Hierarchical Experimentalist Agents: Learning Skills to Self-Improve In-Context via Experimentation

Abhranil Chandra[1], **Sankaran Vaidyanathan**[1]
Utsav Dhanuka[1], **Varun Gandhi**[1], **Scott Niekum**[1]
{abhranilchan,sniekum}@cs.umass.edu

[1]**Manning College of Information and Computer Sciences, University of Massachusetts Amherst**

Abstract

Large language models (LLMs) are increasingly used to take actions in the real world and augment human decision-making. However, most systems rely on parametric knowledge acquired by imitation or are augmented via post-training with fixed data, retrieval, or search. This paradigm breaks down in novel domains and on sophisticated queries that cannot be answered from prior knowledge alone. Knowing the laws of physics, for instance, does not by itself enable LLMs to answer queries or complete long-horizon tasks in a complex physical system. To solve such novel problems generally, agents should have the fundamental ability of *active experimentation*—exploring and gathering targeted query-specific data or general principles about the unseen environment, and acquiring new reusable skills by learning from these diverse interactions and experiences. We thus introduce **Hierarchical Experimentalist Agents (HExA)**, an in-context, experiment-centric self-improvement framework that (1) iteratively designs and refines query-relevant experiments; (2) incrementally learns from experiences a library of reusable and composable skills that accelerate experimentation within and across tasks; and (3) integrates the experimental data to effectively take actions or answer queries. HExA is entirely in-context and training-free, compatible with any black-box model, and does not rely on external supervision, oracles, or offline data. Existing benchmarks test mainly the parametric knowledge of the models or their ability to reason over facts retrieved from search or tools, and not their ability to discover knowledge through sequential decision making. To evaluate agents on active experimentation tasks, we introduce INTERPHYRE, which builds on the PHYRE 2D procedural physics simulation environment with tool-calling and intervention APIs that agents can use to propose and test hypotheses. Our experiments show that current LLM agents still struggle in these settings. On the hardest levels of INTERPHYRE, Claude Sonnet 4.6 only achieves 2% success, while HExA improves the same model to a success rate of up to 77%. We also show similar improvements across all experiments with open weight models and over other agentic baselines like ReAct and Reflexion. The HExA agent also achieves 44% success by not doing any active experimentation and using only skills that were learned and transferred from easier levels, demonstrating the reusability and generalization of HExA's learned skills. HExA demonstrates that learning via experimentation enables agents to effectively and efficiently make progress on novel tasks, by helping them discover new knowledge and acquire reusable skills. Code for HEXA and INTERPHYRE is publicly available.[1]

[1] https://github.com/General-Exp-3-Continual-Learning-Agent/HeXA-Hierarchical-Experimentalist-Agents

1 Introduction

Large Language Models (LLMs) and LLM Agents have demonstrated remarkable capabilities across a wide range of knowledge work and agentic tasks, from code generation to mathematical reasoning to multi-step planning (Gao et al., 2025; Jimenez et al., 2024; Yao et al., 2023b; Schick et al., 2023). The dominant paradigm for training and deploying these models assumes that parametric knowledge—the information encoded during imitative pretraining over massive collections of human-generated data, augmented with post-training, and further scaled with additional test time compute and retrieval augmented generation—is sufficient to address any query at inference time (Anthropic, 2026; OpenAI, 2025; Team et al., 2024; Team, 2026; Gao et al., 2024). This assumption holds reasonably well in domains densely represented in training corpora and domains where reasoning with prior knowledge or via in-context learning over zero-shot or few-shot user prompts is sufficient (Dong et al., 2024; Mei et al., 2025; Dou et al., 2026). However, this recipe breaks down in two ways. First, an agent may face a novel environment whose dynamics, constraints, or solution strategies were never seen during training, or are poorly understood and unknown even to humans and thus cannot be recalled from any training corpus. Second, even when the relevant general knowledge is encoded in the model's parameters, applying it to a specific instance can demand far more than recall and reasoning: knowing the laws of physics, for example, does not by itself solve an experimental physics problem (Silver & Sutton, 2025; Ying et al., 2025).

In real world settings, given the vast space of novel tasks and queries, an agent's success likely cannot rely on pretrained parametric knowledge alone. Instead, we argue that, agents should, in principle, learn like scientists through active experimentation: this entails exploration, information-gathering, proposing falsifiable hypotheses, testing them through interaction and experiments, and reasoning over what the environment confirms or refutes (Spelke & Kinzler, 2007; Team et al., 2023; 2021). This is precisely the information that parametric recall, chains of thought, and static retrieval struggle to supply. Experimentation, however, is costly: each instance may take many such cycles to solve, and an agent typically faces a whole family of related instances rather than one in isolation. An agent that re-explores from scratch every time fails to take advantage of this structure. This motivates, not just experimentation but also learning from it—distilling reusable skills from past experiments and transferring them to related tasks to amortize the cost of experimentation. These observations form the central question of our work:

How can LLM agents efficiently reason and act in complex, novel domains via in-context experimentation, skill learning, and transfer?

We thus propose **Hierarchical Experimentalist Agents (HExA)** (Figure 1), a novel in-context reinforcement learning framework that extends in-context learning along three directions: (1) the agent actively interacts with a simulator as a tool, proposes and test hypotheses, gathering query-relevant information through deliberate experimentation rather than passive observation; (2) the agent reflects on batches of successful and failed experimental observations to distill reusable skills (reward labeled strategies with feedback of when to use, what it helps with, and when to avoid) into a persistent external skill bank; (3) skills from the bank are retrieved and injected into the agent's context at the start of each new episode to improve experimentation efficiency, test new hypotheses, and guide further exploration. This loop amortizes the cost of exploration across problem instances. Our approach can be viewed as hierarchical, cause HExA uses high-level skills that abstract over many low-level skills and simulator actions (specific placements and parameter settings). Since each skill is distilled while the agent already has access to previously learned ones, later skills implicitly build on earlier ones. Beyond skills for individual instances, HExA also constructs higher-order skills that capture how to learn from interaction itself, supporting consistent progress across new seeds and configurations and generalization to structurally novel, harder levels without any target-level experimentation.

Recent benchmarks like CL-Bench (Dou et al., 2026) focus to some extent on evaluating the in-context learning ability of frontier LLMs, but rely only on static factual analysis of unseen data

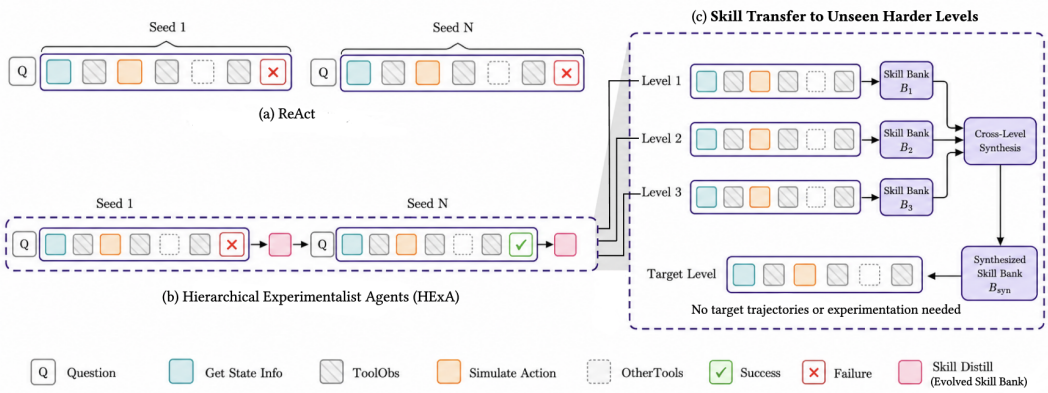


Figure 1: Overview of the HExA framework on Interphyre physics puzzles. (a) Baseline ReAct: each seed is solved independently with no cross-seed learning. Each seed consists of alternating **Get-State-Info** calls (scene layout, gap analysis) and **Simulate-Action** calls (place red ball at (x, y, r)), terminating in success (✓) or failure (×). **(b) HExA:** seeds are grouped into a *meta-episode*. After each seed, an evolver agent distils the trajectory into a *Skill Bank* (physics principles + common mistakes); the evolved bank is injected into subsequent seeds' system prompts. HExA is run for 25 iterations over 50 seeds on each source level to produce a fully-evolved level-specific Skill Bank. **(c) Cross-level transfer to unseen levels:** each source level (`down_to_earth`, `two_body_problem`, `pass_the_parcel`) runs the HExA loop from **(b)** independently (arrows indicate this reuse) to build its Skill Bank. The resulting banks are then synthesised by the *evolver* into a single cross-level Skill Bank for a harder target level (`catapult`), with *no target-level trajectories*.

during pretraining. By contrast, we instantiate and formalize HExA for a broader setting: any task where solving an instance requires active experimentation to gather targeted, instance-specific evidence, rather than recall and reasoning alone. This covers two cases: tasks where the model already holds the relevant general knowledge but must experiment to apply it precisely, and in genuinely unfamiliar domains where the model must learn from scratch through interaction in unseen open-ended environments (Team et al., 2021; Hughes et al., 2024). Physical reasoning is a natural testbed, since models often have the relevant principles yet cannot apply them at the required precision without experimentation and interaction. To study this setting concretely, we introduce Interphyre (Section A.6 and G), an extensible 2D physics reasoning benchmark, built on top of PHYRE (Bakhtin et al., 2019), which adds support for programmatic tool-call and intervention APIs—scene inspection, partial simulation, contact logging, and counterfactual snapshot/restore. This lets us both evaluate agents for their capabilities as well as to teach them to ask query-relevant questions, hypothesize and experiment in unseen domains, and consolidate reusable skills to solve novel tasks efficiently, all in-context.

In summary, our main contributions are:

1. We formalize an **experiment-centric in-context learning problem setting** to evaluate and improve generalist LLM agents in which solving a task requires performing active experimentation with an environment to gather query-specific evidence, either to complement parametric knowledge or to tackle entirely novel, unseen domains. We also introduce **Interphyre**, a benchmark to better evaluate and train agents for this problem setting.
2. We propose **Hierarchical Experimentalist Agents (HExA)**, a training-free, in-context RL framework in which an LLM jointly designs experiments, distills hierarchical and composable skills from interaction experience, and integrates experimental evidence to solve tasks, all without parameter updates or external supervision.
3. We demonstrate that HExA improves both frontier black-box as well as smaller LLMs, achieving in-distribution gains in accuracy of up to **+75%** on the hardest Interphyre level with Claude Sonnet 4.6 as the base model stuck at 2%. Also, the skill banks learned on easier levels transfer to unseen levels without any target-level experimentation, yielding up to **+36%** gains from transferred hierarchical skills alone.

2 Related Works

Learning from Interaction. LLM agents improve decision-making via *parametric fine-tuning*, such as reinforcement learning Ouyang et al. (2022); Rafailov et al. (2023); Guo et al. (2025), or *non-parametric adaptation*. While parametric updates yield strong performance, they often incur high computational costs and risk catastrophic forgetting or over-specialization Ziegler et al. (2020); Shi et al. (2025); Luo et al. (2025), driving the shift toward in-context agents as a more efficient alternative. HEXA, our training-free framework, directly targets this challenge by self-improving LLM agents via in-context experimentation and self-evolution of reusable hierarchical skills.

Skills for LMs. Skills are natural-language modules that capture reusable procedural knowledge for augmenting LMs at inference time (Anthropic, 2025a;b), and skill augmentation has been validated across agent tasks (Si et al., 2023), including coding (Ma et al., 2026a) and web navigation (Wang et al., 2026c). Earlier skill libraries were built largely through human annotation (Li et al., 2026a;b; Liang et al., 2026; Wang et al., 2026b), which is effective but hard to scale in context-learning settings where contexts are long, technical, and domain-specific. Recent work has therefore moved toward automated skill construction. AutoSkill (Yang et al., 2026) extracts reusable behaviors from interaction traces as lifelong, versioned artifacts, while AutoRefine (Qiu et al., 2026) turns agent trajectories into reusable expertise. CoEvoSkills (Zhang et al., 2026) builds multi-file skill packages through generator–verifier co-evolution; EvoSkill (Alzubi et al., 2026) performs failure-driven refinement into structured skill folders; and SkillX (Wang et al., 2026a) distills agent trajectories into a hierarchical skill knowledge base refined through execution feedback. However, these methods typically rely on external feedback signals, such as execution feedback, ground-truth comparison, or task-completion rewards, to evaluate and improve skill quality (Ma et al., 2026b), which are unavailable in context-learning scenarios without automatic feedback. A separate line of work internalizes skills through weight updates: SKILL0 (Lu et al., 2026) uses in-context reinforcement learning (RL) to absorb skills, while SkillRL (Xia et al., 2026) builds a skill bank via RL-guided distillation from stronger teacher trajectories. These approaches require parameter access, limiting their use with closed-source models and sacrificing the interpretability of natural-language skill documents. We therefore propose HEXA as a self-improving framework that discovers and evolves skills through in-context active experimentation.

3 Hierarchical Experimentalist Agents: Learning via Experimentation

HEXA is a training-free, in-context reinforcement learning framework in which an LLM agent learns to solve unfamiliar tasks through active experimentation. Rather than updating model weights, HEXA improves across episodes by distilling interaction experience into a persistent external *skill bank*—a structured collection of natural-language strategies and mistakes defined below—which is re-injected into the agent's context at the start of each new episode. This makes HEXA compatible with any tool-augmented LLM, including closed-source models, and requires no offline data, oracle supervision, or external teacher—only the environment's own interaction feedback. Unlike standard in-context RL, it requires no initial domain-specific training on reward-labeled data; it learns self-improvement from scratch on each new task family. We use the `catapult` level from Interphyre as a running example throughout our paper, as shown in Figure 2 with the skill bank in Figure 7a. The agent has to choose where to place a red ball (x, y) and its dimensions r—so that the ball falls and strikes a pivoting arm, launching a green ball into a distant basket. The full environment descriptions are in Appendix G.

Problem Setup. We consider task families ℓ whose instances $\{(\ell, s_j)\}$ each fix a specific environment configuration: geometry, parameters, and initial conditions. For each family ℓ we have a finite set of instances (or seeds) $\mathcal{S}_\ell = \{s_1, \ldots, s_{|\mathcal{S}_\ell|}\}$, and episodes iterate through $\mathcal{S}_\ell$ in a fixed order. For example, in `catapult`, s_j sets the arm pose, deflector layout, and ceiling-blocker position. The agent has at most T turns to solve s_j. We index turns by t ($t = 0, 1, \ldots, T-1$) and denote the entire experimentation trajectory for s_j as τ_j.

Algorithm 1: HEXA

Input: Actor π_{act}, evolver π_{ev}, level ℓ, seeds of that level $\mathcal{S}_\ell$, rounds R, episodes per round x

```
// Optional warm start (first round only)
```
$\mathcal{K}_0 \leftarrow \pi_{\text{ev}}(\emptyset, \mathcal{T}_0)$ **or** $\mathcal{K}_0 \leftarrow \emptyset$

for $n = 1, \ldots, R$ **do**
  $\mathcal{G}_n \leftarrow$ RETRIEVER$(\mathcal{K}_{n-1}, \ell, M, N)$ `// top-`M` skills + top-`N` mistakes`
  for $i = 1 + x * (n-1), \ldots, x * n$ **do**
    $\tau_i \leftarrow \pi_{\text{act}}(\cdot \mid \ell, s_i, \mathcal{G}_n)$ `// actor prompt: task + tools + skills`
    $r(\tau_i) \leftarrow$ REWARD(τ_i) `// outcome × efficiency; see Appendix A.3.1`
  $\mathcal{K}_n \leftarrow \pi_{\text{ev}}(\mathcal{K}_{n-1}, \mathcal{T}^{(n)})$ `// evolver prompt: bank + trajectories + rewards`

return $\mathcal{K}_R$

The agent interacts through a tool-call interface $\mathcal{F}$ provided by the environment. Interphyre supplies a set of shared tools available on every level—scene inspection, full and partial simulation, and contact logging—as well as some general level-specific analysis tools (shown in Appendix B and Figure 8). For example, `catapult` has access to tools like `describe_scene_geometry`, `predict_first_contact(x,y,r)`, `simulate_with_trace(x,y,r,object_names,stop_step)`, and `finish(x,y,r)`.

At turn t, the agent observes the history $h_t = (a_0, o_0, a_1, o_1, \ldots, a_{t-1}, o_{t-1})$ and selects the next action under actor policy π_{act}. For instance, in `catapult` a typical action is $a_t =$ `simulate_with_trace(0.5,0.4,1.5,["green_ball"])`, returning a set of contact events and a kinematic summary (e.g. peak height, peak speed, and net displacement). A binary predicate GOAL$_\ell$ scores the completed simulation, where SIM(τ) executes trajectory τ in the environment and returns its final state. The episode ends after at most T turns or a terminal action, with success on seed s_j defined as

$$y_j(\pi_{\text{act}}; \ell) = \mathbb{I}[\text{GOAL}_\ell(\text{SIM}(\tau_j)) = \text{true}], \qquad \tau_j \sim \pi_{\text{act}}(\cdot \mid \ell, s_j). \tag{1}$$

The HEXA Loop. As shown in Figure 7b and 6, each round of HEXA runs as follows: a *retriever* selects the most relevant entries from the current skill bank $\mathcal{K}$ and injects them into the *actor*'s context; the actor runs x episodes under that skill context, each scored by a reward; and an *evolver* reads the resulting batch of trajectories and updates $\mathcal{K}$. On the first round, $\mathcal{K}$ may start empty or be warm-started from an offline batch of base-actor trajectories; every subsequent round is identical. Algorithm 1 gives the full loop; initialization and update strategies are ablated in Section 4. HEXA couples three components: an *actor* π_{act} (any tool-augmented LLM) that reasons about experimental hypotheses to test and performs experiments via exploration, interventions and interactions with the environment to generate trajectories; an *evolver* π_{ev} (LLM call to the same model with an evolver prompt) that reads batches of actor trajectories and distills them into an external *skill bank* $\mathcal{K}$; and a *retriever* that selects the most relevant entries from $\mathcal{K}$ and injects them into the actor's context before each episode. We now describe each component in terms of the prompt it receives, what it produces, and why the step is necessary.

Actor prompt and skill-augmented episode. At the start of each episode, the actor receives a prompt containing the task description, the available tools $\mathcal{F}$, and a *skill context* $\mathcal{G}$ retrieved from the current skill bank:

$$\mathcal{G} = \text{RETRIEVER}(\mathcal{K}, \ell, M, N), \tau_j \quad \sim \pi_{\text{act}}(\cdot \mid \ell, s_j, \mathcal{G}). \tag{2}$$

Equation 1 defines the task without method-specific assumptions; any agent—including skill-free baselines such as DIRECT (base model's zero-shot interactions) or REACT—can be evaluated against it. HEXA augments the actor with skill context $\mathcal{G}$ as in Eq. 2, replacing the base policy $\pi_{\text{act}}(\cdot \mid \ell, s_j)$ with the skill-conditioned version. The skill context $\mathcal{G}$ consists of the top-M skills and top-N mistake

records from the bank, ranked using the skill reward labels defined below, prepended as a fixed-size prefix to the actor's system prompt alongside the standard task description and tool documentation. On the first episode, $\mathcal{K}$ may be empty or warm-started (see below); from the second episode onward, it contains skills distilled from prior rounds. The injection mechanism is deliberately simple—it changes only the textual context, not the agent's architecture or decoding—so HEXA works with any tool-augmented LLM. The actor is free to use or disregard individual skills based on its own reasoning about the current instance.

Trajectory reward. Each trajectory τ receives a scalar reward $r(\tau) \in [-1, +1]$ reflecting both *outcome* (success or failure) and *efficiency* (how quickly the outcome was reached, as a fraction of the budget T). Fast, decisive successes earn higher rewards; failures are penalized less when the agent explored extensively before failing, since such trajectories yield richer material for the evolver than one that abandoned the episode after a single attempt. We use discrete reward bins rather than a continuous function because the evolver is an LLM: it can reliably act on qualitative categories (fast success, slow success, exploratory failure, early exit) (Viswanathan et al., 2026; Gunjal et al., 2025), but cannot meaningfully distinguish, say, r=0.73 from r=0.71. The exact reward function is given in Appendix A.3.1.

Evolver prompt and skill distillation. After each round n ($n = 1, \ldots, R$, where $R = |\mathcal{S}_\ell|/x$) of x trajectories, the evolver receives a prompt containing the current skill bank $\mathcal{K}_{n-1}$, the new batch $\mathcal{T}^{(n)}$ (each trajectory rendered as its `(thought,tool_call,observation)` sequence tagged with its reward), and a structured-output instruction. The instruction directs the evolver to use rewards for evaluating trajectory quality and *contrast* high-reward behaviors against low-reward ones, to learn the unique lessons of success over failure based on the observations. From this, the evolver produces two types of distilled knowledge.

Strategy skills are extracted by contrasting high-reward against low-reward trajectories, yielding a set of new skills. Each skill is a structured record defined by a *title*, a *principle* (the insight gained from testing the proposed hypothesis via experimentation), a *when-to-apply* condition, an optional demonstrative *example*, the *source seed* it was derived from, and a *reward score* defined below. For example, the round-14 `catapult` skill `cat_ev_1_001` (Appendix E.4): title "$x \approx 0.5$ is the primary launch sweet spot," principle "placing the red ball at x=0.5 gives a consistent rightward launch; deviate to x=0.3 only on a ceiling hit," applicability "always as the first placement attempt."

Mistake records are extracted from failures alone, yielding records $\mu = (\delta, \rho, \alpha)$ pairing a description δ, a root cause ρ, and a corrective action α. For example: δ "the agent fixates on one launch point and micro-tunes $x/y/r$," ρ "the arm is the obvious mechanism, so failures are met with small perturbations," α "after two failures within $x \pm 0.2$, move to a different x zone." The evolver also recovers *partial skills* from otherwise-failed trajectories: reasoning steps that were individually correct even though the episode failed overall. For example, a trajectory that correctly identifies the catapult arm as the target mechanism but then micro-tunes placement in the wrong direction yields a partial skill encoding the correct mechanism identification, even though the solve failed. This is a learning signal unavailable from successful trajectories alone, since successes do not reveal which intermediate steps were necessary versus incidental. Skills are extracted while the agent already holds previously learned ones, so later skills can build on earlier ones. Whether this compounds into a hierarchy of increasingly abstract principles is an empirical question we investigate in Section 4. Full evolver prompt templates are in Appendix A.3.2.

Skill bank and retrieval. Distilled knowledge is stored in the skill bank $\mathcal{K}$ with two collections, both indexed by task family: a set of *skills* (reusable strategies and principles) and a set of *mistakes* (recurring failure modes to avoid). Each skill carries a *reward score* r_k, computed from the trajectories that demonstrated or used it, as a function of the mean reward of these source trajectories ($\bar{r}_{\text{src}}$). We show an example of this reward label in Section 4.2 with the individual trajectory rewards demonstrated in Appendix A.3.1. This reward label is used for two purposes: ranking skills during retrieval (top-M by r_k), and guiding the evolver in deciding which skills to retain or prune when

the bank is at capacity. Skills derived from fast, efficient successful trajectories carry higher scores; skills from partial reasoning within failed trajectories carry lower scores but may still encode valuable learnings and discoveries.

The bank is capped at $M_{\max}$ skills and $N_{\max}$ mistakes per task family to prevent the injected context from growing without bound and to force the evolver to prioritize the most informative knowledge. Full bank structure and retrieval details are in Appendix A.

The abridged skill bank above shows the information available to HExA *before* it attempts `catapult` seed 45. Unlike raw trajectory memory, the bank organizes evidence from earlier episodes into a default hypothesis, diagnostic conditions, corrective strategies, and mistakes to avoid. Figure 2 subsequently shows how these skills guide the successful HExA trajectory. We can also see how skills from individual experimental and interaction observations are composed together by HExA to from hierarchical and reusable general skills.

