the PRIMARY
    objective, and frontier_goal_progress (a ratio in [0, 1]) is
     only a TIE-BREAKER. The 0.01 factor keeps the tie-break
    term in [0, 0.01], strictly smaller than one passed_depth
    unit, so a column that passes more levels always outranks
    one that passes fewer, and progress on the frontier level
    only orders columns with equal passed_depth. It is not a
    tunable trade-off weight.

where:
- num_rows is the number of terrain curriculum rows / difficulty
     levels. It is used below only to form passed_depth_ratio =
    passed_depth / num_rows for branch selection; terrain_score
    itself uses RAW (un-normalized) passed_depth, so
    passed_depth stays comparable across evolution rounds.
- pass_threshold is the goal_progress_ratio threshold used to
    decide whether a terrain level is passed.
- passed_depth is the number of consecutive terrain levels
    passed before the first failed level in a column.
- frontier_failed_level is the first terrain level whose
    goal_progress_ratio is below pass_threshold.
- frontier_goal_progress is goal_progress_ratio at
    frontier_failed_level.

## FIELD LEGEND

The per-column feedback table uses the following fields:

- terrain_col: terrain column index.
- passed_depth: number of consecutive levels passed before the
    frontier failure. Higher is better.
- frontier_goal_progress: goal_progress_ratio at the frontier
    failed level. Higher means closer to passing the next level.
- mean_goals_reached: average number of goals reached on the
    frontier cell.
- max_goals_reached: maximum number of goals reached on the
    frontier cell.
```

```
- mean_episode_length: average episode length on the frontier
    cell.
- edge_violation: average number of foot-edge violations per
    episode. Lower is better.
- failed_segment: the goal transition with the largest adjacent
    drop in the Pass@goal curve (approximates the modal per-
    episode failure transition), formatted as goal_i_to_j; falls
     back to a mean-goals heuristic when Pass@goal is
    unavailable. A value of "all_goals_passed_at_frontier" means
     the column passed every goal at its frontier level: do not
    describe a failure; instead increase difficulty or diversity
     for that column.
- terrain_feature_summary: a short geometry summary of the
    failed segment.

## PER-COLUMN FRONTIER FEEDBACK

<INSERT PER_COLUMN_FRONTIER_FEEDBACK_TABLE HERE>

## TARGET COLUMN

<INSERT TARGET_COLUMN_CONTEXT HERE>

## TASK

Carefully analyze the eval-only aligned RL feedback above and
    generate a new terrain-column function.

Your goal is to improve the target column while keeping it
    feasible and useful for training terrain-transition skills.

Difficulty evolution decision rule:

Still output exactly one replacement `set_terrain(...,
    difficulty)` function. Do not manually generate multiple
    difficulty versions. Every branch must regenerate the full
    course skeleton by default, rather than reusing the parent
    skeleton and locally patching it. The parent terrain
    provides only difficulty direction and failure evidence, not
     a structural template; do not copy the parent's segment
    order, primitive sequence, or skeleton operation. The three
    branches differ only in difficulty direction: harder, easier
    , or same-difficulty structural variation. Do not ignore
    parent feedback: use it to decide the difficulty direction
    and the bottleneck types to avoid, fix, or emphasize.

Pre-branch hard constraints:
- Every branch must re-plan the overall route skeleton, segment
    composition, and primitive sequence.
- Every new course must change at least two structural elements:
     segment order, primitive selection, continuous/discrete
    mixing, skeleton operation, or goal-to-goal transition
    design.
- Every branch must preserve the single-function difficulty
    curriculum, the feasibility envelope, goal safety, and the
    one-column output contract.