Abridged evolved skill bank used on `catapult` seed 45

HExA *receives a skill bank distilled from earlier* `catapult` *episodes, mapping observed trajectories to search decisions.*

Skill 1: DEFAULT ▷ Use stable launch geometry Place the red ball near $x = 0.5$ with a sufficiently large radius to obtain a stable lever arm and strong rightward launch.
Use when: Starting a new seed; deviate if the launch hits the ceiling or violates placement constraints.

Skill 2: DIAGNOSE ▷ Detect the radius plateau At fixed $x = 0.5$, increasing r beyond roughly 1.5 may not improve range because the arm reaches its rotational limit.
Use when: A large-radius launch fails; change contact geometry instead of further increasing r.

Skill 3: CORRECT ▷ Flatten ceiling-blocked trajectories If a smaller radius falls short but a larger one hits the ceiling, launch angle, not energy, is limiting. Shift x toward 0.1–0.3 to flatten the trajectory.
Use when: Radius changes at the same x yield both a short trajectory and a ceiling collision.

AVOID ▷ Local parameter fixation Do not repeatedly micro-tune (x, y, r) around the same unsuccessful launch point.
Correction: Change the search dimension or move to a qualitatively different placement region.

Guiding seed 45. The default launch produces a ceiling overshoot, which the bank diagnoses as a geometry rather than force limitation. HExA therefore shifts from $x \approx 0.5$ to $x = 0.3$ and succeeds with $(x, y, r) = (0.3, 0.9, 1.5)$ in six iterations (Figure 2).

$$\underbrace{\text{default launch}}_{\text{prior skill}} \rightarrow \underbrace{\text{ceiling hit}}_{\text{observation}} \rightarrow \underbrace{\text{shift geometry}}_{\text{correction}} \rightarrow \underbrace{\text{success}}_{\text{6 iterations}}$$

Cross-task skill transfer. The same mechanism supports transfer across task families. Low-level details such as source-task coordinates or object identities are generally not directly reusable, whereas the mechanism-level abstractions distilled above—such as contact geometry controlling impulse direction, moment arm controlling torque, or collision order controlling downstream motion—can remain valid under a different scene configuration. Given evolved source banks $\mathcal{K}_{\ell_1}, \ldots, \mathcal{K}_{\ell_S}$ and only a textual description of an unseen target family ℓ^*, the evolver synthesizes a target bank without observing any target-task trajectories. It (i) selects source skills whose underlying physical or search principles are structurally relevant to ℓ^*, (ii) removes source-specific parameter and entity bindings, (iii) re-grounds each principle in the entities, constraints, and tools of ℓ^*, and (iv) assigns a transfer score according to the directness of the structural correspondence and the degree of corroboration across source banks.

This procedure transfers abstractions rather than memorized solutions: the evolver is prohibited from inventing target-specific coordinates and must express each transferred skill as a mechanism and an applicability condition. For example, evidence about lever-mediated impulse transfer from several source levels can be re-grounded in the pivoting arm and launch dynamics of `catapult`, even though no successful `catapult` placement has been observed. The resulting bank $\mathcal{K}_{\ell^*}$ is injected through the same retriever, enabling zero-shot cross-task transfer, or used as the initialization $\mathcal{K}_0$ for a subsequent within-task HExA run. Full transfer prompts are in Appendix I.5; results are in Section 4.

4 Experiments and Results on Interphyre

In this section, we provide details of INTERPHYRE, and evaluate different LLMs on experiment-centric tasks using INTERPHYRE levels. Our benchmark is designed with the following key properties:

(i) tasks whose solution is unknown or cannot be read off a static description and must be discovered by interacting with the environment

(ii) a programmatic interface that lets the agent inspect the scene, test interventions, and observe intermediate dynamics

(iii) a controllable difficulty range with procedural generation

Unlike benchmarks that primarily ask whether an agent can solve a task or answer a query, our setting requires an environment in which the agent can deliberately interact and gather evidence: inspecting the current instance, testing interventions, observing intermediate dynamics, diagnosing failures, and reusing what it learned across future seeds and levels in an experimentation loop. INTERPHYRE provides this in a controlled 2D physics domain with continuous actions and long-horizon causal structure. Each task requires placing a red ball at (x, y, r), after which the simulator evaluates whether the level-specific success predicate is achieved.

INTERPHYRE builds on the level designs in the PHYRE benchmark (Bakhtin et al., 2019) by introducing APIs for level editing, test-time interventions on environment parameters, and logging traces from the simulation. These APIs can be used to customize the benchmark for evaluating agents on unseen levels, or provided to an LLM for active experimentation. In particular, our experiments use the INTERPHYRE API to expose a structured tool-calling interface for experimentation via interventional scene inspection, partial simulation, full rollouts, contact logs, and level-specific geometric analysis, for example as shown in Figure 9. However, successful actions often depend on contact chains, lever mechanics, collision timing, and strategy-dependent dynamics. Even strong LLMs struggle to anticipate such interactions reliably during inference, particularly when only given the scene description. Now using these intervention API and traces recovered from INTERPHYRE runs, HExA enables the LLM agent to reason over its proposed hypothesis and corresponding experimental observations, than merely guessing solutions from parametric prior knowledge. This lets the agent leverage the information acquired through exploration and interactions, to improve its performance in-context. INTERPHYRE therefore serves as both a benchmark and a diagnostic environment for experimentation based methods such as HEXA. We provide full details of the environment design, curriculum, tool API, observation modes, and prompts in Appendix G and H.

4.1 Experimental Setup, Evaluations, and Baselines

We evaluate five LLMs including frontier black box and small open LLMs—Claude Sonnet 4.6, Qwen-2.5-{7B, 14B} Instruct, GPT-OSS-{20B,120B}, and a random agent (a heuristic policy that samples uniformly from the valid placement region, with no interaction with the environment)—each capped at 25 interaction turns per trajectory on INTERPHYRE levels. We compare against three baselines: REACT (Yao et al., 2023a), which runs one Thought-Action-Observation trajectory per seed; *Reflexion* (Shinn et al., 2023), which adds verbal self-reflection; and DIRECT, which prompts the model to solve the task in a single shot from the scene description alone, with no tools, no environment interaction, and no skills, measuring the base models' parametric capability. With the smaller models, we also include RL training via *GRPO* (Shao et al., 2024), which fine-tunes Qwen-2.5-3B with a binary $\{0, 1\}$ reward, making it the only method that updates model weights and compares directly against our training-free framework. All methods on a given level are evaluated

Table 1: Base LLM agent's (DIRECT baseline) solve rate (%) across eight INTERPHYRE levels. † Pass the Parcel and Catapult evaluated over 50 seeds; all other levels evaluated over 100 seeds.

	Rand	Q-7B	Q-14B	O-20B	O-120B	Sonnet
Cliffhanger	0	0.0	0.0	0.0	0.0	0.0
Tipping Point	3	0.0	3.0	8.0	2.0	0.0
Basket Case	4	0.0	2.0	14.0	13.0	9.0
Down to Earth	8	18.0	0.0	16.0	3.0	14.0
Two Body	14	4.0	16.0	13.0	1.0	68.0
Falling in Place	1	8.0	1.0	3.0	29.0	23.0
Pass the Parcel	0	0.0	0.0	0.0	3.0	0.0†
Catapult	0	0.0	0.0	0.0	0.0	2.0†

on the same 50 seeds, so comparisons are paired on identical instances (full details in Appendix C). In Table 1 above and Table 9 in Appendix D, we organize the eight levels into two difficulty tiers based on base LLM performance (DIRECT) and its performance with ReAct as agent baseline: Tier 1 (the first six levels), and Tier 2 (`pass_the_parcel` at 0%, `catapult` at 2%, with every open-weight model scoring 0% on the latter). These two hard levels from Tier 2, where purely parametric reasoning is insufficient, are the primary testbed for HExA. Throughout, we report two metrics: *solve rate* (% of seeds satisfying GOAL_ℓ, Eq.1) for task success accuracy and average turns per seed (unsolved seeds capped till maximum 25 turns) for efficiency.

4.2 Instantiation of HExA – Skill Initialization, Update, and Reward Labels

The skill bank in HExA has three orthogonal axes of instantiation (Algorithm 1): how it is *initialized* before experimentation, how it is *updated* between rounds and if we use interaction reward labels with the skills. We test three initializations—*Offline*, *Offline-to-Online* (*Off2On*), and *Online*. Both Offline and Off2On warm-start the bank from N_0 base-actor trajectories collected with no skill context, distilled into a fixed skill bank $\mathcal{K}_0 = \text{EVOLVER}(\emptyset, \mathcal{T}_0)$; Offline then freezes the bank, whereas Off2On keeps updating it after new experiments. Online instead starts empty ($\mathcal{K}_0 = \emptyset$) and bootstraps skills purely from the agent's own interactions. For updates, we use two strategies—*Evolving* and *Iterative Replacement*. In Evolving, we pass the evolver both the current bank and the new rounds trajectories, adding, merging, revising, or pruning skills in light of these new evidence ($\mathcal{K}_n = \text{EVOLVER}(\mathcal{K}_{n-1}, \mathcal{T}^{(n)})$). *Iterative Replacement* instead rebuilds the bank from the current rollouts alone ($\mathcal{K}_n = \text{EVOLVER}(\emptyset, \mathcal{T}^{(n)})$), since in the in-context RL loop the actor has already ingested prior skills, so the new skills carry their influence plus the novelty of the latest rollouts. We thus have five variants of skill bank initialization and update—*Off2On* or *Online* initialization $\times$ *Evolution* or *Iterative Replacement* update—plus the frozen Offline bank ($\mathcal{K}_n = \mathcal{K}_{n-1}$). We ablate these settings in Table 4 and *Off2On + Evolution* performs the best.

HExA is also flexible to effectively ingest and use additional experimental evidence and data like trajectory rewards which are a function of outcome reward and efficiency, as shown in Appendix A.3.1, to further guide skill evolution and retrieval. The skill reward labels r_k are again a function of these individual trajectory rewards that invoked the skill,

$$r_k = \text{clamp}\left(\frac{\bar{r}_{\text{src}} + 1}{2},\ 0.1,\ 1.0\right), \tag{3}$$

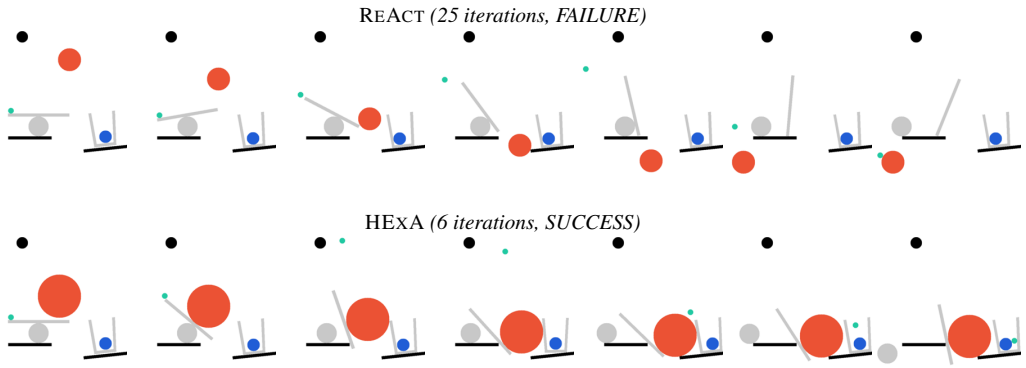


Figure 2: HExA's evolved skill bank enables efficient `catapult` solving. Top: Without hierarchical cross-episode skills, REACT exhausts 25 iterations on ineffective placements. **Bottom:** HExA uses its skill bank to identify launch geometry—not force—as the bottleneck, shifts the drop point to $x = 0.3$, and succeeds in six iterations with $(x, y, r) = (0.3, 0.9, 1.5)$. Frames show the final simulations; full trajectories and bank entries appear in Appendix E.4.

where $\bar{r}_{\text{src}}$ is the mean reward of the source trajectories. Depending on experimental settings, users can define any trajectory reward and skill reward; HExA as a framework is agnostic of these choices.

Method	Acc. (%)	Avg Iter
CLAUDE	2.0	1.0
REACT	8.0	22.9
Reflexion (K=2)	21.3±2.5	21.2±0.7
HEXA (no reward)	50.7±9.4	16.5±2.7
HEXA (Off2On Evol.)	**67.3±9.3**	14.4±1.8

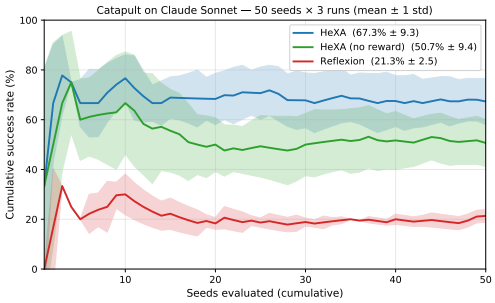


Figure 3: Reward-guided skill reuse substantially improves `catapult`. Claude Sonnet reaches 67.3% with HEXA, versus 8.0% for REACT, 21.3% for Reflexion, and 50.7% without reward guidance, while requiring fewer iterations per solve. **Left:** solve rate and average iterations. **Right:** cumulative success over 50 seeds, averaged over 3 runs; shading denotes ±1 std. See Appendix C.

Table 2: HEXA on open-weight solvers, 50 training/evaluation seeds per cell (max 25 iterations). Qwen-2.5 runs use the Online-Evolving configuration (k=3 seeds per round, 17 rounds); GPT-OSS-120B uses Off2On Evolving. Baselines are the same agent with no skill injection. Per-seed cumulative trajectories in Figure 13.

Level	Method	Acc. (%)	Avg Iters
Down to Earth	Qwen-2.5-3B REACT	8.0	23.8
	Qwen-2.5-3B HEXA	**24.0**	18.6
	Qwen-2.5-7B REACT	62.0	12.5
	Qwen-2.5-7B HEXA	**72.0**	12.6
Two Body Problem	Qwen-2.5-3B REACT	6.0	24.0
	Qwen-2.5-3B HEXA	**14.0**	21.1
	Qwen-2.5-7B REACT	18.0	22.2
	Qwen-2.5-7B HEXA	**34.0**	18.9
Catapult	GPT-OSS-120B REACT	0.0	25.0
	GPT-OSS-120B HEXA	**54.0**	16.2

We also tried skill evolution without rewards, using the evolver LLM's prior knowledge itself to guide skill bank update, ignoring the useful environment grounded rewards. Our ablations in Table 11 show that reward labeled skill banks perform the best. Thus, our default experimental setting for HEXA is Off2On + Evolution with reward labeled skills.

4.3 Results, Analysis, and Ablations

Does active experimentation and skill creation help agents solve complex tasks? On levels where a single REACT trajectory fails, we hypothesize that distilling interaction traces into an evolving skill bank should raise solve rates over base model and non-experimental agent baselines that accumulate no reusable skills (DIRECT, REACT, Reflexion). We test this with frontier model Claude Sonnet, as well as open-weight models like Qwen-2.5-3B and 7B and GPT-OSS-120B against the same non-experimental baselines. On the harder levels where even the base Clause Sonnet model could barely make progress, HEXA learns consistently via in-context experimentation: on `catapult` it reaches 67.3 ± 9.3% against 2.0% (DIRECT), 8.0% (REACT), and 21.3 ± 2.5% (Reflexion) (Figure 3 and Table 3), and on `pass_the_parcel`—which requires discovering the ramp-and-basket mechanism rather than a direct collision—it reaches 60.0% against 24.0%, 16.0%, and 0.0% respectively (Table 12). In Figure 2 we show a representative `catapult` seed solved in a handful of iterations by reusing distilled placement skills while REACT exhausts its budget. The same pattern holds for weaker open models: HEXA improves every open-weight model we tested on every level (Table 2), raising Qwen-2.5-3B from 8.0 → 24.0% and Qwen-2.5-7B from 62.0 → 72.0% on

down_to_earth, 6.0 → 14.0% and 18.0 → 34.0% on two_body_problem, and GPT-OSS-120B from 0.0 → 54.0% on catapult. The improvement therefore comes from distilling past interactions into reusable strategies rather than from extra retries, and also helps solve the tasks more efficiently and in lesser number of turns (full sweep in Figure 13 and Appendix D).

Can HExA leverage learned skills for more efficient exploration and experimentation? In our task settings, simply spending more compute on interactions does not ensure higher task success. The agents benefits from active skill acquisition and reuse (Anthropic, 2025a), helping it with more successes as well as doing so efficiently. These compact skills in turn makes learning more efficient in these complex tasks with lesser interactions. We record the average number of iterations or turns per seed as an efficiency proxy, and track the cumulative success and cumulative iteration curves across our experiments. HExA lowers the average iterations from 22.9 (ReAct) and 21.2 ± 0.7 (Reflexion) to 14.4 ± 1.8, roughly 37% fewer, so each future episode becomes easier to tackle using the accumulated skills (Figure 3 and Figure 10). So the get much more gains in accuracy from HExA at equivalent inference compute. Similar efficiency gains also hold on pass_the_parcel. Reflexion, which adds verbal self-reflection but keeps no persistent skill bank to retrieve and reuse from, takes more trails than HExA. Thus self-reflection between trials help far less than converting past experiences into an explicit, reusable skill bank, and this holds even when the reward labels are removed from the bank. The evolving skill bank in HExA thus helps with structured exploration and amortizes the cost of future experimentation, helping both improve performance and efficiency.

Which ingredients drive improvements in HExA? Using the instantiation axes defined in Section 4.2, we study how these aid learning from experimentation.

How do reward labels help learning via experimentation? We try using HExA with and without the reward labels in the skill bank. In our ablations without the reward labels, every success and failure trajectory and skills formed out of them are treated equally and the LLM evolver implicitly updates the skill bank using its existing knowledge. On Qwen-2.5-7B the unguided reward-free skill variant reaches 64% versus 72% with rewards on down_to_earth and 26% versus 34% on two_body_problem (Table 3). On catapult it drops Claude Sonnet from $67.3 \pm 9.3\%$ to $50.7 \pm 9.4\%$ (Figure 3). Thus the reward labels provide valuable environment feedback grounded in actual explorations and interactions of the LLM actor that better helps assign credit for effective and efficient skill acquisition as well as better guide the LLM evolver to compose, retain or prune skills.

Table 3: Reward-signal ablation on Qwen-2.5-7B, 50 seeds per cell. Full HExA uses reward-guided, two-pass distillation; the no-reward variant treats successful trajectories equally.

Level	Method	Acc. (%)	Iters
Down to Earth	ReAct	62.0	12.5
	HExA no reward	64.0	14.8
	HExA reward	**72.0**	12.6
Two Body	ReAct	18.0	22.2
	HExA no reward	26.0	22.2
	HExA reward	**34.0**	18.9

Table 4: HExA variant accuracy (%) on the two hard levels Claude Sonnet; **Cat.** = catapult; **PtP** = pass_the_parcel. Off2On Evolving (x=3), referred to simply as HExA, is best on both.

Configuration	Catapult	Pass the Parcel
Off2On Evolving (HExA)	**76.0**	**60.0**
Off2On Iterative	56.0	48.0
Online Evolving	44.0	58.0
Online Iterative	36.0	56.0
Offline Static	60.0	48.0

Does skill bank initialization matter? With the update rule fixed, a skill bank initialized with some initial offline bank helps warm-start learning faster, but matters less as more rounds of interaction happens (Section 4.2). As we can see in Table 4, Off2On reaches 76% and 60% on catapult and pass_the_parcel versus 44% and 58% for a pure Online start. The gap on catapult happens where a cold start wastes early rounds rediscovering basic dynamics of the level, an advantage that shrinks once enough enough online experimentation has been done.

What helps skill update more, evolving over the existing bank or rebuilding from scratch? We ablate over the two skill update variants, Evolving and Iterative Replacement (Section 4.2) with the same initialization. Evolving the previous skill bank with new experimentation trajectories across seeds, beats Iterative Replacement where we rebuild the skill bank from scratch every time with new trajectories from the updated LLM actor. On pass_the_parcel the evolving variants reaches

60% accuracy on solving the level versus 48% for skill update from scratch (Table 4). Similarly on `catapult` Evolving skill update helps achieve 76% versus 56% for the iterative variant (Table 4). We also show the cumulative-accuracy curves for these ablations in (Appendix figure 11). We can thus see how gradually evolving the existing skill banks with evidence from new experimentation rounds helps build better skill bank, since re-distilling from scratch discards accumulated skills and amplifies overfitting. A key risk in any iterative skill-update loop is skill overfitting, the skill bank may accumulate increasingly narrow strategies, while discarding general principles that would transfer to new instances. Thus the Evolving update strategy works better, because trajectories in round n are generated while conditioned on $\mathcal{G}_n = \textsc{Retriever}(\mathcal{K}_{n-1}, \ell, M, N)$, they explicitly reflect the influence of previously learned skills. The evolutionary update $\mathcal{K}_n = \pi_{\text{ev}}(\mathcal{K}_{n-1}, \mathcal{T}^{(n)})$ can therefore refine an earlier rule, identify its boundary conditions, merge it with corroborating evidence, or replace it when contradicted by subsequent experience.

Does HExA learn hierarchical skills that build on one another, and do those skills transfer to unseen levels? HExA accumulates skills that compose into increasingly abstract principles, because each round's trajectories are generated while the actor is conditioned on previously retrieved skills, letting a new skill refine, bound, merge with, or overturn an earlier one. Qualitatively, the evolved `catapult` bank shows this hierarchical nature (Appendix E.4): experiments at x=0.5 where r=1.0 fell short and r=1.5 struck the ceiling are not stored as separate parameter settings but consolidated into a higher-level rule—"when radius tuning at fixed x alternates between falling short and hitting the ceiling, stop tuning r and shift x toward 0.1–0.3 to flatten the launch arc"—compressing the many (x, y, r) trials into a physical principle plus a policy for when to abandon local search. The example in Figure 2 applies exactly these hierarchical skills (default launch → diagnosed ceiling failure → corrective geometry shift) to succeed in just six iterations. If the skill banks encode not only level-specific recipes but also abstract principles, those principles should transfer to unseen levels and help experimentation there as well. So we test transfer by synthesizing a target skill bank from easier level banks plus only a textual description of the unseen harder target. No target-level trajectories are used at any stage—and injecting it with no further evolution. Combining evolved banks from three source levels (`down_to_earth`, `two_body_problem`, `pass_the_parcel`) helps improve Claude Sonnet on `catapult` from 2.0% solve rate with the base model and 8.0% with active level interaction via REACT to 44.0% with HExA on the transferred skill bank(Figure 4 and Table 13). Transfer also helps with a single source bank and a smaller actor model like Qwen 7B—transfer on a structurally similar pair (`down_to_earth` → `falling_into_place`) leads to gains of +12% in success rate and the dissimilar pair (`down_to_earth` → `two_body_problem`) still gains +16%. These transfers across dissimilar levels without any online interactions in the held-out test levels indicates that the evolver extracts generalizable abstract principles—momentum transfer, contact geometry, directional impulse—rather than narrow level-specific heuristics. Thus HExA's hierarchical skills and evolution methods help learn and acquire reusable and transferable knowledge that even generalize to unseen complex domains via learning from exploration and experimentation.

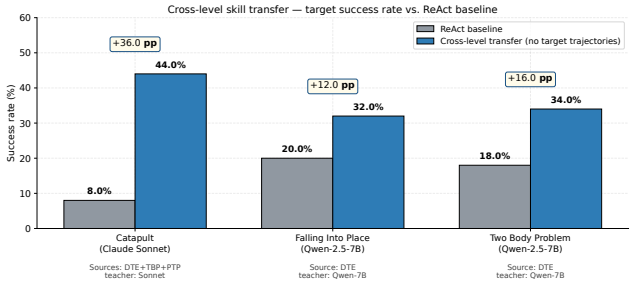


Figure 4: Cross-level skill transfer with no target trajectories. Each target is solved using only skills synthesized from source-level banks; labels report target success and matched REACT baselines. Multi-source transfer to `catapult` yields the largest gain (+36 pp).

How does performance of HExA compare to RL finetuning with weight updates? Finally we ask how learning via experimentation in-context with skill evolution compares to model training with RL in terms of sample-efficient and performance. We compare against GRPO (Shao et al., 2024) fine-tuning Qwen-2.5-3B on the same levels, matched on the number of model updates and

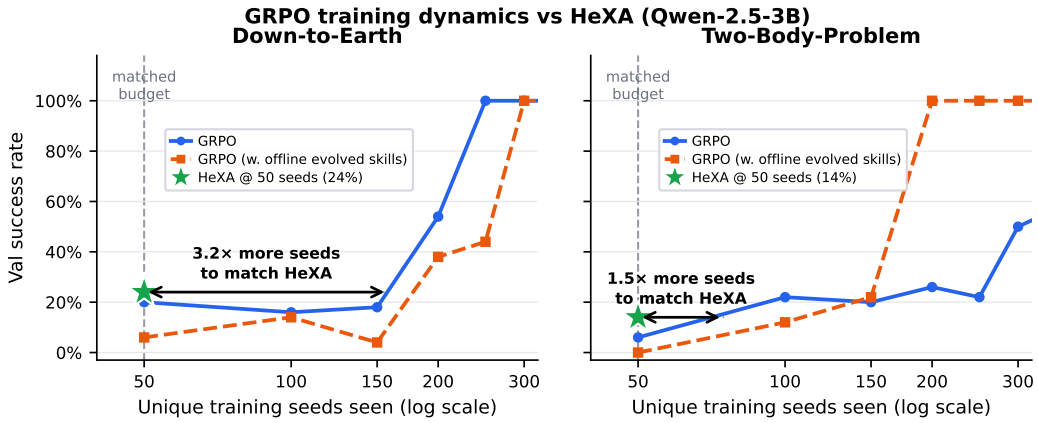


Figure 5: HEXA adapts faster than GRPO at matched interaction budgets. On Qwen-2.5-3B, the green star shows HEXA after 50 training seeds (Down to Earth: 24%; Two Body Problem: 14%), while the dashed line marks the matched-budget GRPO checkpoint. HEXA leads in the low-data regime, although GRPO narrows the gap with more updates.

on 50 unique environment seeds. At this budget GRPO reaches 20% on `down_to_earth` and 6% on `two_body_problem`, versus HEXA's 24% and 14% (Figure 5). The gap reflects the core advantage of in-context skill evolution: strategies distilled from early trajectories are immediately available to later episodes through context, whereas gradient-based methods must accumulate signal over many rollouts before the weights encode equivalent knowledge—so in harder domains where early successes are rare, HEXA can still enable experimentation guided learning from failure and partial-success trajectories. With many more updates, GRPO can eventually close the gap through direct optimization on the environment reward. But combining learning from experimentation in-context to bootstrap initial progress on complex domains with later stages of gradient based RL post-training can be a robust learning method (Appendix D).

5 Conclusion, Limitations, & Future Work

We introduced HEXA, a training-free in-context RL framework in which an LLM designs experiments, collects task-relevant trajectories, distills reward-guided hierarchical skills, and reuses them across tasks without parameter updates, offline data, or external supervision. We also introduced InterPhyRE, a physical-reasoning benchmark that supports active experimentation through APIs for scene inspection, intervention, and trajectory observation. On InterPhyRE, HEXA outperforms base models, ReAct, Reflexion, and matched-budget GRPO, particularly on difficult levels, while reducing the number of interactions required per solve. Its skill banks also transfer zero-shot to unseen levels and improve both frontier and open-weight models. The matched-budget comparison with GRPO highlights a key advantage of in-context learning: a strategy discovered by HEXA can be reused in the next episode through context, whereas gradient-based methods must encode it into model weights over many rollouts. Thus, when interaction data are scarce, skill evolution can amortize exploration more rapidly than weight updates. This suggests using HEXA to bootstrap capabilities in new domains before downstream post-training.

Our evaluation is currently limited to 2D physics, although HEXA itself assumes no reliance on Interphyre and can learn from any interaction trajectories, scalar rewards, and natural-language skills. Evaluating it on other experimentation-driven domains, such as scientific analysis and interactive coding, is therefore an important direction. The framework also depends on an LLM evolver, so skill quality is bounded by its reasoning ability, and its binary success signal may not suit domains with ambiguous objectives. Future work should study cross-domain and out-of-distribution transfer, scalable skill-bank organization and retrieval, and hybrid methods that use evolved skills to guide exploration before consolidating them through parametric RL.

Acknowledgments

This work has taken place in the Safe, Confident, and Aligned Learning and Robotics Lab (SCALAR) at The University of Massachusetts Amherst. SCALAR research is supported in part by the NSF (IIS-2437426), the Long-Term Future Fund, and Open Philanthropy. Scott Niekum holds concurrent appointments as an Associate Professor at the University of Massachusetts Amherst and as an Amazon Scholar. This paper describes work performed at the University of Massachusetts Amherst and was funded by a gift from Amazon.

References

Ossama Ahmed, Frederik Träuble, Anirudh Goyal, et al. CausalWorld: A robotic manipulation benchmark for causal structure and transfer learning. In *International Conference on Learning Representations (ICLR)*, 2021. URL https://arxiv.org/abs/2010.04296.

Kelsey R. Allen, Kevin A. Smith, and Joshua B. Tenenbaum. Rapid trial-and-error learning with simulation supports flexible tool use and physical reasoning. *Proceedings of the National Academy of Sciences*, 117(47):29302–29310, 2020. URL https://arxiv.org/abs/1907.09620.

Salaheddin Alzubi, Noah Provenzano, Jaydon Bingham, Weiyuan Chen, and Tu Vu. Evoskill: Automated skill discovery for multi-agent systems. *arXiv preprint arXiv:2603.02766*, 2026.

Anthropic. Equipping agents for the real world with agent skills. 2025a. URL https://www.anthropic.com/engineering/equipping-agents-for-the-real-world-with-agent-skills.

Anthropic. Introduction to agent skills. 2025b. URL https://anthropic.skilljar.com/introduction-to-agent-skills.

Anthropic. System card: Claude opus 4.6. 2026. URL https://www-cdn.anthropic.com/0dd865075ad3132672ee0ab40b05a53f14cf5288.pdf.

Tayfun Ates, Muhammed Ali Atesoglu, Cansu Yigit, et al. CRAFT: A benchmark for causal reasoning about forces and interactions. In *Findings of the Association for Computational Linguistics: ACL 2022*, 2022. URL https://aclanthology.org/2022.findings-acl.205/.

Anton Bakhtin, Laurens van der Maaten, Justin Johnson, Laura Gustafson, and Ross Girshick. PHYRE: A new benchmark for physical reasoning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2019. URL https://arxiv.org/abs/1908.05656.

Fabien Baradel, Natalia Neverova, Julien Mille, Greg Mori, and Christian Wolf. CoPhy: Counterfactual learning of physical dynamics. In *International Conference on Learning Representations (ICLR)*, 2020. URL https://arxiv.org/abs/1909.12000.

Paul C. Bogdan, Uzay Macar, Neel Nanda, and Arthur Conmy. Thought anchors: Which LLM reasoning steps matter?, 2025. URL https://arxiv.org/abs/2506.19143.

Yuxuan Chen, Piotr Piękos, Mateusz Ostaszewski, Firas Laakom, and Jürgen Schmidhuber. PhysGym. In *Advances in Neural Information Processing Systems (NeurIPS), Datasets & Benchmarks Track*, 2025. URL https://arxiv.org/abs/2507.15550.

Anoop Cherian, Radu Corcodel, Siddarth Jain, and Diego Romeres. LLMPhy: Complex physical reasoning using large language models and world models, 2024. URL https://arxiv.org/abs/2411.08027.

Wei Chow, Jiajun Mao, Bowen Li, Daniel Seita, Vitor Guizilini, and Yue Wang. PhysBench: Benchmarking and enhancing vision-language models for physical world understanding. In *International Conference on Learning Representations (ICLR)*, 2025. URL https://arxiv.org/abs/2501.16411.

Can Demircan, Tankred Saanum, Akshay K. Jagadish, Marcel Binz, and Eric Schulz. Sparse autoencoders reveal temporal difference learning in large language models. In *International Conference on Learning Representations (ICLR)*, 2025. URL https://arxiv.org/abs/2410.01280.

Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Jingyuan Ma, Rui Li, Heming Xia, Jingjing Xu, Zhiyong Wu, Baobao Chang, et al. A survey on in-context learning. In *Proceedings of the 2024 conference on empirical methods in natural language processing*, pp. 1107–1128, 2024.

Shihan Dou, Ming Zhang, Zhangyue Yin, Chenhao Huang, Yujiong Shen, Junzhe Wang, Jiayi Chen, Yuchen Ni, Junjie Ye, Cheng Zhang, Huaibing Xie, Jianglu Hu, Shaolei Wang, Weichao Wang, Yanling Xiao, Yiting Liu, Zenan Xu, Zhen Guo, Pluto Zhou, Tao Gui, Zuxuan Wu, Xipeng Qiu, Qi Zhang, Xuanjing Huang, Yu-Gang Jiang, Di Wang, and Shunyu Yao. CL-Bench: A benchmark for context learning. *arXiv preprint arXiv:2602.03587*, 2026. URL https://arxiv.org/abs/2602.03587.

John Foley, Emma Tosch, Kaleigh Clary, and David Jensen. Toybox: Better atari environments for testing reinforcement learning agents, 2018. URL https://arxiv.org/abs/1812.02850. NeurIPS 2018 Workshop on Systems for ML.

Bofei Gao, Feifan Song, Zhe Yang, Zefan Cai, Yibo Miao, Qingxiu Dong, Lei Li, Chenghao Ma, Liang Chen, Runxin Xu, Zhengyang Tang, Benyou Wang, Daoguang Zan, Shanghaoran Quan, Ge Zhang, Lei Sha, Yichang Zhang, Xuancheng Ren, Tianyu Liu, and Baobao Chang. Omni-MATH: A universal olympiad level mathematic benchmark for large language models. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=yaqPf0KAlN.

Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, Meng Wang, and Haofen Wang. Retrieval-augmented generation for large language models: A survey, 2024. URL https://arxiv.org/abs/2312.10997.

Anisha Gunjal, Anthony Wang, Elaine Lau, Vaskar Nath, Yunzhong He, Bing Liu, and Sean Hendryx. Rubrics as rewards: Reinforcement learning beyond verifiable domains. *arXiv preprint arXiv:2507.17746*, 2025.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bingxuan Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chong Ruan, Damai Dai, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Hanwei Xu, Honghui Ding, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li, Jingchang Chen, Jingyang Yuan, Jinhao Tu, Junjie Qiu, Junlong Li, J. L. Cai, Jiaqi Ni, Jian Liang, Jin Chen, Kai Dong, Kai Hu, Kaichao You, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Liang Zhao, Litong Wang, Liyue Zhang, Lei Xu, Leyi Xia, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingxu Zhou, Meng Li, Miaojun Wang, Mingming Li, Ning Tian, Panpan Huang, Peng Zhang, Qiancheng Wang, Qinyu Chen, Qiushi Du, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, R. J. Chen, R. L. Jin, Ruyi Chen, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shengfeng Ye, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, S. S. Li, Shuang Zhou, Shaoqing Wu, Tao Yun, Tian Pei, Tianyu Sun, T. Wang, Wangding Zeng, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, W. L. Xiao, Wei An, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, X. Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xiaowen Sun, Xiaoxiang Wang, Xinnan Song, Xinyi Zhou, Xianzu Wang, Xinxia Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Yang Zhang, Yanhong Xu, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Yu, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yuan Ou, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou,

Y. X. Zhu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Ying Tang, Yukun Zha, Yuting Yan, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhicheng Ma, Zhigang Yan, Zhiyu Wu, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Zizheng Pan, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, and Zhen Zhang. Deepseek-r1 incentivizes reasoning in llms through reinforcement learning. *Nature*, 645 (8081):633–638, September 2025. ISSN 1476-4687. DOI: 10.1038/s41586-025-09422-z. URL http://dx.doi.org/10.1038/s41586-025-09422-z.

Zhengfu He, Xuyang Ge, Qiong Tang, et al. Dictionary learning improves patch-free circuit discovery in mechanistic interpretability: A case study on othello-GPT, 2024. URL https://arxiv.org/abs/2402.12201.

Edward Hughes, Michael Dennis, Jack Parker-Holder, Feryal Behbahani, Aditi Mavalankar, Yuge Shi, Tom Schaul, and Tim Rocktaschel. Open-endedness is essential for artificial superhuman intelligence. *arXiv preprint arXiv:2406.04268*, 2024.

Dongfu Jiang, Yi Lu, Zhuofeng Li, Zhiheng Lyu, Ping Nie, Haozhe Wang, Alex Su, Hui Chen, Kai Zou, Chao Du, Tianyu Pang, and Wenhu Chen. Verltool: Towards holistic agentic reinforcement learning with tool use, 2025. URL https://arxiv.org/abs/2509.01055.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

Adam Karvonen. Emergent world models and latent variable estimation in chess-playing language models. In *Conference on Language Modeling (COLM)*, 2024. URL https://arxiv.org/abs/2403.15498.

Michael Laskin, Luyu Wang, Junhyuk Oh, Emilio Parisotto, Stephen Spencer, Richie Steigerwald, DJ Strouse, Steven Hansen, Angelos Filos, Ethan Brooks, et al. In-context reinforcement learning with algorithm distillation. *arXiv preprint arXiv:2210.14215*, 2022.

Hao Li, Chunjiang Mu, Jianhao Chen, Siyue Ren, Zhiyao Cui, Yiqun Zhang, Lei Bai, and Shuyue Hu. Organizing, orchestrating, and benchmarking agent skills at ecosystem scale, 2026a. URL https://arxiv.org/abs/2603.02176.

Kenneth Li, Aspen K. Hopkins, David Bau, Fernanda Viégas, Hanspeter Pfister, and Martin Wattenberg. Emergent world representations: Exploring a sequence model trained on a synthetic task. In *International Conference on Learning Representations (ICLR)*, 2023. URL https://arxiv.org/abs/2210.13382.

Shiqian Li, Kewen Wu, Chi Zhang, and Yixin Zhu. I-PHYRE: Interactive physical reasoning. In *International Conference on Learning Representations (ICLR)*, 2024. URL https://arxiv.org/abs/2312.03009.

Xiangyi Li, Wenbo Chen, Yimin Liu, Shenghan Zheng, Xiaokun Chen, Yifeng He, Yubo Li, Bingran You, Haotian Shen, Jiankai Sun, Shuyi Wang, Binxu Li, Qunhong Zeng, Di Wang, Xuandong Zhao, Yuanli Wang, Roey Ben Chaim, Zonglin Di, Yipeng Gao, Junwei He, Yizhuo He, Liqiang Jing, Luyang Kong, Xin Lan, Jiachen Li, Songlin Li, Yijiang Li, Yueqian Lin, Xinyi Liu, Xuanqing Liu, Haoran Lyu, Ze Ma, Bowei Wang, Runhui Wang, Tianyu Wang, Wengao Ye, Yue Zhang, Hanwen Xing, Yiqi Xue, Steven Dillmann, and Han chung Lee. Skillsbench: Benchmarking how well agent skills work across diverse tasks, 2026b. URL https://arxiv.org/abs/2602.12670.

Yuan Liang, Ruobin Zhong, Haoming Xu, Chen Jiang, Yi Zhong, Runnan Fang, Jia-Chen Gu, Shumin Deng, Yunzhi Yao, Mengru Wang, Shuofei Qiao, Xin Xu, Tongtong Wu, Kun Wang, Yang Liu, Zhen Bi, Jungang Lou, Yuchen Eleanor Jiang, Hangcheng Zhu, Gang Yu, Haiwen Hong, Longtao Huang, Hui Xue, Chenxi Wang, Yijun Wang, Zifei Shan, Xi Chen, Zhaopeng Tu, Feiyu Xiong,

Xin Xie, Peng Zhang, Zhengke Gui, Lei Liang, Jun Zhou, Chiyu Wu, Jin Shang, Yu Gong, Junyu Lin, Changliang Xu, Hongjie Deng, Wen Zhang, Keyan Ding, Qiang Zhang, Fei Huang, Ningyu Zhang, Jeff Z. Pan, Guilin Qi, Haofen Wang, and Huajun Chen. Skillnet: Create, evaluate, and connect ai skills, 2026. URL https://arxiv.org/abs/2603.04448.

Zhengxi Lu, Zhiyuan Yao, Jinyang Wu, Chengcheng Han, Qi Gu, Xunliang Cai, Weiming Lu, Jun Xiao, Yueting Zhuang, and Yongliang Shen. SKILL0: In-context agentic reinforcement learning for skill internalization. *arXiv preprint arXiv:2604.02268*, 2026. URL https://arxiv.org/abs/2604.02268.

Yun Luo, Zhen Yang, Fandong Meng, Yafu Li, Jie Zhou, and Yue Zhang. An empirical study of catastrophic forgetting in large language models during continual fine-tuning, 2025. URL https://arxiv.org/abs/2308.08747.

Yingwei Ma, Yue Liu, Xinlong Yang, Yanhao Li, Kelin Fu, Yibo Miao, Yuchong Xie, Zhexu Wang, and Shing-Chi Cheung. Scaling coding agents via atomic skills, 2026a. URL https://arxiv.org/abs/2604.05013.

Ziyu Ma, Shidong Yang, Yuxiang Ji, Xucong Wang, Yong Wang, Yiming Hu, Tongwen Huang, and Xiangxiang Chu. Skillclaw: Let skills evolve collectively with agentic evolver, 2026b. URL https://arxiv.org/abs/2604.08377.

Michael Matthews, Michael Beukman, Chris Lu, and Jakob Foerster. Kinetix: Investigating the training of general agents through open-ended physics-based control tasks. In *International Conference on Learning Representations (ICLR)*, 2025. URL https://arxiv.org/abs/2410.23208. Oral presentation.

Lingrui Mei, Jiayu Yao, Yuyao Ge, Yiwei Wang, Baolong Bi, Yujun Cai, Jiazhi Liu, Mingyu Li, Zhong-Zhi Li, Duzhen Zhang, Chenlin Zhou, Jiayi Mao, Tianze Xia, Jiafeng Guo, and Shenghua Liu. A survey of context engineering for large language models, 2025. URL https://arxiv.org/abs/2507.13334.

Amir Moeini, Jiuqi Wang, Jacob Beck, Ethan Blaser, Shimon Whiteson, Rohan Chandra, and Shangtong Zhang. A survey of in-context reinforcement learning. *arXiv preprint arXiv:2502.07978*, 2025.

OpenAI. Gpt-5 technical report. 2025. URL https://cdn.openai.com/gpt-5-system-card.pdf.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul F Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (eds.), *Advances in Neural Information Processing Systems*, volume 35, pp. 27730–27744. Curran Associates, Inc., 2022.

Judea Pearl. *Causality: Models, Reasoning, and Inference*. Cambridge University Press, Cambridge, 2nd edition, 2009.

Libin Qiu, Zhirong Gao, Junfu Chen, Yuhang Ye, Weizhi Huang, Xiaobo Xue, Wenkai Qiu, and Shuo Tang. Autorefine: From trajectories to reusable expertise for continual llm agent refinement. *arXiv preprint arXiv:2601.22758*, 2026.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Stefano Ermon, Christopher D. Manning, and Chelsea Finn. Direct preference optimization: your language model is secretly a reward model. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA, 2023. Curran Associates Inc.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

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

Haizhou Shi, Zihao Xu, Hengyi Wang, Weiyi Qin, Wenyuan Wang, Yibin Wang, Zifeng Wang, Sayna Ebrahimi, and Hao Wang. Continual learning of large language models: A comprehensive survey. *ACM Computing Surveys*, 58(5):1–42, 2025.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36:8634–8652, 2023.

Shuzheng Si, Wentao Ma, Haoyu Gao, Yuchuan Wu, Ting-En Lin, Yinpei Dai, Hangyu Li, Rui Yan, Fei Huang, and Yongbin Li. SpokenWOZ: A large-scale speech-text benchmark for spoken task-oriented dialogue agents. In *Thirty-seventh Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2023. URL https://openreview.net/forum?id=viktK3nO5b.

David Silver and Richard S Sutton. Welcome to the era of experience. *Google AI*, 1:11, 2025.

Elizabeth S Spelke and Katherine D Kinzler. Core knowledge. *Developmental science*, 10(1):89–96, 2007.

Alex F. Spies, William Edwards, Michael I. Ivanitskiy, et al. Transformers use causal world models in maze-solving tasks. In *International Conference on Learning Representations (ICLR)*, 2025. URL https://arxiv.org/abs/2412.11867.

Adaptive Agent Team, Jakob Bauer, Kate Baumli, Satinder Baveja, Feryal Behbahani, Avishkar Bhoopchand, Nathalie Bradley-Schmieg, Michael Chang, Natalie Clay, Adrian Collister, et al. Human-timescale adaptation in an open-ended task space. *arXiv preprint arXiv:2301.07608*, 2023.

Gemini Team, Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, Katie Millican, et al. Gemini: A family of highly capable multimodal models. arxiv 2023. *arXiv preprint arXiv:2312.11805*, 2024.

Open Ended Learning Team, Adam Stooke, Anuj Mahajan, Catarina Barros, Charlie Deck, Jakob Bauer, Jakub Sygnowski, Maja Trebacz, Max Jaderberg, Michael Mathieu, Nat McAleese, Nathalie Bradley-Schmieg, Nathaniel Wong, Nicolas Porcel, Roberta Raileanu, Steph Hughes-Fitt, Valentin Dalibard, and Wojciech Marian Czarnecki. Open-ended learning leads to generally capable agents, 2021. URL https://arxiv.org/abs/2107.12808.

Qwen Team. Qwen3. 5: Towards native multimodal agents. *URL: https://qwen. ai/blog*, 2026.

Mark Towers, Ariel Kwiatkowski, Jordan Terry, John U. Balis, Gianluca De Cola, Tristan Deleu, Manuel Goulão, Andreas Kallinteris, Markus Krimmel, Arjun KG, et al. Gymnasium: A standard interface for reinforcement learning environments. *arXiv preprint arXiv:2407.17032*, 2024. URL https://arxiv.org/abs/2407.17032.

Vijay Viswanathan, Shiqi Wang, Devamanyu Hazarika, Chirag Nagpal, Tongshuang Wu, Graham Neubig, and Yuning Mao. Discretizing reward models, 2026. URL https://arxiv.org/abs/2606.21795.

Chenxi Wang, Zhuoyun Yu, Xin Xie, Wuguannan Yao, Runnan Fang, Shuofei Qiao, Kexin Cao, Guozhou Zheng, Xiang Qi, Peng Zhang, and Shumin Deng. SkillX: Automatically constructing skill knowledge bases for agents. *arXiv preprint arXiv:2604.04804*, 2026a. URL https://arxiv.org/abs/2604.04804.

Jiayu Wang, Yifei Ming, Zixuan Ke, Shafiq Joty, Aws Albarghouthi, and Frederic Sala. Skillorchestra: Learning to route agents via skill transfer, 2026b. URL https://arxiv.org/abs/2602.19672.

Yi R. Wang, Jiafei Duan, Dieter Fox, and Siddhartha Srinivasa. NEWTON: Are large language models capable of physical reasoning? In *Findings of the Association for Computational Linguistics: EMNLP 2023*, 2023. URL https://arxiv.org/abs/2310.07018.

Zhaoyang Wang, Qianhui Wu, Xuchao Zhang, Chaoyun Zhang, Wenlin Yao, Fazle Elahi Faisal, Baolin Peng, Si Qin, Suman Nath, Qingwei Lin, Chetan Bansal, Dongmei Zhang, Saravan Rajmohan, Jianfeng Gao, and Huaxiu Yao. Webxskill: Skill learning for autonomous web agents, 2026c. URL https://arxiv.org/abs/2604.13318.