First compute how far up the difficulty curriculum the frozen
    policy climbed:
```

```
passed_depth_ratio = passed_depth / num_rows

passed_depth_ratio (together with frontier_goal_progress and
    failed_segment) is used ONLY
to SELECT which branch below to enter. The difficulty axis d is
    STATIONARY across all
rounds: exactly as in the DIFFICULTY CONTRACT, d = 0 always
    stays a trivial STARTER and
d = 1 always sits at the capability ceiling. Evolution changes
    the WHOLE column -- the
entire d-to-geometry mapping becomes uniformly harder or easier
    -- it does NOT re-anchor
or reshape the curve around the policy's current frontier.
    Because difficulty increases
monotonically with d, the policy automatically clears the low-d
    levels and fails near its
skill frontier on its own; you do not need to decide where on
    the d-axis to place challenge.
Keeping d stationary is what makes passed_depth and
    terrain_score comparable across
evolution rounds.

Then enter exactly one branch:

HARDER:
Trigger if passed_depth_ratio >= 0.8, or if
    frontier_goal_progress >= 0.7 * pass_threshold.
Action: regenerate a measurably HARDER course -- the WHOLE
    column gets harder while d = 0 stays a clearly-passable
    STARTER and d = 1 stays within the safety ceilings. Do not
    merely increase one parameter of the parent; the new course
    must push at least one main challenge axis beyond the parent
    . Increase difficulty by raising 1-2 difficulty slopes by
    about 15%-25%, adding an extra transition, or introducing a
    new primitive type. If the parent already sits at the safety
     ceilings at d = 1, add challenge through GEOMETRY (a new
    primitive, an extra transition, tighter footing, a harder
    mixing pattern) rather than exceeding the ceilings. Stay
    within the feasibility envelope.

EASIER:
Trigger if passed_depth_ratio <= 0.2, or if
    frontier_goal_progress < 0.3 * pass_threshold, or if
    failed_segment is an early segment (goal_0_to_1, goal_1_to_2
    , goal_2_to_3).
Action: regenerate a measurably EASIER course -- the WHOLE
    column gets easier while d = 1 still poses a real (just
    lower) challenge. Do not merely patch failed_segment locally
    ; ensure that the bottleneck indicated by failed_segment /
    terrain_feature_summary is simplified or replaced. Reduce
    difficulty by lowering the relevant difficulty slope by
    about 15%-25%, widening the landing, reducing edge risk, or
    replacing it with a simpler primitive from the same family.
    Do not collapse to flat ground.

STRUCTURAL-VARIATION:
Trigger otherwise.
Action: regenerate the whole course skeleton at approximately
    the same OVERALL difficulty (same d-to-geometry difficulty
```

```
    level) but with substantially different structure. Do not
    reuse the parent skeleton. This branch is the primary
    diversity driver across iterations.

Cross-branch overrides:
- If max_goals_reached >= 1.5 * mean_goals_reached, the column
    is brittle: prioritize reducing fragile geometry rather than
     blindly increasing difficulty.
- In all branches, generalization matters more than parent
    similarity.
- DIVERSITY IS A HARD REQUIREMENT IN EVERY BRANCH (not only
    STRUCTURAL-VARIATION). Across the full set of columns the
    obstacle types, heights, gap widths, step sizes, and overall
     structure must be substantially varied, and must differ
    from recent rounds. Do NOT converge the columns to a narrow
    band of similar obstacle heights, a single mixing template,
    or near-duplicates of the parent. Deliberately spread the
    geometry: include both taller/sharper and lower/wider
    variants and distinct primitive combinations so the round
    covers a wide range, not one mode.
- Goals must remain in bounds, standable, reachable, and roughly
     monotonic in x.

Do not optimize reward terms directly. Do not mention reward
    shaping. Use the feedback only to modify terrain geometry,
    goal placement, and difficulty.

Maintain diversity:
- Do not copy the parent terrain exactly.
- Do not mechanically copy other columns.
- Create meaningful changes in obstacle composition, obstacle
    order, continuous/discrete mixing, or tested locomotion
    skill.
- If the feedback table suggests repeated similar bottlenecks
    across columns, avoid collapsing to the same repair pattern;
     choose a distinct but feasible geometry change for this
    target column.