Peng Xia, Jianwen Chen, Hanyang Wang, Jiaqi Liu, Kaide Zeng, Yu Wang, Siwei Han, Yiyang Zhou, Xujiang Zhao, Haifeng Chen, Zeyu Zheng, Cihang Xie, and Huaxiu Yao. Skillrl: Evolving agents via recursive skill-augmented reinforcement learning, 2026. URL https://arxiv.org/abs/2602.08234.

Xinrun Xu, Pi Bu, Ye Wang, Börje F Karlsson, Ziming Wang, Tengtao Song, Qi Zhu, Jun Song, Zhiming Ding, and Bo Zheng. Deepphy: Benchmarking agentic vlms on physical reasoning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pp. 34160–34168, 2026.

Yutao Yang, Junsong Li, Qianjun Pan, Bihao Zhan, Yuxuan Cai, Lin Du, Jie Zhou, Kai Chen, Qin Chen, Xin Li, Bo Zhang, and Liang He. AutoSkill: Experience-driven lifelong learning via skill self-evolution. *arXiv preprint arXiv:2603.01145*, 2026. URL https://arxiv.org/abs/2603.01145.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The eleventh international conference on learning representations*, 2022.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models, 2023a. URL https://arxiv.org/abs/2210.03629.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*, 2023b. URL https://openreview.net/forum?id=WE_vluYUL-X.

Lance Ying, Katherine M Collins, Prafull Sharma, Cedric Colas, Kaiya Ivy Zhao, Adrian Weller, Zenna Tavares, Phillip Isola, Samuel J Gershman, Jacob D Andreas, et al. Assessing adaptive world models in machines with novel games. *arXiv preprint arXiv:2507.12821*, 2025.

Hanrong Zhang, Shicheng Fan, Henry Peng Zou, Yankai Chen, Zhenting Wang, Jiayu Zhou, Chengze Li, Wei-Chieh Huang, Yifei Yao, Kening Zheng, Xue Liu, Xiaoxiao Li, and Philip S. Yu. Coevoskills: Self-evolving agent skills via co-evolutionary verification, 2026. URL https://arxiv.org/abs/2604.01687.

Yanming Zheng, Mingyu Yan, Jingzhou Chen, et al. ContPhy: Continuum physical concept learning and reasoning from videos. In *International Conference on Machine Learning (ICML)*, 2024. URL https://proceedings.mlr.press/v235/zheng24l.html.

Daniel M. Ziegler, Nisan Stiennon, Jeffrey Wu, Tom B. Brown, Alec Radford, Dario Amodei, Paul Christiano, and Geoffrey Irving. Fine-tuning language models from human preferences, 2020. URL https://arxiv.org/abs/1909.08593.

A HExA: More Methodological Details

A.1 Overview

Many agentic tasks require an LLM to go beyond its parametric knowledge and actively interact with an environment to gather the instance-specific information needed to complement existing parametric knowledge in complex queries or experimental tasks; or to fundamentally interact and experiment in an unseen domain to reason, act and correctly complete some novel user task or query beyond the model's existing parametric knowledge and capabilities. An intuitive paradigm for learning in such settings and retaining useful reusable information is via inference-time in-context skill augmentation: distilling reusable strategies and common pitfalls from ***interaction experience and active experimentation*** into compact natural-language ***skills*** that are reusable in the agent's context at the start of each new episode, amortizing cost of exploration and skill reuse in newer tasks or domains. Turning this idea into a working self-improving framework, however, faces three difficulties: (1) the agent has no access to oracle solutions, expert demonstrations, or optimal offline data from the task family; (2) there is no human or oracle guided skills to start with and no external reward model or verifier beyond the interaction feedback on the unseen domain to bootstrap the learning; and (3) the skill library must be autonomously evolved and remain compact and bounded to fit within the agent's context window while still capturing the most useful knowledge.

In our paper, we introduce ***HExA*** (Hierarchical Experimentalist Agents), shown in Figure 1, a training-free, in-context reinforcement learning framework that autonomously discovers, evolves, and transfers reusable skills without parameter updates, external feedback, or human annotation. At its core, HExA operates a two-agent loop over a sequence of rounds (A.3): an *actor* π_{act} actively experiments with the environment through a tool-call interface, generating trajectories of successes and failures; an *evolver* π_{ev} reads batches of these trajectories, contrasts high-reward against low-reward episodes, and distills the result into an evolving external *skill bank* $\mathcal{K}$ via a two-phase contrastive and failure-analytic distillation process (A.3.2). A reward-weighted *retriever* (A.3.4) selects the most relevant skills and injects them into the actor's context before each subsequent episode, so that the cost of exploration is amortized across problem instances rather than paid from scratch each time.

A key risk in any iterative skill-evolution loop is *skill drift*: the evolver may accumulate increasingly narrow strategies that overfit to the particular instances seen so far, while discarding general principles that would transfer to new instances. To address this, HExA supports multiple initialisation and update regimes (A.3.5) that control the balance between retaining proven skills and incorporating new evidence, ranging from a frozen offline bank to a fully online evolution strategy. Beyond within-task learning, HExA further enables *cross-task skill transfer* (A.4): skill banks evolved on one or more source task families are re-grounded by the evolver onto a structurally related target task, enabling zero-shot transfer with no target-task interaction. In this way, HExA autonomously constructs task-specific skills that can be deployed with any frozen LLM at inference time to enable in-context self-improvement without weight updates.

A.2 Problem Formulation

We consider a task family ℓ consisting of a set of task instances $\{(\ell, s_j)\}$, where each instance is identified by an index s_j (e.g., a random seed) that determines the specific configuration of the environment (geometry, parameters, initial conditions). The agent interacts with each instance through a tool-call interface exposing a set of tools $\mathcal{F}$, and the environment defines a binary success predicate $\textsc{Goal}_\ell$ that evaluates the outcome of a completed simulation.

At each iteration t of an episode, the agent observes the conversation history $h_t = (o_0, a_1, o_1, \ldots, a_{t-1}, o_{t-1})$, where o_i is an observation returned by a tool call and $a_i \in \mathcal{F}$ is a tool invocation, and selects the next action according to its policy π. The episode terminates after at most T iterations or when the agent issues a terminal action. We define a success indicator y_j for task

instance s_j as:

$$y_j(\pi;\, \ell,\, \mathcal{G}) \;=\; \mathbb{I}\big[\textsc{Goal}_\ell(\textsc{Sim}(\tau_j)) = \text{true}\big], \qquad \tau_j \sim \pi(\cdot \mid \ell,\, s_j,\, \mathcal{G}), \tag{4}$$

where τ_j is the trajectory generated by π, SIM runs the environment simulation to completion, and $\mathcal{G}$ is an optional skill context.

Our goal is to maximize the solve rate across instances of ℓ without any parameter updates to π. To this end, we introduce a natural-language *skill bank* $\mathcal{K}$, a structured collection of reusable strategies and common-mistake records that is prepended to the agent's system prompt at the start of each episode:

$$\tau_j \sim \pi(\cdot \mid \ell,\, s_j,\, \mathcal{G}), \qquad \mathcal{G} = \textsc{Retriever}(\mathcal{K},\, \ell,\, M,\, N), \tag{5}$$

where RETRIEVER selects the top-M skills by reward and top-N mistakes from $\mathcal{K}$. The skill bank $\mathcal{K}$ is constructed and evolved entirely through HExA's actor-evolver loop (Section A.3), with no human annotation, external teacher, or access to oracle solutions. Each trajectory is additionally assigned a scalar reward $r(\tau_j) \in [-1, +1]$ that reflects both outcome and efficiency (Eq. 6), providing the evolver with a richer learning signal than the binary success indicator alone.

A.3 The HExA Framework

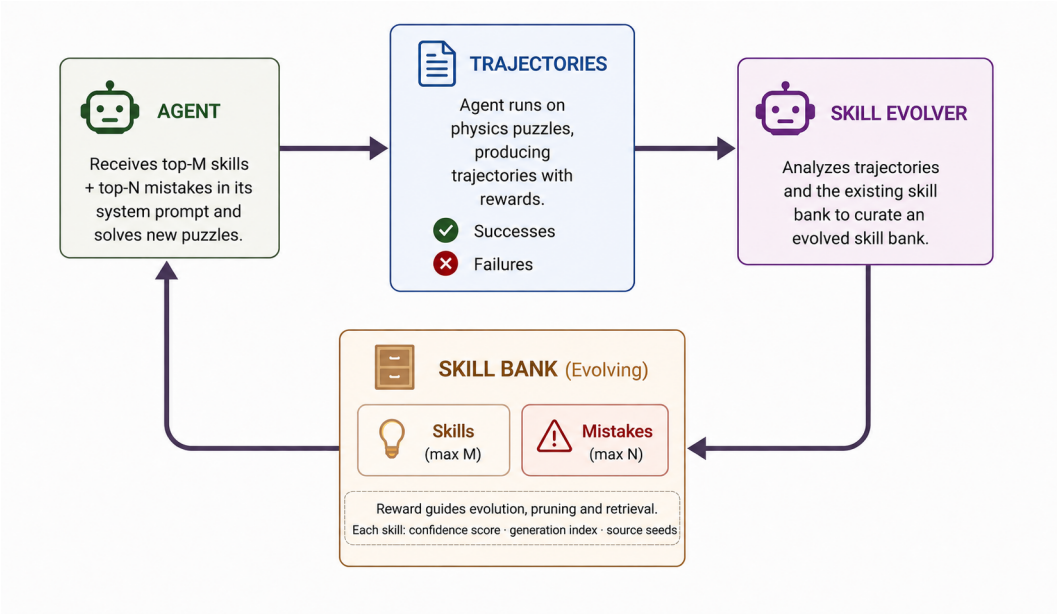
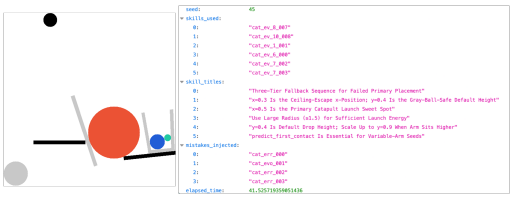


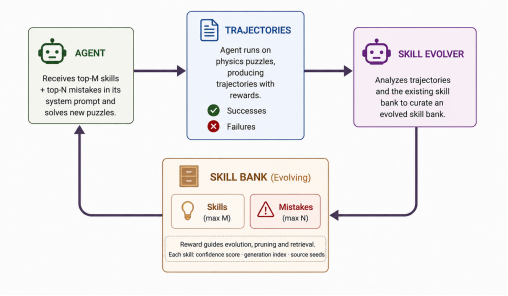
Figure 6: The HExA actor-evolver loop. The actor receives distilled skills in its system prompt and runs on physics puzzles, producing trajectories annotated with rewards (successes ✓, failures ✗). The Skill Evolver analyzes these trajectories and curates the evolving Skill Bank: skills (max M) are merged, pruned, or promoted by reward signal, and common mistakes (max N) are updated accordingly. Each skill carries a generation index tracking its lineage across evolution rounds (Eq. 7). The updated bank is injected back into the actor's system prompt for the next round, closing the loop.

HExA instantiates the skill-augmented policy of Eq. 5 through a two-agent architecture operating over a sequence of *rounds*. An **actor** π_{act} (any tool-augmented LLM) generates trajectories by interacting with the environment, while an **evolver** π_{ev} (which may be the same model) reads batches of actor trajectories and distills them into the skill bank $\mathcal{K}$. At the start of each new episode, the RETRIEVER selects the most relevant skills from $\mathcal{K}$ and injects them into the actor's context, so that each subsequent trajectory benefits from the accumulated experience of all prior ones.

This loop can be viewed as an instantiation of in-context reinforcement learning (Laskin et al., 2022; Moeini et al., 2025): the agent's policy improves over rounds through context augmentation rather



(a) Example on `catapult` with its skill bank.



(b) HExA actor–evolver–retriever loop.

Figure 7: HExA learns reusable skills through an actor–evolver–retriever loop. **(a)**An example on `catapult` level illustrates the kind of physics reasoning skills HExA distills. **(b)** In each round, the actor generates reward-tagged trajectories, the evolver updates the skill bank, and the retriever injects the most relevant skills back into the actor for the next attempt.

than weight updates, and the skill bank serves as a compressed, curated form of the cross-episode context that classical in-context RL methods maintain in raw form. Unlike standard in-context RL, however, HExA does not require pretraining on a task distribution; it operates on a single frozen model and learns from scratch on each task family.

A.3.1 Trajectory Reward

Not all trajectories are equally informative. A fast, decisive solve reveals a high-reward strategy; a failure that explored extensively yields richer material for the evolver than one that abandoned the episode after a single attempt. To communicate this signal, we assign a scalar reward $r(\tau) \in [-1, +1]$ to each trajectory τ, reflecting both *outcome* (success or failure) and *efficiency* (iteration count t at termination):

$$r(\tau) = \begin{cases} +1.00 & \text{success},\ t \leq 3 \\ +0.75 & \text{success},\ t \leq 7 \\ +0.50 & \text{success},\ t \leq 15 \\ +0.25 & \text{success},\ t \leq T \\ -0.50 & \text{failure},\ t \geq 10 \\ -0.75 & \text{failure},\ t < 10 \end{cases} \tag{6}$$

The asymmetry in failure penalties is deliberate: an agent that explores extensively before failing produces richer interaction records for the evolver to extract partial skills and diagnose mistakes.

A.3.2 Skill Distillation from Experimentation on Self-Proposed Hypotheses

At each round n, the evolver π_{ev} receives the batch of trajectories $\mathcal{T}^{(n)} = \mathcal{T}^{(n),+} \cup \mathcal{T}^{(n),-}$ generated by the actor, where $\mathcal{T}^{(n),+}$ and $\mathcal{T}^{(n),-}$ denote the successful and failed subsets respectively, each annotated with its scalar reward. The evolver performs a two-phase distillation to extract complementary forms of knowledge.

Phase 1: Contrastive skill extraction. Given the full batch $\mathcal{T}^{(n)}$ with reward annotations, the evolver contrasts high-reward trajectories against low-reward ones to identify the strategies and insights that distinguish success from failure. This produces 4–6 *strategy skills*, each consisting of a short title, a 2–3 sentence natural-language principle describing the underlying mechanism, and an applicability predicate specifying when the skill is relevant. The evolver is instructed to weight its analysis by reward: skills derived from fast, efficient solves are treated as more reliable than those from trajectories that barely succeeded.

Algorithm 2: HExA

Input: Base agent π_{act}, evolver π_{ev}, level ℓ, regime $\mathcal{R} \in \{\text{Offline}, \text{Off2On}, \text{Online}\}$, update strategy $\mathcal{U} \in \{\text{Evolution}, \text{Iterative}, \text{None}\}$, rounds R, trajectories per round x, initial batch size N_0, retrieval sizes M, N

Output: Final skill bank $\mathcal{K}_R$ and per-round trajectories $\{\mathcal{T}^{(n)}\}_{n=1}^{R}$

```
// Stage 1: Initialise skill bank K_0
if R ∈ {Offline, Off2On} then
    Collect batch T_0 of N_0 trajectories from π_act on ℓ
    K_0 ← Evolver(∅, T_0)                    // teacher-driven distillation
else
    K_0 ← ∅                                  // cold start (Online)

// Stage 2: Iterative rounds of generation, distillation, and update
for n = 1, 2, ..., R do
    G_n ← Retriever(K_{n-1}, ℓ, M, N)        // retrieve top-M skills, top-N mistakes
    T^(n) ← ∅
    for i = 1, ..., x do
        τ_i ← π_act(ℓ | G_n)                 // run skill-augmented agent
        r(τ_i) ← REWARD(τ_i)                 // Eq. 6
        T^(n) ← T^(n) ∪ {(τ_i, r(τ_i))}
    if R = Offline then
        K_n ← K_{n-1}                        // frozen bank
    else if U = Evolution then
        K_n ← Evolver(K_{n-1}, T^(n))        // merge, revise, prune
    else if U = Iterative then
        K_n ← Evolver(∅, T^(n))              // re-distil from new trajectories only
return K_R, {T^(n)}_{n=1}^R
```

Phase 2: Mistake and partial-skill extraction. Focusing on the failure subset $\mathcal{T}^{(n),-}$, the evolver produces two complementary outputs. *Common mistakes* are structured records $\mu = (\delta, \rho, \alpha)$, where δ describes the error, ρ identifies its root cause (the broken causal belief that led to it), and α specifies a concrete corrective strategy. *Partial skills* capture correct reasoning steps that appeared *within* otherwise failed trajectories: an agent may have identified a valid strategy or correctly diagnosed a mechanism before making an error elsewhere. These partial successes cannot be discovered from successful trajectories alone and are a distinctive source of learning signal in HExA.

A.3.3 Skill Bank

Distilled knowledge is stored in the skill bank $\mathcal{K}$ with two collections, both indexed by task family: a set of *skills* (reusable strategy and principles) and a set of *mistakes* (patterns recording and describing recurring failure modes to avoid). Each skill carries a reward label, computed from the interaction rewards of the trajectories that demonstrated it:

$$r_k \;=\; \text{clamp}\left(\frac{\bar{r}_{\text{src}} + 1}{2},\; 0.1,\; 1.0\right), \tag{7}$$

where $\bar{r}_{\text{src}}$ is the mean reward of the source trajectories from which the skill was extracted. Skills derived from fast, efficient solves carry higher rewards; skills surfaced from partial reasoning within failed trajectories carry lower rewards but may still encode valuable discoveries. The skill bank is hard-capped at $M_{\max}$ skills and $N_{\max}$ mistakes per task family to prevent the injected context from

growing without bound and to force the evolver to prioritize the most informative knowledge during evolution rounds. This is where the reward labels and the creation of hierarchical skills, help in improving the skill bank even though it is capped at a maximum number of skills.

A.3.4 Skill Retrieval and Injection

At the start of each episode, the RETRIEVER selects the top-M skills from $\mathcal{K}$ sorted by reward r_k and the top-N mistakes for the relevant task family, and injects them into the actor's system prompt as the structured skill context $\mathcal{G}$ defined in Eq. 5. The actor receives this block alongside the standard task description and tool documentation, and is free to use or disregard individual skills based on its own reasoning about the current instance. This injection mechanism is deliberately simple: it adds a fixed-size prefix to the prompt rather than modifying the agent's architecture or decoding procedure, making HExA compatible with any tool-augmented LLM, including closed-source API-served models.

A.3.5 Initialisation, Update Regimes, and Full Algorithm

The loop described above admits several configurations that vary along two orthogonal axes: how the skill bank is *initialised* before evaluation begins, and how it is *updated* between rounds. Algorithm 2 summarizes the full procedure; we now describe the design axes it parameterizes.

Initialisation. In the **Offline** and **Offline-to-Online** (Off2On) regimes, a warm-start batch of N_0 trajectories is collected from the base actor (without any skill context) and distilled into an initial bank $\mathcal{K}_0$ via Evolver$(\emptyset, \mathcal{T}_0)$ (Algorithm 2, lines 3–4). In the **Pure Online** regime, the bank starts empty ($\mathcal{K}_0 = \emptyset$) and the agent must bootstrap skills entirely from its own online interactions (line 6).

Update strategy. Given trajectories from round n, the bank can be updated in two ways. Under **Evolution**, the evolver receives both the current bank and the new trajectories, producing $\mathcal{K}_n =$ Evolver$(\mathcal{K}_{n-1}, \mathcal{T}^{(n)})$ by merging, revising, or pruning existing skills in light of new evidence (line 17). Under **Iterative Replacement**, the evolver distills $\mathcal{K}_n =$ Evolver$(\emptyset, \mathcal{T}^{(n)})$ purely from the new trajectories, discarding all prior skills (line 19). In the Offline regime, the bank is frozen after initialisation ($\mathcal{K}_n = \mathcal{K}_{n-1}$, line 15).

Crossing these axes yields five concrete variants: Offline (static bank), Off2On with Evolution, Off2On with Iterative Replacement, Pure Online with Evolution, and Pure Online with Iterative Replacement. We compare these empirically in 4.

Finally we see that the Off2On + Evolution variant works best across all settings and is thus the variant that we generally refer to as HExA in the rest of the paper.

A.4 Cross-Task Skill Transfer

The skill banks produced by HExA encode task-family-specific strategies, but the underlying principles are often more general: momentum conservation, lever mechanics, collision geometry, and similar structural insights recur across task families that share physical or procedural primitives. HExA exploits this by synthesizing a *cross-task skill bank* for a target task family from the evolved banks of one or more source families, with no target-task trajectories required.

Given source banks $\mathcal{K}_{\ell_1}, \ldots, \mathcal{K}_{\ell_S}$ and a textual description of the target task family ℓ^*, the evolver receives all source skills together with the target description and is prompted to:

1. identify which source skills encode principles that are structurally relevant to the target (based on shared physical or procedural primitives),
2. re-ground each selected principle in the entities and mechanics of the target scene, and
3. calibrate reward based on how directly the principle transfers (skills corroborated by multiple source banks receive higher reward).

The synthesised bank $\mathcal{K}_{\ell^*}$ is then injected into the actor's context via the same RETRIEVER mechanism as within-task skills. This enables zero-shot transfer: the actor attempts the target task with the benefit of cross-task skills but without having seen any target-task trajectories.

Importantly, cross-task transfer and within-task skill evolution are complementary. The synthesised bank can serve as the initialisation $\mathcal{K}_0$ for a subsequent within-task HExA run, combining the benefits of transferred knowledge with task-specific refinement.

A.5 Design Principles

Several design choices in HExA merit explicit discussion.

Domain agnosticism. Nothing in the framework above assumes a particular domain. The actor interacts with the environment through a generic tool-call interface; the evolver reads trajectory text and produces structured JSON. The only domain-specific components are the tool suite exposed by the environment and the system prompt describing the task, both of which are inputs to HExA rather than parts of its architecture. We instantiate and evaluate HExA on physics puzzles in §4, but the same framework applies to any setting where an agent must learn from sequential interactions with an environment.

Reward Function Agnostic. As we have shown previously in our experiments, rewards from environment interactions and shaped according to the needs of the user help HExA more. But the evolver can guide skill creation and learning even without it. So this reward agnostic nature of HExA make it widely applicable in domains where even environment reward might not be present. Moreover in cases where there is presence of such rewards and additional feedback, we can plug in any reward function to further guide skill acquisition either guided by users or designed by the LLM evolver itself.

No weight updates. HExA is entirely in-context: neither the actor nor the evolver undergoes any parameter updates. All adaptation occurs through the evolving skill bank, which modifies the actor's behavior by changing the textual context it conditions on. This makes HExA compatible with closed-source, API-served models and avoids the computational cost and catastrophic-forgetting risks of fine-tuning.

Hierarchical skill structure. The skills in $\mathcal{K}$ are hierarchical in two senses. First, they abstract over low-level actions (specific tool calls and parameter values) into high-level strategy descriptions, compressing lengthy trajectories into concise principles. Second, skills are learned iteratively while the agent has access to previously distilled skills, so later skills implicitly build on earlier ones. The evolver may revise a skill in light of new evidence or merge two complementary skills into a more general one, producing a form of meta-learning: the agent learns not just strategies but *how to learn strategies* from interactions.