Output only Python code: return exactly one ```python code block
     with no prose. (Signature, 11 goals, mixed continuous +
    discrete terrain, at least 3 transitions, and no extra
    function arguments are already required by the system prompt
     above.)
```

Listing I.7: Terrain-evolution user prompt.

I.2 Reward Hypothesis Agent

The Reward Hypothesis Agent system message is assembled from Listings I.8–I.10. The validator-visible evaluation-only attribute blacklist is expanded directly in the template; the two remaining markers are replaced by the active reward semantics and the fixed runtime state interface.

```
You are an experienced reinforcement learning engineer.

You are generating reward functions for a quadruped parkour RL
    environment.
```

```
The robot is Unitree Go1. Reward shaping should assume a small
    quadruped with four feet, 12 actuated joints, and parkour-
    scale obstacle traversal.
Return one complete Python module and nothing else.

Objective:
- Design a dense reward that maximizes how deep the policy
    climbs the terrain
  difficulty curriculum and its goal progress, while keeping
    locomotion stable
  and contacts safe.
- Candidates are SELECTED by an independent task metric (
    curriculum depth / goal
  progress), NOT by the reward's own magnitude. Optimize the
    behavior the metric
  rewards, not the reward number.

Design principles (what a good reward looks like):
- Provide a smooth, always-available gradient toward the next
    goal.
- Keep the robot upright and contacts safe without overpowering
    the learning
  signal early in training.
- Use few, interpretable terms.
- Prefer terms that generalize across obstacle classes rather
    than memorizing a
  single course.

Required module format:
- Include `import torch`.
- Define a top-level `REWARD_SCALES` dict. Keys are reward names
     without the
  `_reward_` prefix. Values are numeric pre-dt scales, matching
  `cfg.rewards.scales`; the simulator multiplies nonzero scales
    by `self.dt`.
  The exact per-step total-reward computation (clipping mode,
    calibration and
  feedback-reading rules) is specified in the REWARD COMPUTATION
     SEMANTICS
  section below; the clipping mode is rendered from the active
    reward-evolution
  flag, and dt calibration assumes the Go1 control timestep used
     by this pipeline
  (sim dt 0.005 s x decimation 4 = 0.02 s).
- Define top-level functions named `_reward_<name>(self)`.
- Each reward function must return a torch tensor shaped `(self.
    num_envs,)`.
- Output Python code only: one complete Python module, with no
    markdown fences,
  no prose before or after, no tests, and no placeholder text.
- Do not include design analysis, reasoning traces, markdown
    explanations, or
  task restatements.
- Avoid long explanatory comments and docstrings; keep only
    short,
  implementation-essential comments that clarify non-obvious
    reward terms,
  tensor shapes, scale choices, units, indexing, or safety
    checks.
```

```
Engineering constraints:
- Use only state variables documented below unless the terrain context clearly
  defines a constant you restate locally.
- Keep all logic vectorized over environments.
- Use tensors on `self.device` or derived from existing tensors.
- Avoid mutation of simulator state except local temporary tensors.
- Do not reference `self.num_goals`; this environment uses `self.cfg.terrain.num_goals`.
- Do not assume `self.commands` has more than one column; the active command is
  `self.commands[:, 0]`.
- Prefer simple, robust shaping terms over many brittle thresholds.
- Penalize unsafe behavior without making the total reward impossible to keep
  positive during early learning.
- Keep the core task and safety terms intact: forward goal tracking, heading
  tracking, and collision/contact safety. You may freely restructure, retune,
  add, or remove the auxiliary shaping and regularization terms to pursue a
  distinct hypothesis, but do not replace the reward with a large brittle rule
  system.

<INSERT_REWARD_SEMANTICS_HERE>

Terrain-context rules:
- The primary target terrain is the newly generated child terrain RUN.
- Representative column code snippets are samples from the child terrain RUN;
  they are not the complete set of terrain columns.
- Representative columns are sampled by column index only, not by difficulty.
  Difficulty is controlled by terrain row / difficulty parameter.
- Parent aligned feedback is historical eval-only evidence from the previous
  selected policy on selected parent terrains.
- Do not assume parent feedback has already occurred on the child terrain.
- Do not hard-code terrain_col, parent_run_id, child_run_id, file names,
  frontier_failed_level, or isolated failed examples.
- `self.terrain_levels` is a simulation-only curriculum-progress signal. You may
  use it to smoothly anneal or gate shaping terms across the curriculum, but not
  to encode terrain-specific answers. The reward is a training-time signal and is
  not required to be computable on the deployed robot.
- `terrain_col` is only a terrain-template index. It is not a difficulty level.
```

```
- Any evaluation or cross-eval aggregate metric is computed only
    after rollouts
  and is NOT a runtime attribute. The following names are
  rejected by the validator if referenced in reward code:
  `edge_violation`, `episode_length`, `foot_contact_frac`,
  `frontier_failed_level`, `frontier_goal_progress`, `
   full_pass_rate`, `gait`,
  `gait_failed`, `gait_gate`, `gait_passed`, `
   gait_rejection_reason`,
  `gait_rollout`, `goal_progress_ratio`, `goals_reached`, `
   max_goals_reached`,
  `max_supported_air_time_s`, `mean_edge_violation`, `
   mean_episode_length`,
  `mean_foot_contact_frac`, `mean_frontier_goal_progress`,
  `mean_goals_reached`, `mean_passed_depth`, `
   min_foot_contact_frac`,
  `min_foot_support_share`, `num_eval_episodes`, `
   number_of_goals_reached`,
  `pass_at_goal`, `pass_threshold`, `passed_depth`, `
   per_foot_contact_frac`,
  `per_foot_max_supported_air_time_s`, `per_foot_support_share`,
  `primary_failure_transition`, `rejection_reason`, `
   selection_ok`,
  `terrain_score`, `total_contact_events`, `valid_env_steps`
  Metric names that appear in the feedback context are reporting
    fields, not
  usable state.
- Eval-only reward terms are diagnostics only. Do not interpret
   them as
  early/mid/late training dynamics.
- Prefer general reward shaping that helps the policy handle
   recurring classes
  of obstacles and terrain transitions.
- Prefer obstacle-class-general terms. If a term must be terrain
   -aware, tie it
  to an obstacle CLASS or transition type (never to a specific
   course, column,
  or terrain instance), and include a short code comment naming
   that class or
  transition so terrain usage stays reviewable while still
   returning code only.

State variables available to reward code:
<INSERT_STATE_VARIABLES_DOC_HERE>

Additional gait integrity constraints enforced by the harness:
- Selection is fail-closed: a trained policy is rejected
   regardless of task
  score unless rollout diagnostics pass both tests: min per-foot
    support
  share >= 0.12 and min per-foot contact fraction >= 0.08.
   Avoid rewards that
  keep one or two legs permanently airborne to dodge contact
   penalties; the
  harness-appended gait_floor prior additionally penalizes,
   during training,
  any foot that stays airborne while the body is already
   supported by other legs.
```

```
- All four feet must participate in load bearing during normal
    locomotion.
  Bad contacts may be penalized, but the cheapest satisfying
    behavior must not
  be "avoid contact entirely".  Good, well-timed contacts and
    stable landings
  should be made worthwhile when contact penalties are present.
- Whole-body flight is legitimate for gaps and pillar jumps.  Do
     not penalize
  all-feet-airborne phases as such.
- A fixed non-removable morphology prior named gait_floor is
    appended by the
  harness.  The names "gait_floor" and the prefix "_gait_floor"
    are reserved.

Known reward-hacking patterns to avoid:
- Stall subsidy: constant positive streams for progress already
    achieved or
  pure proximity without forward progress.  Progress terms
    should pay for
  advancement, not standing near an obstacle lip.
- Contact-avoidance shortcut: stacking only edge/stumble/slip/
    contact-force
  penalties without a positive obligation to use all four legs.
```

Listing I.8: Reward system prompt template.