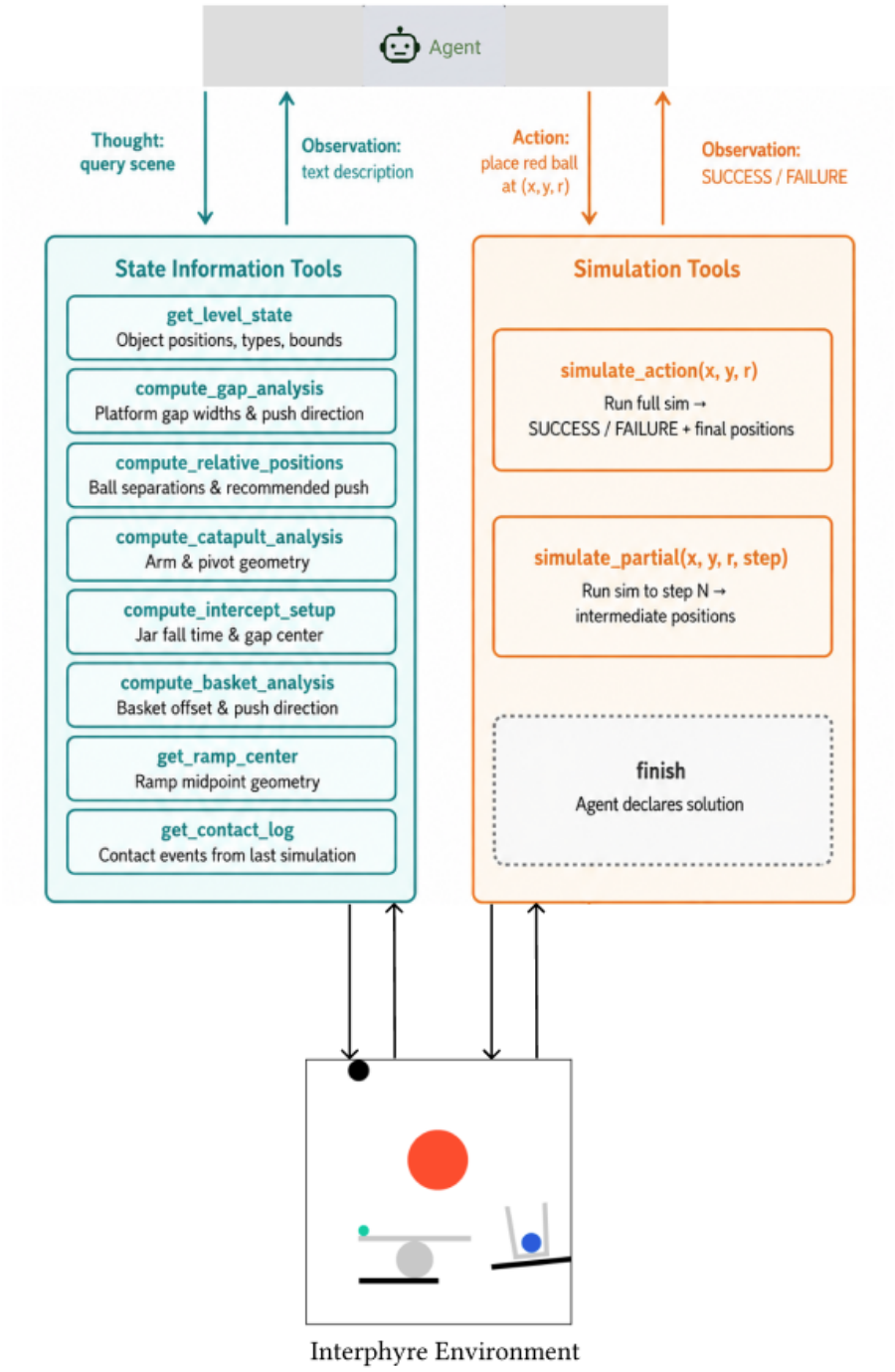


Figure 8: Tool interface between the HEXA agent and the Interphyre environment. The agent issues two classes of tool calls: **State Information Tools** (left) query the scene without advancing the simulation—returning object positions, gap widths, relative ball positions, and level-specific geometry—and **Simulation Tools** (right) place the red ball at (x, y, r) and run the physics engine, returning `SUCCESS`/`FAILURE` with final object positions (`simulate_action`) or intermediate state at a chosen timestep (`simulate_partial`). The `finish` action terminates the episode. Level-specific state tools (`compute_gap_analysis`, `compute_relative_positions`, `get_ramp_center`, etc.) are available only on their respective levels; shared tools are available on all levels. Selected tools presented here. Full tool signatures and usage instructions are provided in Appendix B.

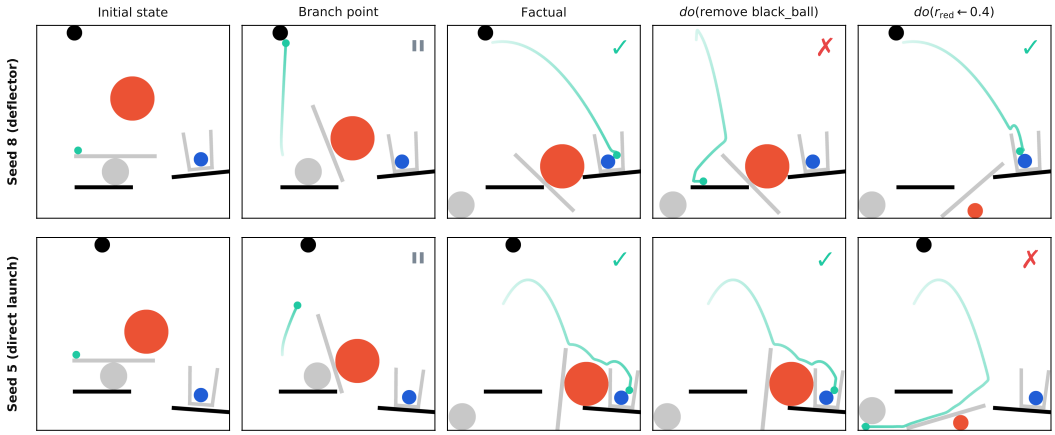


Figure 9: INTERPHYRE's snapshot/restore API. This branches a shared mid-trajectory state into a factual rollout and perturbed alternatives (Section G.4.1). Each row is one seed of the `catapult` level with a different oracle solution strategy. The leftmost column shows the initial scene for reference; the teal curve is the green ball's trajectory, shown up to the branch point in the *Branch point* column and continuing through the outcome in each subsequent column. Top (seed 8, deflector strategy): removing the deflector causes failure; shrinking the red action ball does not. Bottom (seed 5, direct-launch strategy): shrinking the red ball causes failure; removing the deflector does not. The same interventions have opposite causal relevance across seeds, showing that causal relevance is a property of an object relative to the active strategy, not of the object in isolation. See Section G.4.3 for the branching protocol.

A.6 INTERPHYRE: A Testbed for Learning through Experimentation

B Tool Descriptions

This appendix provides complete descriptions of all tools available to the REACT agent. Tools are partitioned into *shared tools* (available on every level) and *level-specific tools* (exposed only on the corresponding puzzle).

B.1 Shared Tools

`get_level_state()`. Returns a structured text description of the current puzzle scene: the name, position, radius (for balls), dimensions and angle (for bars), and dynamic flag of every object; the world bounds ($x, y \in [-5, 5]$); and the level-specific success condition.

`simulate_action(x, y, radius)`. Places the red ball at (x, y) with the given `radius` $\in [0.1, 2.0]$ and runs the full physics simulation (up to 2,000 steps at 60 Hz) to completion. Returns the outcome (`SUCCESS`/`FAILURE`), total step count, and final positions and velocities of all objects. Before launching the simulation the toolkit performs *pre-simulation validation*: if the placement violates world bounds, overlaps an existing ball, or intersects a platform, the call returns a detailed error specifying the minimum adjustment required rather than consuming a simulation attempt.

`simulate_partial(x, y, radius, stop_step)`. Identical to `simulate_action` but halts after `stop_step` physics steps (or earlier if the success condition is satisfied). Returns object positions and velocities at the chosen checkpoint, enabling mid-simulation diagnostics without committing to a full run.

`get_contact_log()`. Returns timestamped collision events from the most recent simulation: a list of `(step, object_A, object_B)` entries (capped at 20; a notice is appended if more exist), or `"No contact events recorded"` if no contacts occurred. Must be called after `simulate_action` or `simulate_partial`.

`finish(x, y, radius).` Submits the agent's final answer: places the red ball at (x, y, r) and ends the episode. The episode is scored as a success if the physics engine confirms the goal condition.

B.2 Tools for Levels

B.2.1 Down to Earth — `compute_gap_analysis()`

Analyses the gaps on each side of the platform that separates the green ball from the floor. Returns:

- platform span (left and right x-coordinates of the platform edges);
- green ball diameter;
- left gap width (distance from left wall to platform left edge) and right gap width (distance from platform right edge to right wall);
- `YES`/`NO` for whether the green ball can fit through each gap;
- a recommendation indicating which side has the larger viable gap, or a `WARNING` if neither gap is wide enough.

B.2.2 Two Body Problem — `compute_relative_positions()`

Analyses the spatial relationship between the green and blue balls. Returns:

- position and radius of each ball;
- horizontal separation Δx, vertical separation Δy, and centre-to-centre distance;
- minimum contact distance (sum of radii);
- relative direction of the blue ball from the green ball (`LEFT`/`RIGHT`).

This information guides placement of the red ball to push the green ball into contact with the blue ball.

B.2.3 Catapult — Four Analysis Tools

The Catapult level exposes four complementary tools to handle its multi-body lever mechanics.

`describe_scene_geometry().` Returns a strategy-neutral geometric inventory of the scene with no prescriptive advice: all balls (position, radius, dynamic flag), all bars (centre, angle, length, dynamic flag), and all baskets (position, dynamic flag); plus the key pairwise distance (green–blue) and the success condition. Intended as a first-pass survey before committing to a simulation strategy. No arguments.

`simulate_with_trace(x, y, radius, object_names, stop_step).` Places the red ball and runs a full (or truncated) simulation, then returns a per-object kinematic-extrema summary for every name in `object_names` (e.g. `["green_ball", "catapult_arm"]`). The summary per object includes: `peak_y` (maximum height reached), `min_y` (minimum height), `v_max` (peak speed), $\Delta\mathrm{pos}$ (net displacement), and, for bodies that rotate appreciably, peak angular speed $\omega_{\max}$ and angle range $[\theta_{\min}, \theta_{\max}]$. The most relevant contact events involving the red ball or any traced object are also returned (capped at 15). Used to verify lever mechanics and confirm that the green ball reaches the basket.

`trace_green_ball(x, y, radius).` A lighter trajectory probe than `simulate_with_trace`: only the green ball is sampled. Places the red ball, runs the simulation up to a 600-step ceiling, and returns the green ball's (x, y) waypoints at fixed 30-step intervals together with a start/end/peak summary (start position, end position, Δ, peak y, peak speed). Stops early once the green ball settles (consecutive samples with $|\Delta\mathrm{pos}| < 0.10$) or once the success condition fires. Use this when the only quantity of interest is *where* the green ball travels rather than full per-object kinematics or contact events.

`predict_first_contact(x, y, radius).` A cheap pre-simulation check (at most 90 physics steps, ≈1.5 s of simulated time) that identifies the first object the red ball contacts after release. Returns: placement validity, the name of the first object hit, the step at which contact occurs, approach speed, approximate contact point, and surface normal. Used to verify that the red ball reaches the intended catapult arm before committing to a full simulation.

B.2.4 Falling Into Place — `compute_intercept_setup()`

Computes the intercept geometry for the timed-interception puzzle. Returns:

- green ball position and which platform it rests on (`LEFT`/`RIGHT`);
- blue jar position and travel direction the green ball must take;
- the platform edge the green ball must cross and the gap centre x-coordinate;
- estimated time until the jar falls to platform height (in seconds and in 60 Hz physics steps).

B.2.5 Basket Case — `compute_basket_analysis()`

Analyses the spatial relationship between the green ball and the basket that the agent must avoid. Returns:

- green ball position and radius;
- basket position and scale, including opening half-width;
- purple ground position (if present);
- horizontal distance from the green ball to the basket centre;
- recommended push direction (`LEFT`/`RIGHT`) to deflect the green ball away from the basket.

.

B.2.6 Pass the Parcel — `get_ramp_center()`

Analyses the ramp-and-basket mechanism in the Pass the Parcel puzzle. Returns the centre coordinates (x, y) of the ramp, together with inferred geometry including the top-basket position, bottom-basket position, platform position, ramp angle, and ramp bounds. This information guides placement of the red ball to roll it onto the ramp and interact with downstream mechanisms.

B.2.7 Cliffhanger — `compute_cliffhanger_analysis()`

Analyses the cliffhanger geometry: a vertical green bar standing on a black platform with a ceiling above and the purple ground below. Returns:

- green bar centre, length, thickness, and angle, plus the (x, y) coordinates of its bottom point (resting on the platform) and top point (opposite end);
- black platform extents (left and right x, centre y, top-surface y, thickness);
- ceiling y and purple-ground y where present;
- the bar's bottom-point distance to the platform's left and right edges;
- closer platform edge (`LEFT`/`RIGHT`) — i.e. the edge the bar must fall past — and the width of the falling gap on that side (between the platform edge and the world wall);
- the implied tip direction the bar's top end must rotate (`LEFT`/`RIGHT`) for the bar's centre of mass to cross the closer edge.

B.2.8 Tipping Point — `compute_tipping_point_analysis()`

Analyses the tipping-point geometry: a vertical green bar pinned at its base by a small gray basket, with a static purple wall flush against either the left or right side of the box. Returns:

- green bar centre, length, thickness, and angle, plus the (x, y) coordinates of the free top end and the basket-pinned bottom end;
- gray basket centre (when present);

- purple wall x and its top/bottom y;
- the wall's side relative to the bar (`LEFT`/`RIGHT`);
- horizontal distance from the bar centre to the wall;
- approximate angle the bar must tip through, treated as a rigid stick pivoting at its base, to reach the wall ($\arcsin(\Delta x/L)$ when $\Delta x < L$, else `N/A`);
- suggested tip direction (`LEFT`/`RIGHT`).

C Baseline Implementation Details

This appendix documents the three baselines summarized in Section 4.1: the per-step REACT loop, the Reflexion wrapper (K=2 trials), and the GRPO fine-tune. We run open weight models using vLLM, and Huggingface on upto 4 A100 GPUs with 80GB VRAM each and the Claude Sonnet model using the Anthropic API. Together with the per-level system prompts (Appendix H) and tool descriptions (Appendix B), the contents below are sufficient to reproduce every baseline number reported in Section 4.

C.1 ReAct: Inner Loop

Every method we evaluate runs the same REACT loop on top of the INTERPHYRE tool API. HEXA adds a skill block to the system prompt; Reflexion adds a reflection block; both leave the inner loop otherwise unchanged.

Per-step protocol. At each iteration the agent emits exactly one block of three lines:

```
Thought: <free-form reasoning over the conversation history>
Action: <tool name, drawn from the level's API>
Action Input: <JSON arguments, or empty for nullary tools>
```

The harness parses these three fields, dispatches the tool against the simulator, and appends the returned text as

```
Observation: <tool output>
```

The agent then produces the next `Thought`/`Action`/ `Action Input` block, conditioned on the full conversation so far.

Termination, attempts, and scoring. A trial ends as soon as one of three things happens. (1) The agent emits `Action: finish` with arguments (x, y, r); the harness places the red ball and the simulator runs to completion. (2) The agent emits `Action: simulate_action` with arguments (x, y, r) and the simulator confirms the goal predicate (the named target pair stays in contact for at least 3 seconds, i.e. 180 physics steps at 60 Hz). (3) The iteration count reaches the cap of 25 without either of the above. Each `simulate_action` call counts as a placement attempt; an agent may make up to roughly 24 attempts within a single trial.

Decoding. All baselines share decoding: temperature 0.3, max-new-tokens 700 per agent turn, and a single tool call per turn (parsed greedily from the first `Action` line in the model's reply).

C.2 Reflexion: K=2 Trial Wrapper

Reflexion wraps the same single-trial REACT agent (above) in K=2 trials per seed each trial runs for 12 iterations. Trial 1 runs as a standard REACT loop. If trial 1 fails, the harness calls a separate Claude Sonnet instance to write a verbal reflection, prepends the reflection to the system prompt of trial 2, and runs trial 2 from a fresh 25-iteration budget. The episode is scored a success if either trial succeeds; iteration count and wall-clock time aggregate across the two trials.

Reflection model. The reflection step uses Claude Sonnet 4.6 (matching the actor for our `catapult` run), invoked with a single non-streaming completion. The reflection prompt is held fixed across seeds and levels.

Reflection prompt. The reflector receives the full REACT trajectory of the failed trial (`Thought`, `Action`, `Observation` per step) plus the final observation. Long observations are truncated to the first 300 and last 300 characters with an ellipsis to keep the prompt under context. The prompt template is:

```
SYSTEM:
```

```
You are analyzing a failed attempt at a 2D physics puzzle.

You will receive the full Thought/Action/Observation trajectory from a single failed attempt
     by a ReAct-style agent, followed by the final outcome and any prior reflections
     accumulated on this same task.

Your job: produce a short reflection (<=5 sentences) that the agent will read before its next
     attempt on the SAME task. Cover:
(a) which strategy/approach the agent pursued in this attempt;
(b) the specific kinematic, geometric, or procedural reason it failed (cite concrete
     coordinates, distances, or contact events from the observations);
(c) one concrete different action or strategy to try next -- be specific (object names,
     approximate (x, y, radius), expected mechanism).

Hard rules:
- Do NOT repeat lessons that already appear in the prior reflections list.
- Do NOT re-state the goal or the puzzle rules.
- Output ONLY the reflection text. No preamble, no headers, no markdown.

USER:
Outcome: FAILURE.
Final observation:
{final_observation}

Trajectory:
--- Step 1 ---
Thought: {thought}
Action: {action}
Observation: {observation_truncated}
... (one block per step) ...

## Prior reflections (do not repeat these)
1. {prior_reflection_1}
...

Write the reflection now. Be specific and concise (<=5 sentences).
```

Trial 2 conditioning. The returned reflection text is appended to the level's system prompt under a heading `## Reflexion memory` before trial 2 starts. The actor sees this block exactly once at trial 2's first turn, in addition to the unchanged level system prompt and tool list. Trial 2 then proceeds as a fresh REACT loop with the standard 25-iteration cap.

Empty / failed reflections. If trial 1 already succeeds, no reflection is generated and trial 2 is skipped (we still report the seed as a success but the reported iteration count is just trial 1's). If the reflection model returns an empty string (CLI error or empty output), trial 2 runs without a memory block, which makes it equivalent to a second REACT trial from scratch.

C.3 Direct: One-Shot, No-Tool Baseline

DIRECT removes the REACT loop entirely. The agent never emits a `Thought`/`Action`/`Action Input` block, never calls a simulator-write tool, and never observes the consequence of a placement before committing. It serves as a lower-bound test for how much of each level is solvable purely from the static scene description.

Table 5: DIRECT baseline solve rate (%) across eight INTERPHYRE levels. DIRECT peeks at the level state once and makes a single model call to `finish()`, with no environment interaction. Underlined levels (Pass the Parcel, Catapult) form the harder tier, based on Claude Sonnet performance. †Partial run (Pass the Parcel: 50 seeds; Catapult: user-reported); all other cells 100 seeds.

	Q-7B	Q-14B	O-20B	O-120B	Son.
Cliffhanger	0.0	0.0	0.0	0.0	0.0
Tipping Point	0.0	3.0	8.0	2.0	0.0
Basket Case	0.0	2.0	14.0	13.0	9.0
Down to Earth	18.0	0.0	16.0	3.0	14.0
Two Body	4.0	16.0	13.0	1.0	68.0
Falling in Place	8.0	1.0	3.0	29.0	23.0
Pass the Parcel	0.0	0.0	0.0	3.0	0.0†
Catapult	0.0	0.0	0.0	0.0	2.0†

Per-seed protocol. The harness performs exactly two operations per seed:

1. A single scripted call to `get_level_state` via the same INTERPHYRE toolkit used by REACT. The returned object table (positions, sizes, dynamic flags, and any level-specific geometry) is captured verbatim.
2. One non-streaming Claude Sonnet 4.6 call. The conversation has a level-specific system prompt and a single user message that embeds the scene table from step 1 and demands the final (x, y, r) placement as JSON. The harness parses the JSON, runs `simulate_action` once, and records success/failure.

The model is given no read tools, no probe tools (`simulate_partial`, `get_contact_log`), and no retries. There is one inspection of the scene and one committed placement.

Actor model and decoding. Actor model: Claude Sonnet 4.6 invoked via the `claude -p` CLI with `-max-turns 1` and `-no-session-persistence`, matching the actor used by the REACT and Reflexion baselines on the same level so the comparison isolates the harness, not the model. Temperature 0.3 and `max-new-tokens` 700, identical to REACT.

Answer prompt (no-CoT variant). We use the no-CoT variant of the answer block: the model emits a single `Action`/`Action Input` pair and no preceding reasoning. This shape mirrors the per-iteration output of REACT, which keeps Sonnet's response routine and avoids triggering extended-thinking trajectories that would otherwise inflate latency by an order of magnitude. The exact answer block appended to the user message is:

```
Submit your FINAL red ball placement now using the finish action.

Output EXACTLY this format and NOTHING else (no Thought line, no preamble,
no commentary):
Action: finish
Action Input: {"x": <X>, "y": <Y>, "radius": <R>}

Replace <X>, <Y>, <R> with your chosen floating-point numbers (subject to
the placement constraints above).
```

System prompt. The system prompt is self-contained and does not reuse the REACT level prompts, so the baseline cannot accidentally inherit ReAct framing. It states the world dimensions, gravity direction, placement constraints, and a per-level *elements* block enumerating the dynamic and static objects with one-line descriptions. The full per-level blurbs are in the released code (`react_agent/run_direct_answer_claude.py`).

Termination and scoring. A seed terminates after the single `simulate_action` call. Success is the same simulator predicate used by every other method (target pair in contact for ≥ 3 s). If the model's response cannot be parsed into a valid (x, y, r) tuple, the seed is recorded as a failure with no simulator call (these are extremely rare: the no-CoT format leaves only one syntactic shape to emit).

C.4 GRPO: Weight-Update Baseline

We use Group Relative Policy Optimization (Shao et al., 2024) to fine-tune Qwen-2.5-3B-Instruct on the same two levels reported in Section D (`down_to_earth`, `two_body_problem`) using the VERL-TOOL framework (Jiang et al., 2025). GRPO is included purely as a sample-efficiency comparison for Section D; it is the only method we evaluate that updates model weights.

Setup. Per-step batch size 1, four rollouts per gradient step (n=4), binary $\{0, 1\}$ environment reward (the simulator's success predicate, identical to the scoring rule used by every other method). One *epoch* corresponds to a full pass over the 50 training seeds, i.e. 50 gradient updates. We train for 10 epochs (500 total gradient steps).

Hyperparameters. All runs share the configuration in Table 6; the only difference between the non-skilled and static-skilled variants is `max_prompt_length` (2048 vs. 3072), which is increased to accommodate the skill-bank prefix ($\approx$ 2200 tokens vs. $\approx$ 1400 tokens without skills).

Table 6: GRPO hyperparameters shared across all four runs.

Parameter	Value
Model	Qwen/Qwen2.5-3B-Instruct
Algorithm	GRPO
Training seeds	1–50 (val: 51–100)
Batch size	1 seed / gradient step
Group size n	4 rollouts per prompt
Total epochs	10 (500 gradient steps)
Learning rate	1×10^{-6}
KL loss coefficient	1×10^{-4} (`low_var_kl`)
Entropy coefficient	1×10^{-3}
Temperature	1.0
Max prompt length	2048 (non-skilled) / 3072 (static-skilled)
Max response length	4096 tokens
Max observation length	512 tokens
Max turns	25
GPU memory utilization	0.3
Observation masking	True (observation tokens excluded from loss)
Hardware	1× A100 80 GB

Prompt variants. We evaluate two prompt variants. The *non-skilled* variant uses the level's full system prompt (Appendix H) without any skill block; the *static-skilled* variant pre-pends the same skill bank HExA ends with after its final training pass, baked in at data generation time so the GRPO trainer sees a fixed augmented prompt across all rollouts. The static-skilled variant therefore tests whether the skill bank is also useful as a frozen prefix to gradient-based RL.

Results. Table 7 reports greedy val success rate (`mean@1`, 50 val seeds) at each epoch boundary. Figure 5 visualizes the same data on a log-scaled x-axis to emphasis the low-data region. Three observations follow.

Table 7: GRPO val success rate (greedy `mean@1`, 50 val seeds) per epoch. Epoch 0 is the pre-training baseline (no gradient updates). non-skilled DTE, non-skilled TBP, static-skilled DTE, static-skilled TBP.