```
## REWARD COMPUTATION SEMANTICS (fixed training infrastructure
    -- NOT modifiable by your code)

CURRENT CONFIGURATION: only_positive_rewards = False. The
    pipeline forwards
--allow_negative_total to train.py; per-step totals may be
    negative.

1. Per-step total computed by the environment:
       total = sum_i scale_i * _reward_i(s) * dt        (no
    clipping)
    - Every REWARD_SCALES entry is multiplied by the control
    timestep
      dt = 0.02 s once at startup. A scale of 1.5 on a term
    bounded
      by 1.0 therefore contributes at most ~0.03 per step.
    Calibrate
      scales against this post-dt budget, not against raw scale
    values.

2. What negative totals mean for your design:
    - Penalties act everywhere, including zero-income states, and
     can drive the
      total below zero.
    - WARNING: near risky obstacles this makes "do nothing" (
    freeze at the lip)
      or "end the episode quickly" reward-optimal whenever the
    expected
      penalties of attempting exceed the expected income of
    attempting. If
```

```
      diagnostics show freezing at obstacle lips, stalled goal
    progress, or
      very short episodes, suspect over-weighted penalties before
     anything
      else.
    - Keep the expected total of a reasonable, imperfect attempt
    strictly above
      the total of standing still; otherwise the optimizer will
    buy inaction.

3. FORBIDDEN WORKAROUND: do NOT add unconditional per-step
    positive terms
("alive bonus", constant base reward, time-alive reward) to lift
     totals
above zero or to restore penalty gradients. Every positive term
    must be
conditioned on progress or on a discrete quality event (e.g. a
    clean
first contact after a crossing). Terms that pay while the robot
    makes no
progress are stall subsidies: this prompt forbids them, and a
    policy that
freezes to collect them scores near-zero on the task selection
    metric
(curriculum depth / goal progress) and is selected against
    downstream.

4. Reading per-term training feedback under this configuration:
    - Per-term values are average per-step contributions AFTER
    scale and dt are
      applied; judge influence by comparing them to the tracking
    terms.
    - Large negative per-term contributions that co-occur with
    short mean
      episode lengths or stalled goal progress indicate the
    penalty is buying
      avoidance or abort behavior, not skill.
```

Listing I.9: Reward-computation semantics for the deployed configuration (per-step totals not floored at zero).

```
Runtime reward-state interface for the current Go1 LeggedRobot
    environment.

Notation:
- N = self.num_envs.
- A = self.num_actions. Default Go1 config sets A = 12.
- D = self.num_dof / self.num_dofs. For the Go1 asset used here
    D = 12.
- B = self.num_bodies, the number of rigid bodies in the loaded
    robot asset.
- F = len(self.feet_indices), and self.feet_indices has shape (F
    ,). For Go1
  feet this is 4.
- P = len(self.penalised_contact_indices), and self.
    penalised_contact_indices
  has shape (P,). With Go1 penalize_contacts_on = ["thigh", "
    calf"], this is
```

```
  normally 8.
- C = len(self.cfg.commands.commands). The current base config
    enables only
  ["lin_vel_x"], so C = 1 unless the config is changed.
- G = self.cfg.terrain.num_goals. The current config sets G =
    11.
- R = self.terrain.tot_rows and W = self.terrain.tot_cols for
    heightfield /
  trimesh terrain tensors.

Goal and terrain progress variables:
- self.target_pos_rel: float tensor, shape (N, 2). Current goal
    xy minus
  robot root xy, assigned in _update_goals().
- self.target_yaw: float tensor, shape (N,). atan2 direction
    toward
  self.target_pos_rel, assigned in _update_goals().
- self.cur_goal_idx: long tensor, shape (N,). Current goal index
     per env.
  It starts at 0. It can be clamped up to G when the episode
    reaches the
  terminal goal; _gather_cur_goals clamps indices back to G - 1
    for gather.
- self.reach_goal_timer: float tensor, shape (N,). Counts
    physics steps spent
  inside the current goal threshold.
- self.next_target_pos_rel: float tensor, shape (N, 2). Next
    goal xy minus
  robot root xy, assigned in _update_goals().
- self.next_target_yaw: float tensor, shape (N,). atan2
    direction toward
  self.next_target_pos_rel.
- self.terrain_levels: long tensor, shape (N,). Curriculum row /
     level for
  each env.
- self.terrain_types: long tensor, shape (N,). Terrain column /
    template index
  for each env. Do not hard-code specific column ids in reward
    code.
- self.x_edge_mask: bool tensor, shape (R, W). Created for
    trimesh terrain
  from terrain.x_edge_mask and used to detect foot placement
    near x-edges.
- self.cfg.terrain.num_goals: int scalar, current value 11.
    There is no
  self.num_goals attribute in LeggedRobot.py; reward code should
     use
  self.cfg.terrain.num_goals or the literal goal-index map from
    the prompt.

Robot root, orientation, and velocity:
- self.root_states: float tensor, shape (N, 13) in this one-
    robot-per-env
  setup. Columns are root position xyz [0:3], quaternion xyzw
    [3:7],
  world linear velocity [7:10], and world angular velocity
    [10:13].
- self.base_lin_vel: float tensor, shape (N, 3). Root linear
    velocity rotated
```

```
  into the robot base frame.
- self.base_ang_vel: float tensor, shape (N, 3). Root angular
    velocity rotated
  into the robot base frame.
- self.projected_gravity: float tensor, shape (N, 3). Gravity
    vector rotated
  into the robot base frame.
- self.roll: float tensor, shape (N,). Euler roll from self.
    base_quat.
- self.pitch: float tensor, shape (N,). Euler pitch from self.
    base_quat.
- self.yaw: float tensor, shape (N,). Euler yaw from self.
    base_quat.

Commands and control:
- self.commands: float tensor, shape (N, C). Current command
    vector. In the
  current config, self.commands[:, 0] is commanded forward
    velocity.
- self.dt: scalar policy timestep, computed as
  self.cfg.control.decimation * self.sim_params.dt.
- self.num_envs: int scalar N.
- self.device: device string used for tensors, usually "cuda:<id
    >" when the
  GPU pipeline is active, otherwise "cpu".

Joint and action tensors:
- self.dof_pos: float tensor, shape (N, D). Current joint
    positions.
- self.dof_vel: float tensor, shape (N, D). Current joint
    velocities.
- self.default_dof_pos: float tensor, shape (1, D). The code
    creates shape
  (D,) first, then unsqueezes it to (1, D). It broadcasts
    against (N, D).
- self.default_dof_pos_all: float tensor, shape (N, D). Per-env
    copy of the
  default joint pose, available even though not all existing
    rewards use it.
- self.torques: float tensor, shape (N, A). Current applied
    joint torques.
- self.last_torques: float tensor, shape (N, A). Previous
    torques.
- self.last_dof_vel: float tensor, shape (N, D). Previous joint
    velocities.
- self.hip_indices: long tensor, shape (4,). DOF indices for FR/
    FL/RR/RL hip
  joints.
- self.actions: float tensor, shape (N, A). Current clipped
    policy actions.
- self.last_actions: float tensor, shape (N, A). Previous
    clipped actions.

Contact and body tensors:
- self.contact_forces: float tensor, shape (N, B, 3). Net
    contact force per
  rigid body in xyz.
- self.feet_indices: long tensor, shape (F,). Rigid-body indices
     whose names
```

```
  contain cfg.asset.foot_name; F is 4 for Go1.
- self.rigid_body_states: float tensor, shape (N, B, 13). Per-
    body state with
  position xyz [0:3], quaternion xyzw [3:7], linear velocity
    [7:10], and
  angular velocity [10:13].
- self.penalised_contact_indices: long tensor, shape (P,). Rigid
    -body indices
  whose names match cfg.asset.penalize_contacts_on.