Epoch	DTE (ns)	DTE (sk)	TBP (ns)	TBP (sk)
0	6%	10%	4%	4%
1	20%	6%	6%	0%
2	16%	14%	22%	12%
3	18%	4%	20%	22%
4	54%	38%	26%	100%
5	100%	44%	22%	100%
6	100%	100%	50%	100%
7	100%	100%	56%	100%
8	100%	86%	48%	100%
9	100%	100%	50%	100%
10	100%	100%	40%	100%
HExA @ ep. 1	24%		14%	

Compute matching. The figure in Section D marks the matched-sample-budget point at 50 unique seeds (one epoch). At this point, GRPO has performed 50 gradient updates and produced $50 \times 4 = 200$ rollouts; HExA has generated 51 trajectories total (5 warm-start + 17×3 training). We compare on *unique seeds seen*, not on rollouts or gradient steps, since counting rollouts or gradient steps would artificially advantage the gradient-based method.

C.5 Seed Ranges per (method, level) Cell

For full reproducibility, Table 8 lists the integer seed range used by each (method, level) cell reported in Section 4. Seeds are passed directly to the simulator's level generator and determine the randomised positions, masses, and dimensions of the level's objects (see Section A.6). All cells use exactly 50 seeds.

Table 8: Seed range used by each (method, level) cell. All within-level cells share the evaluation pool 6–55, so comparisons across REACT, Reflexion, and HEXA on a given level are paired on identical seeds.

Method	Level	Seed range
REACT baseline	every level	6–55
Reflexion (K=2)	`catapult`	6–55
HEXA (within)	every level	6–55 (warm-start 1–5 held out)
HEXA contrastive	Qwen-7B `DTE`, `TBP`	6–55
Cross-level → catapult	`catapult`	6–55
Cross-level → FIP / TBP	`falling_into_place`, `two_body_problem`	6–55
GRPO training	Qwen-2.5-3B `DTE`, `TBP`	1–50

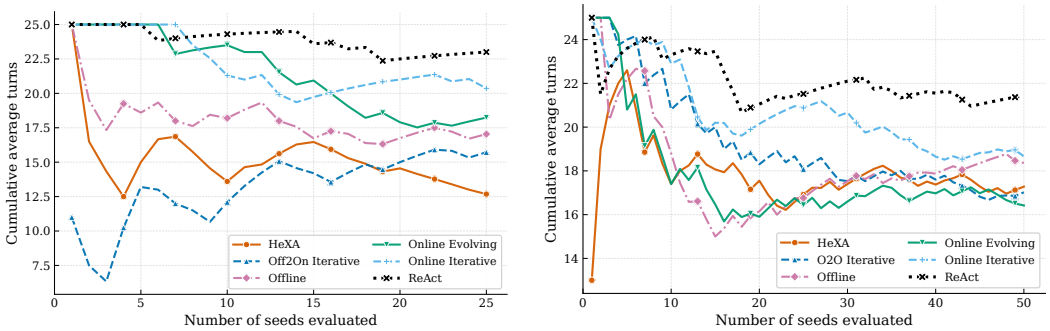
The seed scheme has two properties worth flagging.

Within-level: paired comparison. Every method on a given level is evaluated on the same 50 seeds, so the REACT, Reflexion, and HEXA numbers in Tables 3, 12, 10, and 11 are paired contrasts on identical instances rather than independent samples. HEXA additionally uses seeds 1–5 as a warm-start batch to construct the offline bank $\mathcal{K}_0$; these seeds are not part of the evaluation pool and never appear in any reported solve-rate cell.

Round-by-round: bank has not yet been distilled on the attempted seed. Within HEXA's online rounds, the agent attempts a fresh batch of seeds (k=3 or k=5 per round) using the current bank, and only after that round is the evolver allowed to update the bank from those trajectories. Every reported HEXA attempt is therefore made with a bank that has not yet been distilled on the seed being attempted, even though the bank is trained from earlier seeds on the same level.

Cross-level: same-pool comparison against target baseline. For cross-level transfer the synthesised target bank is evaluated on the same 50 seeds as the target-level REACT baseline, so the cross-level numbers in Table 13 are paired with the corresponding baseline numbers on identical instances. The synthesis itself never sees a target-level trajectory at any seed, so the comparison remains strictly zero-shot at the trajectory level.

D Additional Details of Experiments and Results

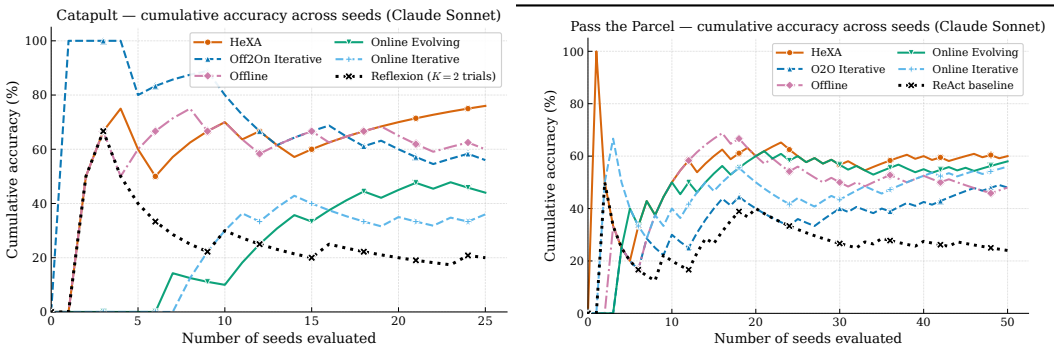


(a) Catapult **(b)** Pass the Parcel

Figure 10: Cumulative average turns per seed for every HEXA variant against the baseline (Claude Sonnet); lower is more efficient. On both levels the Off2On Evolving configuration converges to the lowest cost per seed, while the iterative and pure-online variants remain higher, indicating that HEXA's gains come from guiding search more efficiently with accumulated experience rather than from spending more computation per instance.

Table 9: REACT baseline solve rate (%) across eight INTERPHYRE levels. Underlined levels (Pass the Parcel, Catapult) form the harder tier, based on Claude Sonnet performance. †Evaluated on 50 seeds; ‡25 seeds; all others 100 seeds. — = not evaluated.

	Rand	Q-7B	Q-14B	O-20B	O-120B	Son.
Cliffhanger	0	2.0	6.0	30.5	64.0	100‡
Tipping Point	3	24.0	42.0	80.0	90.1	96‡
Basket Case	4	8.0	10.0	14.0	33.0	100‡
Down to Earth	8	62.0	72.0	35.0	72.5	100
Two Body	14	18	24.0	26.0	77.8	100
Falling in Place	1	20.0	52.0	58.0	100	100†
Pass the Parcel	0	7.8	9.8	16	22	24†
Catapult	0	0.0	0.0	0.0	0.0	8†

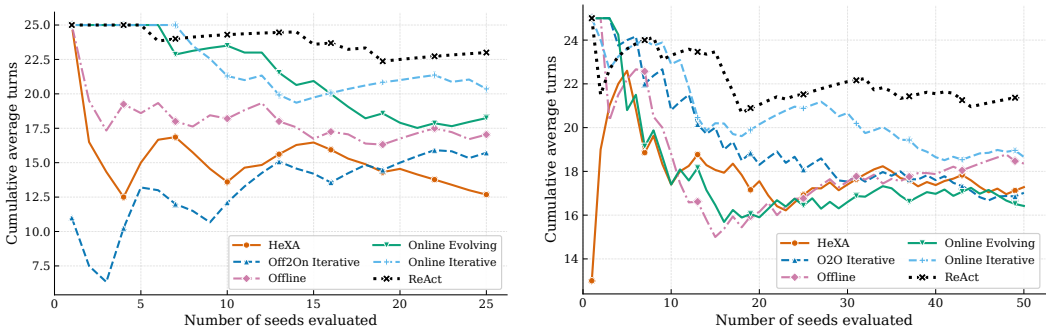


(a) Catapult **(b)** Pass the Parcel

Figure 11: Cumulative solve rate as seeds are evaluated, for every HEXA variant against the strongest agent-side baseline (Claude Sonnet). Each curve is the running mean over the seeds seen so far. On both levels the Off2On Evolving configuration finishes highest—76% on `catapult` and 60% on `pass_the_parcel`—and stays consistently above the other variants and baselines throughout the sweep, so the ranking is not an artifact of a few lucky seeds.

The gains extend to open-weight solvers We also uniformly notice that gains from HEXA is not limited to a frontier-model agent. It also improves smaller open-weight agents when the same skill-evolution mechanism is applied. Table 10 shows improvements for every open-weight model-level pair tested.

For Qwen-2.5-3B, HEXA improves `down_to_earth` from 8.0% to 24.0% and `two_body_problem` from 6.0% to 14.0%. For Qwen-2.5-7B, it improves `down_to_earth` from 62.0% to 72.0% and `two_body_problem` from 18.0% to 34.0%. The largest open-weight



(a) Catapult **(b)** Pass the Parcel

Figure 12: Cumulative average turns per seed for every HEXA variant against the baseline (Claude Sonnet); lower is more efficient. On both levels the Off2On Evolving configuration converges to the lowest cost per seed, while the iterative and pure-online variants remain higher, indicating that HEXA's gains come from guiding search more efficiently with accumulated experience rather than from spending more computation per instance.

Table 10: HeXA on open-weight solvers, 50 training/evaluation seeds per cell. Qwen-2.5 runs use Online-Evolving configuration (k=3 seeds per round, 17 rounds). GPT-OSS-120B uses the same Off2On Evolving configuration. Baselines are the same agent without any skill injection. Per-seed cumulative trajectories are unchanged from the per-seed metric reported in Figure 13.

Level	Method	Acc. (%)	Avg Iters (max 25)	Δ vs. baseline
Down to Earth	Qwen-2.5-3B REACT baseline	8.0	23.8	—
	Qwen-2.5-3B HEXA	**24.0**	18.6	+16.0
	Qwen-2.5-7B REACT baseline	62.0	12.5	—
	Qwen-2.5-7B HEXA	**72.0**	12.6	+10.0
Two Body Problem	Qwen-2.5-3B REACT baseline	6.0	24.0	—
	Qwen-2.5-3B HEXA	**14.0**	21.1	+8.0
	Qwen-2.5-7B REACT baseline	18.0	22.2	—
	Qwen-2.5-7B HEXA	**34.0**	18.9	+16.0
Catapult	GPT-OSS-120B REACT baseline	0.0	25.0	—
	GPT-OSS-120B HEXA	**54.0**	16.2	+54.0

gain occurs on `catapult`: GPT-OSS-120B scores 0.0% under the REACT baseline but reaches 54.0% with HEXA. These results show that the method is not merely exploiting Claude Sonnet's stronger prior knowledge; the skill bank provides a general in-context adaptation mechanism that can lift weaker agents as well.

Incontext RL method HExA vs. Gradient-Based LLM RL method GRPO Figure 5 shows validation accuracy across 6 GRPO epochs for two variants: one where agent has access to no skills and other where a previously evolved skill bank using HExA is baked into the prompt at data-generation time). With sufficient training, GRPO's direct optimisation on the exact environment reward dominates: `down_to_earth` reaches 100% by epoch 5 (GRPO) and epoch 6 (GRPO w. evolved skills), while `two_body_problem` converges to 100% by epoch 4 for (GRPO w. evolved skills) and plateaus around 40–50% for (GRPO).

To situate HEXA in the broader RL landscape, we train the same Qwen-2.5-3B model on the same two levels with Group Relative Policy Optimization (GRPO) (Shao et al., 2024), using batch size 1, n=4 rollouts per step, and a binary $\{0, 1\}$ reward from the environment. With 50 training seeds and batch size 1, one epoch of GRPO corresponds to 50 gradient updates — the same number of unique puzzle instances seen by HEXA in a single pass. Figure 5 shows val accuracy across 6 epochs for two variants: GRPO and (GRPO w. skills) (evolved skill bank baked into the prompt at data-generation

Table 11: Reward-signal ablation on Qwen-2.5-7B, 50 seeds per cell. We compare full HExA distillation (reward-weighted, two-pass) against a contrastive-only variant that drops the reward from Eq. 6. The ReAct baseline is the same Qwen-2.5-7B agent with no skill bank.

Level	Method	Acc. (%)	Avg Iters
Down to Earth	ReAct baseline	62.0	12.5
	HExA (no reward)	64.0	14.8
	HExA (reward-weighted)	**72.0**	12.6
Two Body Problem	ReAct baseline	18.0	22.2
	HExA (no reward)	26.0	22.2
	HExA (reward-weighted)	**34.0**	18.9

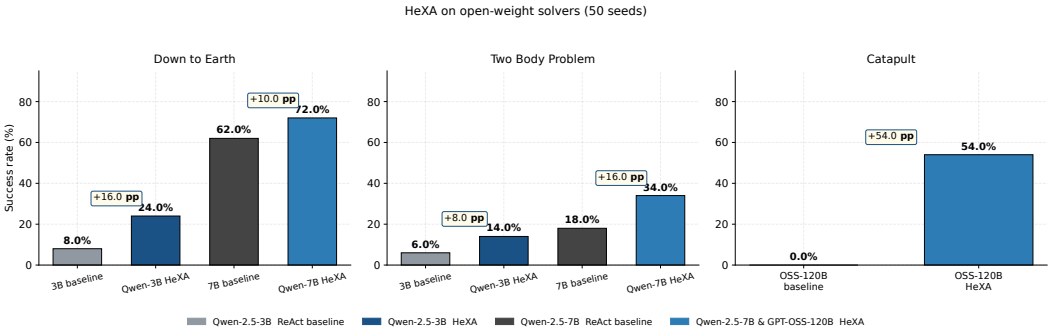


Figure 13: HExA shows consistent improvement on small open LLMs as well.

time). At the matched sample budget (50 unique seeds, vertical dashed line), GRPO non-skilled reaches 20% on `down_to_earth` and 6% on `two_body_problem`, compared to HExA's 24% and 14% respectively. This gap is consistent with the established finding that in-context methods are more sample-efficient in new domains: skills and strategies distilled from early trajectories are immediately available to subsequent episodes via context injection, whereas GRPO must accumulate gradient signal over many rollouts before policy weights encode the same knowledge.

However, given sufficient training, GRPO's exact environment reward eventually dominates: `down_to_earth` reaches 100% by epoch 5 (GRPO) and epoch 6 (GRPO w. skills), while `two_body_problem` peaks at 56% and plateaus around 40–50% for GRPO and converges to 100% by epoch 4 for (GRPO w. skills). The static-skilled variant additionally exhibits a pronounced epoch-1 dip (6% on DTE, 0% on TBP) before recovering, consistent with the model needing several epochs to re-adapt its generation format to the longer skill-augmented prompt ($\sim$2200 vs. $\sim$1400 tokens) before it can leverage the injected skills. Together, these results confirm that HExA's in-context skill evolution is the preferred approach when the sample budget is limited to a small size and interaction or experimentation budget is limited, while gradient-based RL with exact reward remains a strong option to further bootstrap learning after in-context warm-start.

Table 12: HEXA configuration comparison on `pass_the_parcel` (Claude Sonnet, 50 seeds for every method, sorted by accuracy). The five configurations cross two design axes: initialisation (*Offline* / Off-to-Online *(Off2On)* / pure *Online*) and per-round update (*Static* / *Iterative* re-distillation / *Evolving* merge). The headline configuration — Off2On Evolving with x=3 seeds per round — is the one we refer to simply as HEXA elsewhere in the paper. Per-seed cumulative trajectories are shown in Figure 11b. The DIRECT baseline (one scripted scene read + one Claude call, no simulator feedback) is reported below REACT for reference; its *Avg Iter* column reports the single committed model call per seed. Reflexion (K=2 trials with verbal self-reflection between trials) is reported below REACT; the placeholder cells (–) will be filled in once the full 50-seed run completes.

Init	Update	Acc. (%)	Succ.	Fail	Avg Iter
Off2On	Evolving (x=3) (HeXA)	**60.0**	30	20	17.3
Online	Evolving (x=3)	58.0	29	21	16.4
Online	Iterative	56.0	28	22	18.7
Off2On	Iterative	48.0	24	26	17.0
Offline	Static	48.0	24	26	18.4
REACT baseline		24.0	12	38	21.3
Reflexion (K=2 trials)		16	8	42	22.4
DIRECT (one-shot, no tools)		0.0	0	50	1.0

Table 13: Cross-level skill transfer results. For each *target* level we report (i) the source skill banks consumed by $\mathcal{M}_\theta$, with their per-bank skill counts; (ii) the size of the synthesised target bank (n_k skills / n_μ mistakes); (iii) the agent that solves the target with *only* the synthesised bank injected; (iv) the resulting target success rate; and (v) the matched REACT baseline. No target-level trajectories are seen at any stage of synthesis.

Target	Levels of Skills	Solver	Acc. (%)	Baseline (%)	Δ
Catapult	DTE, TBP, PTP	Sonnet	**44.0** (22/50)	8.0 (4/50)	+36.0
Falling Into Place	DTE	Qwen-7B	**32.0** (16/50)	20.0 (10/50)	+12.0
Two Body Problem	DTE	Qwen-7B	**34.0** (17/50)	18.0 (9/50)	+16.0

E Working Example: ReAct vs. HEXA on Level Catapult of InterPhyre

This appendix walks through a single Interphyre seed — `catapult` seed 45 — on which the REACT baseline fails and HEXA succeeds, with both agents using Claude Sonnet 4.6 as the actor. We reproduce, in order: the system prompt and full Thought/Action/Observation transcript for REACT (failure, 25 iterations); the round-14 evolving skill bank that the HEXA agent has access to; the system prompt and full transcript for HEXA (success in 6 iterations, final placement $(x{=}0.3, y{=}0.9, r{=}1.5)$). The two prompts differ only in the LEARNED PHYSICS SKILLS block injected by the evolver ρ ahead of the user prompt; the tool list, format instructions, and kickoff message are identical, isolating the contribution of the skill bank.

E.1 Sparsified simulation frames (catapult seed 45)

Figure 2 shows seven sparsely-sampled frames from each agent's final accepted simulation, in left-to-right time order. The top row is REACT: after 25 iterations the agent never produces a placement that brings the green ball into contact with the blue ball. The bottom row is HEXA's 6-iteration solve at $(x{=}0.3,\ y{=}0.9,\ r{=}1.5)$: the heavy red ball lands on the right end of the catapult arm, the lever flings the green ball over the ceiling-blocker, and the green ball settles into the basket against the blue ball.

E.2 REACT system prompt (catapult seed 45)

```
============================================================
  SYSTEM PROMPT
============================================================
You are an expert physics reasoning agent solving a 2D physics puzzle. You have access
  to a physics simulator and can test your ideas before submitting a final answer.

**Puzzle: Catapult**

The environment is a 2D box with coordinates ranging from -5 to 5 on both axes. Gravity
  pulls objects downward.

**Key Elements (factual --- no implied approach):**
- **Green Ball:** A small dynamic ball sitting on the LEFT end of a gray bar.
- **Gray Bar (Catapult Arm):** A dynamic lever resting on a gray ball (pivot). The green
   ball sits on its left end.
- **Gray Ball (Pivot):** A dynamic ball acting as the fulcrum. It sits on the left black
   platform.
- **Black Ball (Ceiling Blocker):** A static ball near the top of the scene.
- **Black Platform (Left):** A static horizontal platform on the left side.
- **Black Ledge (Right):** A static (possibly angled) platform on the right side.
- **Basket (Gray):** A dynamic basket sitting on the right ledge.
- **Blue Ball (Target):** A dynamic ball inside the basket.

Use more radius for better energy (r>1)
**The Goal:**
Place ONE Red Ball somewhere in the box so that, once the simulation runs, the green
  ball contacts the blue ball for at least 3 seconds. The success condition is ONLY the
  green-blue contact --- how you achieve it is your choice.

**Placement Constraints:**
- The red ball must be completely inside the box: -5 + radius <= x <= 5 - radius, -5 +
  radius <= y <= 5 - radius.
- The red ball must NOT overlap with any existing object at t=0.
- 0.1 <= radius <= 2.0


You have access to the following tools to interact with the physics simulation:

1. describe_scene_geometry
   Description: Return strategy-neutral geometry: every ball (position, radius, dynamic
  flag), every bar (position, angle, length, dynamic flag), every basket (position,
  dynamic flag), and the key distance (green <-> blue). No prescriptive advice; you
  interpret the layout to form a strategy.
   Arguments: None
   Usage: Action: describe_scene_geometry


1. get_level_state
   Description: Get the current level layout including all object positions, sizes, and
  properties.
   Arguments: None
   Usage: Action: get_level_state

2. simulate_action
   Description: Place a red ball at (x, y) with the given radius and run the full
  physics simulation to completion. Returns whether the goal was achieved, final
  positions of all objects, and total simulation steps. If the placement is invalid (out
   of bounds or overlaps), returns a detailed error with how far to move the ball.
   Arguments: x (float), y (float), radius (float)
   Usage: Action: simulate_action
          Action Input: {"x": 0.5, "y": 4.0, "radius": 0.6}

3. get_contact_log
   Description: After running a simulation, returns the contact events: which objects
  touched and when.
   Arguments: None
   Usage: Action: get_contact_log

4. simulate_partial
   Description: Place a red ball and run the simulation only up to the specified step.
  Returns object positions and velocities at that point. Useful for observing
  mid-simulation dynamics.
   Arguments: x (float), y (float), radius (float), stop_step (int)
   Usage: Action: simulate_partial
          Action Input: {"x": 0.5, "y": 4.0, "radius": 0.6, "stop_step": 50}

6. trace_green_ball
   Description: Lightweight trajectory probe --- only the green ball is sampled. Places a
```

```
  red ball, runs the simulation, and returns the green ball's (x, y) waypoints at fixed
  step intervals plus start/end/peak summary. Stops early once the green ball comes to
  rest (capped at ~600 steps). Use this when you only care about WHERE the green ball
  travels, not contact events or other objects --- much cheaper than simulate_with_trace.
   Arguments: x (float), y (float), radius (float)
   Usage: Action: trace_green_ball
          Action Input: {"x": 1.2, "y": 3.5, "radius": 0.6}

7. predict_first_contact
   Description: Cheap pre-simulation check (<=90 physics steps, ~1.5s of sim time). Runs
  just long enough to find the FIRST object the red ball touches after it is released,
  and reports: placement validity, the other object's name, the step of impact, approach
   speed, approximate contact point, and surface normal. Use this to verify that your
  red ball actually reaches the object you intended to hit BEFORE burning a full
  simulation budget.
   Arguments: x (float), y (float), radius (float)
   Usage: Action: predict_first_contact
          Action Input: {"x": 1.2, "y": 3.5, "radius": 0.6}


9. finish
   Description: Submit your final answer. Use this when you are confident in your
  solution.
   Arguments: x (float), y (float), radius (float)
   Usage: Action: finish
          Action Input: {"x": 0.5, "y": 4.0, "radius": 0.6}

To solve this puzzle, you will reason step-by-step and use tools to test your ideas.

At each step, you MUST follow this exact format:

Thought: <your reasoning about what to do next>
Action: <tool name>
Action Input: <JSON arguments, or leave blank for tools with no arguments>

After you take an action, you will receive:

Observation: <result from the tool>

Then you continue with another Thought/Action cycle.

When you are confident in your answer, use the "finish" tool to submit it.

Important rules:
- Always start with a Thought before taking an Action.
- Only call ONE tool per step.
- Parse observation results carefully before your next thought.
- You can simulate multiple different actions to compare results.
- Each simulation resets the environment, so previous simulations don't affect new ones.


============================================================
  USER PROMPT
============================================================
Solve the Catapult puzzle. You have 25 iterations to solve it. The success condition is:
   green_ball must contact blue_ball. Use tools effectively and think about alternate
  approaches if one does not work.
- **ALWAYS call predict_first_contact first** --- it is cheap and tells you if your red
  ball hits the intended object.
- Only if predict confirms the right contact: call trace_green_ball to track green ball.
```

E.3 REACT trajectory (catapult seed 45, 25 iterations, FAILURE)

```
--- Step 1 ---
Thought:
I need to first understand the scene layout before planning my approach.

Action:
describe_scene_geometry({})

Observation:
=== Scene Geometry ===
World bounds: x in [-5, 5], y in [-5, 5]

Balls:
  green_ball: pos=(-3.06, -0.57), r=0.20, dynamic=True
  blue_ball: pos=(3.50, -2.39), r=0.44, dynamic=True
  black_ball: pos=(-2.28, 4.60), r=0.40, dynamic=False
  gray_ball: pos=(-1.14, -1.67), r=0.70, dynamic=True
```

```
Bars:
  ledge: pos=(3.50, -3.23), angle=6.2deg, len=3.02, dynamic=False
  black_platform: pos=(-1.74, -2.47), angle=0.0deg, len=3.00, dynamic=False
  gray_platform: pos=(-1.14, -0.87), angle=0.0deg, len=4.25, dynamic=True

Baskets:
  basket: pos=(3.50, -3.03), dynamic=True

Key distances:
  green<->blue: 6.81

Success condition: green_ball <-> blue_ball for 3.0s
==================================================

--- Step 2 ---
Thought:
Let me analyze the scene. The catapult arm (gray_platform) is a dynamic bar at (-1.14,
  -0.87) with the green ball sitting on its left end at (-3.06, -0.57). The gray ball
  pivot is at (-1.14, -1.67). The blue ball is in the basket at (3.50, -2.39).

To launch the green ball toward the blue ball, I need to drop a red ball on the RIGHT
  end of the catapult arm to push it down and launch the green ball upward and to the
  right. The right end of the catapult arm would be around (1.0, -0.87) since the arm is
   4.25 long centered at (-1.14, -0.87). Let me drop a red ball above the right end of
  the catapult arm.

Action:
predict_first_contact({'x': 1.5, 'y': 2.0, 'radius': 0.5})

Observation:
=== Predicted First Contact ===
placement_valid: True
red_ball drop: x=1.500, y=2.000, radius=0.500
first_contact_object: ledge
contact_step: 62
approach_speed: 9.055
contact_point (approx midpoint): (2.487, -3.269)
contact_normal (red->other inverted): (-0.999, -0.034)

--- Step 3 ---
Thought:
The red ball at (1.5, 2.0) fell to the ledge, not the catapult arm. I need to aim for
  the right end of the catapult arm. The catapult arm is centered at (-1.14, -0.87) with
   length 4.25, so its right end is around (-1.14 + 4.25/2, -0.87) = (0.99, -0.87). I
  should drop the red ball above the right end of the arm, around x=1.0, y=1.0.

Action:
predict_first_contact({'x': 1.0, 'y': 1.0, 'radius': 0.4})

Observation:
=== Predicted First Contact ===
placement_valid: True
red_ball drop: x=1.000, y=1.000, radius=0.400
first_contact_object: gray_platform
contact_step: 28
approach_speed: 2.006
contact_point (approx midpoint): (-0.074, -0.470)
contact_normal (red->other inverted): (0.940, 0.342)

--- Step 4 ---
Thought:
The red ball at (1.0, 1.0) hits the gray_platform (catapult arm). Let me run a full
  simulation with trace to see if this launches the green ball toward the blue ball.

Action:
simulate_with_trace({'x': 1.0, 'y': 1.0, 'radius': 0.4, 'object_names': ['green_ball',
  'blue_ball', 'catapult_arm']})

Observation:
=== Simulation Result (steps: 2000) ===
Success: False
Note: unknown object_names: catapult_arm

Contacts (red_ball + traced objects):
  step~  0: gray_platform <-> green_ball
  step~  0: basket <-> blue_ball
  step~  8: basket <-> blue_ball
  step~ 13: basket <-> blue_ball
  step~ 27: gray_platform <-> red_ball
  step~ 32: gray_platform <-> red_ball
```

```
  step~ 51: basket <-> blue_ball
  step~ 82: bottom_wall <-> red_ball
  step~105: bottom_wall <-> red_ball
  step~110: bottom_wall <-> red_ball
  step~114: black_platform <-> green_ball
  step~132: black_platform <-> green_ball
  step~190: black_platform <-> green_ball
  step~229: bottom_wall <-> green_ball
  step~247: bottom_wall <-> green_ball
  ... +3 more

Object summary (key kinematic extrema):
  green_ball: y_peak=0.67, y_min=-4.78, v_max=7.27, Deltapos=(-1.72, -4.20)
    omega_peak=4.69 rad/s, thetain[-0.7deg, +887.4deg]
  blue_ball: y_peak=-2.40, y_min=-2.52, v_max=1.31, Deltapos=(-0.25, -0.12)
    omega_peak=1.37 rad/s, thetain[+0.0deg, +33.1deg]

--- Step 5 ---
Thought:
The green ball only went up to y=0.67 and fell back down without reaching the blue ball
  at (3.50, -2.39). I need the green ball to fly further right. I'll try dropping a
  larger red ball from higher above the right end of the catapult arm to give it more
  momentum.

Action:
predict_first_contact({'x': 1.0, 'y': 3.0, 'radius': 0.8})

Observation:
=== Predicted First Contact ===
placement_valid: True
red_ball drop: x=1.000, y=3.000, radius=0.800
first_contact_object: gray_platform
contact_step: 42
approach_speed: 5.428
contact_point (approx midpoint): (-0.098, -0.148)
contact_normal (red->other inverted): (0.841, 0.541)

--- Step 6 ---
Thought:
The red ball hits the gray_platform with higher speed. Let me simulate this to see if
  the green ball gets launched further toward the blue ball.

Action:
simulate_with_trace({'x': 1.0, 'y': 3.0, 'radius': 0.8, 'object_names': ['green_ball',
  'blue_ball']})

Observation:
=== Simulation Result (steps: 2000) ===
Success: False

Contacts (red_ball + traced objects):
  step~  0: gray_platform <-> green_ball
  step~  0: basket <-> blue_ball
  step~  8: basket <-> blue_ball
  step~ 13: basket <-> blue_ball
  step~ 41: gray_platform <-> red_ball
  step~ 44: gray_platform <-> red_ball
  step~ 45: gray_platform <-> green_ball
  step~ 51: basket <-> blue_ball
  step~ 70: ledge <-> red_ball
  step~ 83: bottom_wall <-> red_ball
  step~101: bottom_wall <-> red_ball
  step~102: left_wall <-> green_ball
  step~155: left_wall <-> red_ball
  step~161: green_ball <-> red_ball
  step~167: left_wall <-> green_ball
  ... +16 more

Object summary (key kinematic extrema):
  green_ball: y_peak=2.34, y_min=-4.79, v_max=10.22, Deltapos=(+2.41, -4.21)
    omega_peak=10.75 rad/s, thetain[-321.0deg, +975.4deg]
  blue_ball: y_peak=-2.40, y_min=-2.52, v_max=1.31, Deltapos=(-0.25, -0.12)
    omega_peak=1.37 rad/s, thetain[+0.0deg, +33.1deg]

--- Step 7 ---
Thought:
The green ball peaked at y=2.34 and moved right by 2.41 units but didn't reach blue_ball
   at (3.50, -2.39). I need more force and a better angle. Let me try a larger ball
  dropped from higher to hit the far right end of the catapult arm harder, ideally
  giving the green ball more rightward velocity.
```

```
Action:
predict_first_contact({'x': 0.9, 'y': 4.5, 'radius': 1.0})

Observation:
INVALID placement:
  - Top edge (y+radius = 5.5000) exceeds ceiling (5).
placement_valid=False; no contact check performed.

--- Step 8 ---
Thought:
The radius is too large for that height. Let me try dropping from y=3.5 with radius 1.0,
   which should fit within bounds.

Action:
predict_first_contact({'x': 0.9, 'y': 3.5, 'radius': 1.0})

Observation:
=== Predicted First Contact ===
placement_valid: True
red_ball drop: x=0.900, y=3.500, radius=1.000
first_contact_object: gray_platform
contact_step: 44
approach_speed: 6.152
contact_point (approx midpoint): (-0.155, -0.018)
contact_normal (red->other inverted): (0.784, 0.621)

--- Step 9 ---
Thought:
Good, the red ball hits the gray_platform with speed 6.152. Let me simulate this to see
  if the green ball gets launched further toward the blue ball.

Action:
simulate_with_trace({'x': 0.9, 'y': 3.5, 'radius': 1.0, 'object_names': ['green_ball',
  'blue_ball']})

Observation:
=== Simulation Result (steps: 2000) ===
Success: False

Contacts (red_ball + traced objects):
  step~  0: gray_platform <-> green_ball
  step~  0: basket <-> blue_ball
  step~  8: basket <-> blue_ball
  step~ 13: basket <-> blue_ball
  step~ 43: gray_platform <-> red_ball
  step~ 45: gray_platform <-> red_ball
  step~ 46: gray_platform <-> green_ball
  step~ 51: basket <-> blue_ball
  step~ 72: ledge <-> red_ball
  step~ 80: bottom_wall <-> red_ball
  step~ 91: left_wall <-> green_ball
  step~100: bottom_wall <-> red_ball
  step~159: left_wall <-> red_ball
  step~160: gray_ball <-> green_ball
  step~166: left_wall <-> green_ball
  ... +8 more

Object summary (key kinematic extrema):
  green_ball: y_peak=3.29, y_min=-1.99, v_max=9.19, Deltapos=(-1.72, -1.41)
    omega_peak=14.31 rad/s, thetain[-0.7deg, +1148.2deg]
  blue_ball: y_peak=-2.40, y_min=-2.52, v_max=1.31, Deltapos=(-0.25, -0.12)
    omega_peak=1.37 rad/s, thetain[+0.0deg, +33.1deg]

--- Step 10 ---
Thought:
The green ball keeps going left/up rather than right toward the blue ball at (3.50,
  -2.39). I need to hit the far right end of the catapult arm (around x=0.985) to get
  more rightward torque. Let me try a smaller ball placed further right and close above
  the arm level to hit the arm's tip.

Action:
predict_first_contact({'x': 1.5, 'y': 0.5, 'radius': 0.3})

Observation:
=== Predicted First Contact ===
placement_valid: True
red_ball drop: x=1.500, y=0.500, radius=0.300
first_contact_object: bottom_wall
contact_step: 62
approach_speed: 2.025
contact_point (approx midpoint): (0.750, -4.825)
```

```
contact_normal (red->other inverted): (0.975, 0.221)

--- Step 11 ---
Thought:
The red ball at (1.5, 0.5) missed the arm entirely. The arm's right end is around
  x=0.985. Previous hits showed contact far from the right tip. Let me try placing
  closer to the arm surface just above the right end tip to get a better catapult
  effect.

Action:
predict_first_contact({'x': 0.85, 'y': 0.2, 'radius': 0.3})

Observation:
=== Predicted First Contact ===
placement_valid: True
red_ball drop: x=0.850, y=0.200, radius=0.300
first_contact_object: gray_platform
contact_step: 20
approach_speed: 0.390
contact_point (approx midpoint): (-0.146, -0.594)
contact_normal (red->other inverted): (0.965, 0.262)

--- Step 12 ---
Thought:
I need to hit the far right end of the catapult arm (right tip at ~x=0.985) to maximize
  the catapult lever effect. Previous hits were landing near the center (-0.1). Let me
  try placing directly above the right tip.

Action:
predict_first_contact({'x': 0.85, 'y': -0.3, 'radius': 0.2})

Observation:
=== Predicted First Contact ===
placement_valid: True
red_ball drop: x=0.850, y=-0.300, radius=0.200
first_contact_object: gray_platform
contact_step: 13
approach_speed: 0.350
contact_point (approx midpoint): (-0.144, -0.677)
contact_normal (red->other inverted): (0.983, 0.185)

--- Step 13 ---
Thought:
The catapult is launching the green ball but it keeps going left instead of right. Let
  me try a different strategy --- knock the blue ball out of the basket and have it roll
  toward the green ball. I'll drop a red ball from above the basket area.

Action:
predict_first_contact({'x': 3.5, 'y': 0.0, 'radius': 0.4})

Observation:
=== Predicted First Contact ===
placement_valid: True
red_ball drop: x=3.500, y=0.000, radius=0.400
first_contact_object: blue_ball
contact_step: 35
approach_speed: 0.632
contact_point (approx midpoint): (3.442, -2.100)
contact_normal (red->other inverted): (0.143, 0.990)

--- Step 14 ---
Thought:
The red ball hits the blue_ball directly. Let me simulate this and trace both blue and
  green balls to see if knocking the blue ball out of the basket causes it to roll
  toward the green ball.

Action:
simulate_with_trace({'x': 3.5, 'y': 0.0, 'radius': 0.4, 'object_names': ['green_ball',
  'blue_ball']})

Observation:
=== Simulation Result (steps: 2000) ===
Success: False

Contacts (red_ball + traced objects):
  step~  0: gray_platform <-> green_ball
  step~  0: basket <-> blue_ball
  step~  8: basket <-> blue_ball
  step~ 13: basket <-> blue_ball
  step~ 34: blue_ball <-> red_ball
  step~ 42: blue_ball <-> red_ball
```

```
  step~ 48: basket <-> blue_ball
  step~ 52: basket <-> red_ball
  step~ 98: bottom_wall <-> green_ball
  step~118: bottom_wall <-> green_ball
  step~141: gray_platform <-> green_ball
  step~142: left_wall <-> green_ball
  step~147: gray_platform <-> green_ball

Object summary (key kinematic extrema):
  green_ball: y_peak=-0.57, y_min=-4.78, v_max=8.04, Deltapos=(-1.72, -4.21)
    omega_peak=9.95 rad/s, thetain[-0.7deg, +591.9deg]
  blue_ball: y_peak=-2.40, y_min=-2.52, v_max=1.69, Deltapos=(-0.26, -0.12)
    omega_peak=1.58 rad/s, thetain[+0.0deg, +32.8deg]

--- Step 15 ---
Thought:
The catapult launches the green ball upward/left, but I need it to reach the blue ball
  at (3.50, -2.39). Let me try dropping a maximum-size ball from as high as possible
  directly on the catapult arm's right tip (x~=0.985) to maximize launch force and
  rightward arc.

Action:
predict_first_contact({'x': 0.95, 'y': 3.8, 'radius': 1.0})

Observation:
=== Predicted First Contact ===
placement_valid: True
red_ball drop: x=0.950, y=3.800, radius=1.000
first_contact_object: gray_platform
contact_step: 46
approach_speed: 6.415
contact_point (approx midpoint): (-0.136, 0.007)
contact_normal (red->other inverted): (0.784, 0.621)

--- Step 16 ---
Thought:
The best rightward launch so far was with r=0.8 from (1.0, 3.0), giving green
  Deltapos=(+2.41 right). I need more force to get the green ball to travel 6.56 units right
   to reach blue at (3.50, -2.39). Let me try r=0.8 from higher up (y=4.0) to get more
  speed while still hitting near the arm's right tip.

Action:
predict_first_contact({'x': 1.0, 'y': 4.0, 'radius': 0.8})

Observation:
=== Predicted First Contact ===
placement_valid: True
red_ball drop: x=1.000, y=4.000, radius=0.800
first_contact_object: gray_platform
contact_step: 49
approach_speed: 6.294
contact_point (approx midpoint): (-0.119, -0.082)
contact_normal (red->other inverted): (0.825, 0.565)

--- Step 17 ---
Thought:
The green ball has been going upward-left rather than reaching the blue ball at (3.50,
  -2.39). Let me simulate the high-speed r=0.9 hit on the catapult arm to see if more
  force gets the green ball further right.

Action:
simulate_with_trace({'x': 0.95, 'y': 3.8, 'radius': 0.9, 'object_names': ['green_ball',
  'blue_ball']})

Observation:
=== Simulation Result (steps: 2000) ===
Success: False

Contacts (red_ball + traced objects):
  step~  0: gray_platform <-> green_ball
  step~  0: basket <-> blue_ball
  step~  8: basket <-> blue_ball
  step~ 13: basket <-> blue_ball
  step~ 46: gray_platform <-> red_ball
  step~ 51: basket <-> blue_ball
  step~ 72: ledge <-> red_ball
  step~ 78: left_wall <-> green_ball
  step~ 90: bottom_wall <-> red_ball
  step~106: bottom_wall <-> red_ball
  step~142: left_wall <-> red_ball
  step~152: green_ball <-> red_ball
```

```
  step~159: bottom_wall <-> red_ball
  step~164: bottom_wall <-> red_ball
  step~180: left_wall <-> green_ball
  ... +15 more

Object summary (key kinematic extrema):
  green_ball: y_peak=1.89, y_min=-4.78, v_max=9.57, Deltapos=(+2.05, -4.21)
    omega_peak=12.02 rad/s, thetain[-0.7deg, +1320.1deg]
  blue_ball: y_peak=-2.40, y_min=-2.52, v_max=1.31, Deltapos=(-0.25, -0.12)
    omega_peak=1.37 rad/s, thetain[+0.0deg, +33.1deg]

--- Step 18 ---
Thought:
The green ball has been launching leftward or not reaching the blue ball. Let me try a
  much larger ball (r=2.0) on the right end of the catapult arm for maximum force to see
   if the green ball can travel the required ~6.5 units to the right.

Action:
predict_first_contact({'x': 0.9, 'y': 2.8, 'radius': 2.0})

Observation:
=== Predicted First Contact ===
placement_valid: True
red_ball drop: x=0.900, y=2.800, radius=2.000
first_contact_object: gray_platform
contact_step: 35
approach_speed: 5.267
contact_point (approx midpoint): (-0.138, 0.116)
contact_normal (red->other inverted): (0.730, 0.684)

--- Step 19 ---
Thought:
Let me simulate the large r=2.0 ball hitting the catapult arm to see if maximum mass
  provides enough force to launch the green ball to reach the blue ball.

Action:
simulate_with_trace({'x': 0.9, 'y': 2.8, 'radius': 2.0, 'object_names': ['green_ball',
  'blue_ball']})

Observation:
=== Simulation Result (steps: 2000) ===
Success: False

Contacts (red_ball + traced objects):
  step~  0: gray_platform <-> green_ball
  step~  0: basket <-> blue_ball
  step~  8: basket <-> blue_ball
  step~ 13: basket <-> blue_ball
  step~ 34: gray_platform <-> red_ball
  step~ 42: gray_platform <-> red_ball
  step~ 48: basket <-> red_ball
  step~ 48: basket <-> blue_ball
  step~ 52: basket <-> blue_ball
  step~ 58: black_platform <-> red_ball
  step~ 62: basket <-> red_ball
  step~ 75: ledge <-> red_ball
  step~ 86: ledge <-> red_ball
  step~ 90: top_wall <-> green_ball
  step~ 94: basket <-> blue_ball
  ... +23 more

Object summary (key kinematic extrema):
  green_ball: y_peak=4.78, y_min=-4.78, v_max=11.52, Deltapos=(-1.72, -4.21)
    omega_peak=12.52 rad/s, thetain[-0.7deg, +2042.6deg]
  blue_ball: y_peak=-2.25, y_min=-2.52, v_max=1.53, Deltapos=(-0.03, -0.10)
    omega_peak=1.94 rad/s, thetain[-10.8deg, +51.9deg]

--- Step 20 ---
Thought:
The green ball keeps launching left and not reaching the blue ball. Let me try a very
  different approach --- drop a precisely-aimed, small red ball from very high above the
  arm's right tip (x~=0.985) to hit it with maximum speed at the tip for a stronger
  rightward catapult.

Action:
predict_first_contact({'x': 0.985, 'y': 4.6, 'radius': 0.2})

Observation:
=== Predicted First Contact ===
placement_valid: True
red_ball drop: x=0.985, y=4.600, radius=0.200
```

```
first_contact_object: bottom_wall
contact_step: 83
approach_speed: 2.711
contact_point (approx midpoint): (0.493, -4.878)
contact_normal (red->other inverted): (0.973, 0.232)

--- Step 21 ---
Thought:
The small ball missed the arm entirely. Let me try a completely different approach ---
  drop the red ball on the basket to knock the basket (with blue ball inside) leftward,
  so the blue ball rolls toward the green ball. The basket is dynamic.

Action:
predict_first_contact({'x': 3.5, 'y': 3.0, 'radius': 0.5})

Observation:
=== Predicted First Contact ===
placement_valid: True
red_ball drop: x=3.500, y=3.000, radius=0.500
first_contact_object: basket
contact_step: 51
approach_speed: 7.452
contact_point (approx midpoint): (3.493, -1.859)
contact_normal (red->other inverted): (-0.003, 1.000)

--- Step 22 ---
Thought:
The red ball hits the basket directly with speed 7.452. Let me simulate to see if the
  basket (dynamic) gets knocked leftward, potentially bringing the blue ball toward the
  green ball.

Action:
simulate_with_trace({'x': 3.5, 'y': 3.0, 'radius': 0.5, 'object_names': ['green_ball',
  'blue_ball', 'basket']})

Observation:
=== Simulation Result (steps: 2000) ===
Success: False

Contacts (red_ball + traced objects):
  step~  0: gray_platform <-> green_ball
  step~  0: basket <-> blue_ball
  step~  7: basket <-> ledge
  step~  8: basket <-> blue_ball
  step~ 13: basket <-> blue_ball
  step~ 50: basket <-> red_ball
  step~ 50: basket <-> blue_ball
  step~ 58: blue_ball <-> red_ball
  step~ 64: basket <-> blue_ball
  step~ 66: basket <-> ledge
  step~ 78: basket <-> red_ball
  step~ 81: blue_ball <-> red_ball
  step~ 98: bottom_wall <-> green_ball
  step~118: bottom_wall <-> green_ball
  step~120: basket <-> blue_ball
  ... +5 more

Object summary (key kinematic extrema):
  green_ball: y_peak=-0.57, y_min=-4.78, v_max=8.04, Deltapos=(-1.72, -4.21)
    omega_peak=9.95 rad/s, thetain[-0.7deg, +591.9deg]
  blue_ball: y_peak=-2.40, y_min=-2.52, v_max=1.36, Deltapos=(-0.25, -0.12)
    omega_peak=1.34 rad/s, thetain[+0.0deg, +34.2deg]
  basket: y_peak=-3.04, y_min=-3.12, v_max=2.17, Deltapos=(-0.01, -0.08)
    omega_peak=2.01 rad/s, thetain[+4.8deg, +9.4deg]

--- Step 23 ---
Thought:
The r=0.8 case sent the green ball rightward (+2.41) with v_max=10.22, suggesting it may
   have traveled further right before bouncing. I need to try a moderate-force hit that
  sends the green ball in a shallower arc to land in the basket at (3.50, -3.03) where
  the blue ball is. Let me try r=0.4 from a moderate height.

Action:
simulate_with_trace({'x': 1.0, 'y': 2.0, 'radius': 0.4, 'object_names': ['green_ball',
  'blue_ball']})

Observation:
=== Simulation Result (steps: 2000) ===
Success: False

Contacts (red_ball + traced objects):
```

```
  step~  0: gray_platform <-> green_ball
  step~  0: basket <-> blue_ball
  step~  8: basket <-> blue_ball
  step~ 13: basket <-> blue_ball
  step~ 36: gray_platform <-> red_ball
  step~ 45: gray_platform <-> red_ball
  step~ 51: basket <-> blue_ball
  step~ 72: ledge <-> red_ball
  step~ 89: bottom_wall <-> red_ball
  step~105: bottom_wall <-> red_ball
  step~108: bottom_wall <-> red_ball
  step~128: bottom_wall <-> green_ball
  step~132: left_wall <-> green_ball
  step~151: bottom_wall <-> green_ball
  step~155: bottom_wall <-> green_ball
  ... +6 more

Object summary (key kinematic extrema):
  green_ball: y_peak=-0.05, y_min=-4.78, v_max=9.58, Deltapos=(-1.71, -4.21)
    omega_peak=6.06 rad/s, thetain[-0.7deg, +498.3deg]
  blue_ball: y_peak=-2.40, y_min=-2.52, v_max=1.31, Deltapos=(-0.25, -0.12)
    omega_peak=1.37 rad/s, thetain[+0.0deg, +33.1deg]

--- Step 24 ---
Thought:
Let me try a partial simulation with r=0.8 at (1.0, 3.0) to track the green ball's
  trajectory and see if it passes near the basket/blue ball region around step 100-150.

Action:
simulate_partial({'x': 1.0, 'y': 3.0, 'radius': 0.8, 'stop_step': 120})

Observation:
Simulation state at step 120:

  green_ball: pos=(-4.689, 1.626), vel=(0.311, -3.514)
  red_ball: pos=(-1.833, -4.175), vel=(-3.941, 0.000)
  blue_ball: pos=(3.246, -2.518), vel=(0.000, 0.000)
  ledge: pos=(3.500, -3.235), vel=(0.000, 0.000)
  basket: pos=(3.493, -3.119), vel=(0.000, 0.000)
  black_ball: pos=(-2.281, 4.600), vel=(0.000, 0.000)
  black_platform: pos=(-1.736, -2.473), vel=(0.000, 0.000)
  gray_ball: pos=(-2.219, -1.668), vel=(-0.876, 0.000)
  gray_platform: pos=(-0.997, -0.228), vel=(0.600, 0.011)

--- Step 25 ---
Thought:
I need to understand the green ball's initial velocity direction right after catapult
  launch. Let me check a partial sim at step 50 with the r=0.8 trigger to see the green
  ball's exact state right after launch.

Action:
simulate_partial({'x': 1.0, 'y': 3.0, 'radius': 0.8, 'stop_step': 50})

Observation:
Simulation state at step 50:

  green_ball: pos=(-3.433, -0.337), vel=(-1.778, 6.657)
  red_ball: pos=(1.084, -0.229), vel=(0.532, -6.311)
  blue_ball: pos=(3.258, -2.519), vel=(-0.584, -0.064)
  ledge: pos=(3.500, -3.235), vel=(0.000, 0.000)
  basket: pos=(3.499, -3.119), vel=(-0.000, 0.000)
  black_ball: pos=(-2.281, 4.600), vel=(0.000, 0.000)
  black_platform: pos=(-1.736, -2.473), vel=(0.000, 0.000)
  gray_ball: pos=(-1.221, -1.669), vel=(-0.713, 0.000)
  gray_platform: pos=(-1.352, -0.853), vel=(-1.090, 0.517)
```