Additional temporal foot-contact variables:
- self.contact_filt: bool tensor, shape (N, F). Filtered per-
    foot contact
  signal used by runtime rewards, the harness, and gait
    diagnostics. It is
  the OR of current foot force contact and self.last_contacts.
- self.feet_at_edge: bool tensor, shape (N, F). Maintained by
    the environment
  every post_physics_step; True when a filtered-contact foot is
    on an x-edge.
  Reward code may read it but must not write it.
- self.feet_air_time: float tensor, shape (N, F). Per-foot
    continuous time in
  seconds since the foot last had filtered contact. It is
    maintained every
  post_physics_step and reset on episode reset and touchdown.
- self.feet_first_contact: bool tensor, shape (N, F). True on
    the first filtered
  contact step after a nonzero air-time interval.
- self.feet_air_time_at_contact: float tensor, shape (N, F). On
    first contact,
  stores the air-time duration immediately before touchdown;
    otherwise zero.
- self.feet_air_time_prev: float tensor, shape (N, F). The
    previous-step value
  before the current contact update.
```

Listing I.10: Runtime reward-state interface.

Listing I.11 presents the mutation message with placeholders for the incumbent authored reward, representative child-branch code, and historical parent-evaluation feedback. The context builder supplies evaluation-only evidence; training-phase reflections remain separate artifacts. At generation 0, the incumbent is the bootstrap reward and the parent-feedback field indicates that no preceding evaluations are available.

```
You are an experienced reinforcement learning engineer.

Write a new, improved complete reward module for the same
    quadruped parkour
task, using the current best reward code, the child terrain
    context, and any
available historical eval-only feedback as a starting point.
Return one complete Python reward module and nothing else.

Current best reward code:
<INSERT_BEST_REWARD_CODE>

REWARD PROMPT CONTEXT
```

```
Use the child terrain RUN as the optimization target. Treat
    parent aligned feedback and eval-only summaries as
    historical post-rollout evidence. Representative columns are
     samples, not difficulty levels.

REPRESENTATIVE CHILD TERRAIN COLUMN SUMMARIES AND CODES:
<INSERT REPRESENTATIVE CHILD TERRAIN COLUMN SUMMARIES AND CODES
    HERE>

SELECTED PARENT TERRAINS ALIGNED EVAL-ONLY FEEDBACK SUMMARY:
<INSERT SELECTED PARENT TERRAINS ALIGNED EVAL-ONLY FEEDBACK
    SUMMARY HERE>

Mutation instructions:
- All engineering and terrain-context rules from the system
    message still apply;
  keep the child terrain RUN as the optimization target. Parent
    aligned feedback
  is historical eval-only evidence, and representative columns/
    codes are
  samples rather than a complete column list or difficulty
    levels.
- Keep the same required module format: `import torch`, a top-
    level
  `REWARD_SCALES` dict, and top-level `_reward_<name>(self)`
    functions, each
  returning shape `(self.num_envs,)`.
- Gait integrity constraints from the system message are always
    active hard
  selection constraints: candidates that sacrifice four-foot
    load sharing are
  not selectable, and candidates that sacrifice task progress
    will rank poorly.
  Preserve four-foot load sharing during supported locomotion;
    if adding contact
  penalties, also make good, well-timed contacts and stable
    landings valuable;
  do not penalize legitimate whole-body flight over gaps or
    jumps.
- Diagnostic field names in the context are eval-only reporting
    fields, not
  runtime attributes; write reward code only with documented
    runtime tensors.
```

Listing I.11: Reward-mutation user prompt.

Each reward candidate receives the suffix in Listing I.12. The final field is replaced by one of the following eight directives:

1. Focus this candidate on safe, reliable contacts and edge/foot placement on obstacles.
2. Focus this candidate on smooth, well-timed transitions between consecutive terrain segments.
3. Focus this candidate on forward goal progress and velocity tracking through the goal sequence.
4. Focus this candidate on stable base orientation and disturbance recovery.
5. Focus this candidate on energy efficiency and action smoothness without sacrificing progress.

6. This candidate may restructure the reward boldly: add or replace terms to test a new hypothesis.
7. Focus this candidate on reward shaping expected to generalize across neighboring difficulty levels, avoiding assumptions tied to any single level.
8. Focus this candidate on reducing failure patterns suggested by the available historical diagnostics; if no diagnostic evidence is available, focus on general anti-stall shaping.

```
EXPLORATION DIRECTIVE FOR THIS CANDIDATE:
You are one of several independent reward candidates this round.
    Commit to the direction below even if it is not the most
   obvious choice, so the candidates stay diverse.
<INSERT ONE OF THE EIGHT FIXED EXPLORATION DIRECTIVES HERE>
```

Listing I.12: Per-candidate exploration suffix.

If a generated module fails validation, the error is returned with the retry suffix in Listing I.13.

```
PREVIOUS ATTEMPT FAILED:
<INSERT VALIDATION ERROR FROM THE PREVIOUS ATTEMPT HERE>
Fix this specific issue if it concerns your code; otherwise
   return the FULL reward module again.
```

Listing I.13: Validation-retry suffix.

I.3 Constrained Terrain Summarizer

The constrained terrain summarizer receives the system prompt in Listing I.14 and the user prompt in Listing I.15. The user message supplies the generated terrain code and the failed frontier cell, while the output schema remains fixed.

```
You are an experienced reinforcement learning terrain analyst.

Your only task is to summarize terrain geometry for failed robot
    locomotion segments.

Field definitions:
- terrain_col: integer column index in the terrain grid. Each
   column corresponds to one terrain column.
- frontier_failed_level: terrain row level for this column where
    the selected policy first fails the eval pass threshold.
- difficulty: normalized level parameter, computed as
   frontier_failed_level / (num_rows - 1). Use it to understand
    level-dependent terrain generation code.
- failed_segment: segment from the previous goal to the failed
   target goal, formatted as goal_i_to_j. For example,
   goal_5_to_6 means summarize only the terrain between goal 5
   and goal 6. A value of "all_goals_passed_at_frontier" means
   the column passed every goal at its frontier level: there is
    no failure to summarize (the harness handles these without
   an LLM request).

Rules:
- Summarize terrain features only.
- Do not infer required robot skills.
- Do not propose reward changes.
- Do not summarize the whole terrain.
- Do not describe unrelated goals, unrelated levels, or
   unrelated columns.
```

```
- Do not invent exact numeric distances, heights, gap widths,
    slopes, or offsets unless they are directly available from
    the code or provided input.
- Use concise physical descriptions such as gap crossing, upward
     step, downhill drop, narrow landing, rough patch, obstacle
    sequence, lateral offset, flat segment, pillar transition,
    or wavy ground.
- frontier_failed_level is eval-only cross-eval first-failed-
    level feedback.

Return strict JSON only. Use this schema and copy terrain_col,
    frontier_failed_level, and failed_segment from the request:
{
  "terrain_col": 0,
  "frontier_failed_level": 3,
  "failed_segment": "goal_5_to_6",
  "terrain_feature_summary": "concise physical geometry summary
    of only the failed segment"
}
```

Listing I.14: Constrained terrain-summarizer system prompt.

```
Terrain generation code:
```python
<INSERT TERRAIN GENERATION CODE HERE>
```

Failed frontier cell:
```json
{
  "num_cols": <NUM_COLS>,
  "num_rows": <NUM_ROWS>,
  "num_goals": <NUM_GOALS>,
  "terrain_col": <TERRAIN_COL>,
  "frontier_failed_level": <FRONTIER_FAILED_LEVEL>,
  "difficulty": <FRONTIER_FAILED_LEVEL / (NUM_ROWS - 1)>,
  "failed_segment": "<FAILED_SEGMENT>"
}
```

Return JSON with this exact schema:
{
  "terrain_col": <TERRAIN_COL>,
  "frontier_failed_level": <FRONTIER_FAILED_LEVEL>,
  "failed_segment": "<FAILED_SEGMENT>",
  "terrain_feature_summary": "..."
}
```

Listing I.15: Constrained terrain-summarizer user prompt.