E.4 Round-14 evolving skill bank used by HExA

```
# CATAPULT SKILLS

[cat_ev_6_000]
  Title:
Use Large Radius (>=1.5) for Sufficient Launch Energy
  generation: 10    confidence: 0.90
  source seeds: [44]
  Principle:
    The catapult arm is a lever --- the red ball's mass (proportional tor^3) determines
     ↪ angular momentum
      imparted to the arm. Below r~=1.2, the arm rotates too slowly to launch the green
```

```
      ball across the ~7-unit gap to the basket. r=1.5 is the minimum reliable
      threshold; r=2.0 adds negligible additional range at x=0.5 due to arm rotation
      saturation. Only reduce radius when forced by overlap constraints, accepting
      reduced range.
  When to apply:
Always use r=1.5 as the baseline. Scale down only when overlap constraints require it.
  Example:
r=1.5 at x=0.5, y=0.4 is the canonical primary placement (Seed 44).

[cat_ev_1_001]
  Title:
x~=0.5 Is the Primary Catapult Launch Sweet Spot
  generation: 10    confidence: 0.91
  source seeds: [44]
  Principle:
    Placing the red ball at x=0.5 on the catapult arm produces a consistent rightward
      launch arc. The contact point is on the right side of the pivot, creating
      sufficient lever arm for strong rotation while staying in the geometrically stable
       zone. Only deviate to x=0.3 when a ceiling hit is observed.
  When to apply:
    Always as the first placement attempt. Deviate only when ceiling hits or overlap
      constraints require it.
  Example:
x=0.5, y=0.4, r=1.5 is the canonical primary (Seed 44).

[cat_ev_7_002]
  Title:
y=0.4 Is Default Drop Height; Scale Up to y=0.9 When Arm Sits Higher
  generation: 10    confidence: 0.90
  source seeds: [44]
  Principle:
    When the arm y <= -1.5, y=0.4 gives adequate fall distance for the red ball to build
      approach speed. When the arm sits higher (y > -1.5), increase to y=0.9.
      Critically, y=0.4 also serves as the safe ceiling-escape fallback height --- even
      when using x=0.3 for arc flattening, y=0.4 avoids gray_ball pivot overlap without
      sacrificing arc geometry.
  When to apply:
    Set y=0.4 by default. Use y=0.9 when arm is high. Use y=0.4 even during
      ceiling-escape x-shifts to avoid gray_ball overlap.
  Example:
    Seed 44 arm at y=-1.59: y=0.4 default; ceiling-escape fallback x=0.3, y=0.4 also
      used y=0.4.

[cat_ev_7_003]
  Title:
predict_first_contact Is Essential for Variable-Arm Seeds
  generation: 10    confidence: 0.88
  source seeds: [44]
  Principle:
    The catapult arm's y-position varies across seeds, making the overlap boundary
      unpredictable. predict_first_contact verifies both placement validity and that the
       red ball contacts gray_platform (not another object), confirming the catapult
      mechanism activates. It also catches gray_ball pivot overlaps before expensive
      simulation runs.
  When to apply:
    Before every simulate_with_trace call. If placement_valid=False, adjust position
      before simulating.
  Example:
    Seed 44: predict_first_contact caught x=0.3, y=-0.3, r=2.0 overlapping gray_ball
      before simulation was wasted.

[cat_ev_7_004]
  Title:
Ceiling Blocker Lethality Depends on Its x-Position and Arc Angle
  generation: 10    confidence: 0.87
  source seeds: [44]
  Principle:
    The static black ball near y~=4.6 varies in x-position across seeds (x~=-3.85 to
      x~=-1.14). When the ceiling blocker is at x > -2.5, primary placement causes a
      ceiling hit. When the blocker is at x < -3.5 (far left), x=0.3 launches allow the
      green ball to arc up and even bounce off the top wall before descending into the
      basket --- the blocker is off to the side of the arc.
  When to apply:
    Check ceiling blocker x in describe_scene_geometry. If x > -2.5, ceiling escape is
      needed. If x < -3.5, x=0.3 launches can reach the top wall and still succeed.
  Example:
    Seed 44: black_ball at x=-3.63 --- green ball bounced off top_wall (step 72) then
      landed in basket.

[cat_ev_7_005]
  Title:
```

```
Radius Plateau at x=0.5: r Beyond 1.5 Yields No Additional Range
  generation: 10    confidence: 0.79
  source seeds: [42]
  Principle:
    At x~=0.5, the catapult arm's rotation saturates --- r=1.5 and r=2.0 produce identical
      green ball landing positions. Increasing mass beyond 1.5 at this x cannot convert
      to additional angular momentum because the arm hits its geometric rotation limit.
      Intermediate radii (r=1.1--1.4) also fail to bridge ceiling-range tradeoffs when
      the root cause is launch angle.
  When to apply:
When r=1.5 fails at x=0.5, do NOT try r=2.0. Change x-position instead.
  Example:
    Seed 42: agent tried r=1.1, 1.2, 1.3, 1.4, 2.0 at x=0.5 --- none resolved the
      ceiling-range tradeoff.

[cat_ev_4_005]
  Title:
x-Position Fine-Tunes Landing Range; Bifurcation Zones at x<=-0.3 and x=0.7--1.5
  generation: 10    confidence: 0.82
  source seeds: [42, 44]
  Principle:
    Small x shifts (0.1--0.2 units) cause large changes in green ball landing position.
      The stable launch zone is x~=0.0 to 0.5. Positions x<=-0.3 and x=0.7--1.5 are
      bifurcation zones where trajectory is chaotic and small changes produce
      unpredictable outcomes. Within x=0.1--0.5, landing is predictable and monotonically
       tunable.
  When to apply:
    When adjusting x to change landing range, stay within x=0.0--0.5. Immediately reset
      to x=0.2--0.4 if any attempt lands outside this band.
  Example:
    x=0.3 vs x=0.5 both produce reliable arcs (Seed 44); x=0.9 produces chaotic bounces
      (Seed 42).

[cat_ev_8_007]
  Title:
Three-Tier Fallback Sequence for Failed Primary Placement
  generation: 10    confidence: 0.93
  source seeds: [44]
  Principle:
    When primary (x=0.5, y=0.4, r=1.5) fails: (1) ceiling hit -> immediately try x=0.3,
      y=0.4, r=1.5 (gray-ball-safe first choice); if still ceiling hit, try x=0.3,
      y=-0.3, r=2.0 but verify no overlap first. (2) short range -> increase y to 0.9 or
      shift x to 0.4--0.45. (3) persistent failure after 2+ radius variations at same x ->
       stop tuning, shift x to 0.2--0.3.
  When to apply:
    Immediately when simulate_with_trace shows ceiling hit or short range. Do not
      micro-tune the failed placement.
  Example:
    Seed 44: primary hit ceiling -> x=0.3, y=0.4, r=1.5 succeeded (y=-0.3 options were
      blocked by gray_ball).

[cat_ev_10_008]
  Title:
x=0.3 Is the Ceiling-Escape x-Position; y=0.4 Is the Gray-Ball-Safe Default Height
  generation: 10    confidence: 0.93
  source seeds: [44]
  Principle:
    Shifting x from 0.5 to 0.3 flattens the green ball's launch arc by changing the arm
      contact point --- this x-shift is the primary ceiling-escape mechanism, not lowering
       y. When x=0.3, y=-0.3, r=2.0 is invalid due to gray_ball pivot overlap (the pivot
       at r~=0.70 can be within 2.7 units of large red balls at low y), use x=0.3, y=0.4,
       r=1.5 --- y=0.4 avoids the overlap zone while preserving the arc-flattening from
      the x-shift.
  When to apply:
    When primary (x=0.5) causes ceiling hit, try x=0.3, y=0.4, r=1.5 first. Only attempt
       y=-0.3 if the gray_ball pivot is positioned far from x=0.3.
  Example:
    Seed 44: x=0.3, y=-0.3, r=2.0 overlapped gray_ball; x=0.3, y=0.4, r=1.5 succeeded
      with green ball bouncing off top wall into basket.

[cat_ev_11_009]
  Title:
When Ceiling-Range Tradeoff Is Unsolvable by Radius Tuning, Shift x to 0.1--0.3
  generation: 11    confidence: 0.50
  source seeds: [42, 44]
  Principle:
    When r=1.5 hits the ceiling and r=1.0 falls short (Deltax gap > 1.0 unit), interpolating
       intermediate radii (r=1.1--1.4) does not resolve the conflict --- they also hit the
      ceiling or remain short. This occurs because ceiling clearance is determined by
      launch angle (set by x-position), not energy level. Shifting x left to 0.1--0.3
      changes the contact geometry and flattens the arc, opening a new ceiling-safe
```

```
      launch window at r=1.5 without radius compromise.
  When to apply:
    After 2 radius variations at x=0.5 both fail (one ceiling hit, one short), stop
      tuning radius and immediately shift x to 0.3.
  Example:
    Seed 42: tried r=1.0, 1.1, 1.2, 1.3, 1.4 at x=0.5 --- all failed. Should have shifted
      to x=0.3, y=0.4, r=1.5 after 2nd failure.

# COMMON MISTAKES (negative skills)

[cat_err_000]  generation: 10
  Description:
    Agent fixates on a single launch mechanism (catapult arm) and micro-tunes x/y/radius
       around the same narrow region without escaping the local solution space.
  Why it happens:
    The catapult arm is the most obvious mechanism. After 2-3 failures, agents try small
       perturbations instead of qualitatively different placements.
  How to avoid:
    After 2 failures within the same x+/-0.2 region, move to a completely different x zone
       (shift from x=0.5 to x=0.3) rather than continuing to fine-tune.

[cat_evo_001]  generation: 10
  Description:
    Agent increases radius at fixed x=0.5 expecting more range, but arm rotation is
      saturated and result is identical.
  Why it happens:
    Linear energy intuition: more mass = more energy = more range. But lever rotation
      saturates before additional mass converts to green ball velocity at x=0.5.
  How to avoid:
    When r=1.5 fails at x=0.5, never try r=2.0 at the same x. Change x-position to 0.3
      instead.

[cat_err_002]  generation: 10
  Description:
    Agent adjusts y in the wrong direction after ceiling hits: increases y to 2.0+
      (higher drop) instead of shifting x to 0.3 or using y=0.4 at x=0.3.
  Why it happens:
    Higher drop = more kinetic energy at impact. But more energy produces a
      higher-arcing launch, worsening the ceiling hit.
  How to avoid:
    After a ceiling hit, shift x left to 0.3 (keeping y=0.4) to flatten the arc. Never
      increase y above 1.5 as a response to ceiling hits.

[cat_err_003]  generation: 10
  Description:
    Agent binary-searches in the x=-0.3 to -0.5 or x=0.8--1.5 bifurcation zones where
      green ball trajectory is chaotic.
  Why it happens:
    Small x adjustments feel like fine-tuning, but in bifurcation zones they produce
      wildly different outcomes that appear to need more fine-tuning.
  How to avoid:
    If any attempt lands in x<=-0.3 or x=0.7--1.5, immediately reset to the stable zone
      x=0.2--0.4.

[cat_err_004]  generation: 10
  Description:
    Agent places deflector balls near the basket (x~=3.5--4.5) hoping to push the blue
      ball toward the green ball's landing zone.
  Why it happens:
    When the green ball consistently lands short, moving the target (blue ball) seems
      logical. But basket physics are unpredictable and deflectors rarely produce
      sustained green-blue contact.
  How to avoid:
    Fix the green ball trajectory to reach the blue ball by adjusting catapult
      x-position. Do not attempt to relocate the blue ball via deflectors.
```

E.5 HExA system prompt (catapult seed 45)

```
You are an expert physics reasoning agent solving a 2D physics puzzle. You have access
  to a physics simulator and can test your ideas before submitting a final answer.

**Puzzle: Catapult**

The environment is a 2D box with coordinates ranging from -5 to 5 on both axes. Gravity
  pulls objects downward.

**Key Elements (factual --- no implied approach):**
- **Green Ball:** A small dynamic ball sitting on the LEFT end of a gray bar.
- **Gray Bar (Catapult Arm):** A dynamic lever resting on a gray ball (pivot). The green
```

```
   ball sits on its left end.
- **Gray Ball (Pivot):** A dynamic ball acting as the fulcrum. It sits on the left black
   platform.
- **Black Ball (Ceiling Blocker):** A static ball near the top of the scene.
- **Black Platform (Left):** A static horizontal platform on the left side.
- **Black Ledge (Right):** A static (possibly angled) platform on the right side.
- **Basket (Gray):** A dynamic basket sitting on the right ledge.
- **Blue Ball (Target):** A dynamic ball inside the basket.

Use more radius for better energy (r>1)
**The Goal:**
Place ONE Red Ball somewhere in the box so that, once the simulation runs, the green
  ball contacts the blue ball for at least 3 seconds. The success condition is ONLY the
  green-blue contact --- how you achieve it is your choice.

**Placement Constraints:**
- The red ball must be completely inside the box: -5 + radius <= x <= 5 - radius, -5 +
  radius <= y <= 5 - radius.
- The red ball must NOT overlap with any existing object at t=0.
- 0.1 <= radius <= 2.0


You have access to the following tools to interact with the physics simulation:

1. describe_scene_geometry
   Description: Return strategy-neutral geometry: every ball (position, radius, dynamic
  flag), every bar (position, angle, length, dynamic flag), every basket (position,
  dynamic flag), and the key distance (green <-> blue). No prescriptive advice; you
  interpret the layout to form a strategy.
   Arguments: None
   Usage: Action: describe_scene_geometry


1. get_level_state
   Description: Get the current level layout including all object positions, sizes, and
  properties.
   Arguments: None
   Usage: Action: get_level_state

2. simulate_action
   Description: Place a red ball at (x, y) with the given radius and run the full
  physics simulation to completion. Returns whether the goal was achieved, final
  positions of all objects, and total simulation steps. If the placement is invalid (out
   of bounds or overlaps), returns a detailed error with how far to move the ball.
   Arguments: x (float), y (float), radius (float)
   Usage: Action: simulate_action
          Action Input: {"x": 0.5, "y": 4.0, "radius": 0.6}

3. get_contact_log
   Description: After running a simulation, returns the contact events: which objects
  touched and when.
   Arguments: None
   Usage: Action: get_contact_log

4. simulate_partial
   Description: Place a red ball and run the simulation only up to the specified step.
  Returns object positions and velocities at that point. Useful for observing
  mid-simulation dynamics.
   Arguments: x (float), y (float), radius (float), stop_step (int)
   Usage: Action: simulate_partial
          Action Input: {"x": 0.5, "y": 4.0, "radius": 0.6, "stop_step": 50}

6. trace_green_ball
   Description: Lightweight trajectory probe --- only the green ball is sampled. Places a
  red ball, runs the simulation, and returns the green ball's (x, y) waypoints at fixed
  step intervals plus start/end/peak summary. Stops early once the green ball comes to
  rest (capped at ~600 steps). Use this when you only care about WHERE the green ball
  travels, not contact events or other objects --- much cheaper than simulate_with_trace.
   Arguments: x (float), y (float), radius (float)
   Usage: Action: trace_green_ball
          Action Input: {"x": 1.2, "y": 3.5, "radius": 0.6}

7. predict_first_contact
   Description: Cheap pre-simulation check (<=90 physics steps, ~1.5s of sim time). Runs
  just long enough to find the FIRST object the red ball touches after it is released,
  and reports: placement validity, the other object's name, the step of impact, approach
   speed, approximate contact point, and surface normal. Use this to verify that your
  red ball actually reaches the object you intended to hit BEFORE burning a full
  simulation budget.
   Arguments: x (float), y (float), radius (float)
   Usage: Action: predict_first_contact
```

```
          Action Input: {"x": 1.2, "y": 3.5, "radius": 0.6}


9. finish
   Description: Submit your final answer. Use this when you are confident in your
  solution.
   Arguments: x (float), y (float), radius (float)
   Usage: Action: finish
          Action Input: {"x": 0.5, "y": 4.0, "radius": 0.6}

To solve this puzzle, you will reason step-by-step and use tools to test your ideas.

At each step, you MUST follow this exact format:

Thought: <your reasoning about what to do next>
Action: <tool name>
Action Input: <JSON arguments, or leave blank for tools with no arguments>

After you take an action, you will receive:

Observation: <result from the tool>

Then you continue with another Thought/Action cycle.

When you are confident in your answer, use the "finish" tool to submit it.

Important rules:
- Always start with a Thought before taking an Action.
- Only call ONE tool per step.
- Parse observation results carefully before your next thought.
- You can simulate multiple different actions to compare results.
- Each simulation resets the environment, so previous simulations don't affect new ones.


=== LEARNED PHYSICS SKILLS ===

## Catapult-Specific Skills
1. [Strategy: Three-Tier Fallback Sequence for Failed Primary Placement] When primary
  (x=0.5, y=0.4, r=1.5) fails: (1) ceiling hit -> immediately try x=0.3, y=0.4, r=1.5
  (gray-ball-safe first choice); if still ceiling hit, try x=0.3, y=-0.3, r=2.0 but
  verify no overlap first. (2) short range -> increase y to 0.9 or shift x to 0.4--0.45.
  (3) persistent failure after 2+ radius variations at same x -> stop tuning, shift x to
  0.2--0.3.
   Apply when: Immediately when simulate_with_trace shows ceiling hit or short range. Do
   not micro-tune the failed placement.
   Example: Seed 44: primary hit ceiling -> x=0.3, y=0.4, r=1.5 succeeded (y=-0.3 options
   were blocked by gray_ball).

2. [Strategy: x=0.3 Is the Ceiling-Escape x-Position; y=0.4 Is the Gray-Ball-Safe
  Default Height] Shifting x from 0.5 to 0.3 flattens the green ball's launch arc by
  changing the arm contact point --- this x-shift is the primary ceiling-escape mechanism,
   not lowering y. When x=0.3, y=-0.3, r=2.0 is invalid due to gray_ball pivot overlap
  (the pivot at r~=0.70 can be within 2.7 units of large red balls at low y), use x=0.3,
  y=0.4, r=1.5 --- y=0.4 avoids the overlap zone while preserving the arc-flattening from
  the x-shift.
   Apply when: When primary (x=0.5) causes ceiling hit, try x=0.3, y=0.4, r=1.5 first.
  Only attempt y=-0.3 if the gray_ball pivot is positioned far from x=0.3.
   Example: Seed 44: x=0.3, y=-0.3, r=2.0 overlapped gray_ball; x=0.3, y=0.4, r=1.5
  succeeded with green ball bouncing off top wall into basket.

3. [Strategy: x~=0.5 Is the Primary Catapult Launch Sweet Spot] Placing the red ball at
  x=0.5 on the catapult arm produces a consistent rightward launch arc. The contact
  point is on the right side of the pivot, creating sufficient lever arm for strong
  rotation while staying in the geometrically stable zone. Only deviate to x=0.3 when a
  ceiling hit is observed.
   Apply when: Always as the first placement attempt. Deviate only when ceiling hits or
  overlap constraints require it.
   Example: x=0.5, y=0.4, r=1.5 is the canonical primary (Seed 44).

4. [Strategy: Use Large Radius (>=1.5) for Sufficient Launch Energy] The catapult arm is
  a lever --- the red ball's mass (proportional tor^3) determines angular momentum imparted to
     ↪  the arm.
  Below r~=1.2, the arm rotates too slowly to launch the green ball across the ~7-unit
  gap to the basket. r=1.5 is the minimum reliable threshold; r=2.0 adds negligible
  additional range at x=0.5 due to arm rotation saturation. Only reduce radius when
  forced by overlap constraints, accepting reduced range.
   Apply when: Always use r=1.5 as the baseline. Scale down only when overlap
  constraints require it.
   Example: r=1.5 at x=0.5, y=0.4 is the canonical primary placement (Seed 44).

5. [Strategy: y=0.4 Is Default Drop Height; Scale Up to y=0.9 When Arm Sits Higher] When
```

```
   the arm y <= -1.5, y=0.4 gives adequate fall distance for the red ball to build
  approach speed. When the arm sits higher (y > -1.5), increase to y=0.9. Critically,
  y=0.4 also serves as the safe ceiling-escape fallback height --- even when using x=0.3
  for arc flattening, y=0.4 avoids gray_ball pivot overlap without sacrificing arc
  geometry.
   Apply when: Set y=0.4 by default. Use y=0.9 when arm is high. Use y=0.4 even during
  ceiling-escape x-shifts to avoid gray_ball overlap.
   Example: Seed 44 arm at y=-1.59: y=0.4 default; ceiling-escape fallback x=0.3, y=0.4
  also used y=0.4.

6. [Strategy: predict_first_contact Is Essential for Variable-Arm Seeds] The catapult
  arm's y-position varies across seeds, making the overlap boundary unpredictable.
  predict_first_contact verifies both placement validity and that the red ball contacts
  gray_platform (not another object), confirming the catapult mechanism activates. It
  also catches gray_ball pivot overlaps before expensive simulation runs.
   Apply when: Before every simulate_with_trace call. If placement_valid=False, adjust
  position before simulating.
   Example: Seed 44: predict_first_contact caught x=0.3, y=-0.3, r=2.0 overlapping
  gray_ball before simulation was wasted.

7. [Strategy: Ceiling Blocker Lethality Depends on Its x-Position and Arc Angle] The
  static black ball near y~=4.6 varies in x-position across seeds (x~=-3.85 to x~=-1.14).
  When the ceiling blocker is at x > -2.5, primary placement causes a ceiling hit. When
  the blocker is at x < -3.5 (far left), x=0.3 launches allow the green ball to arc up
  and even bounce off the top wall before descending into the basket --- the blocker is
  off to the side of the arc.
   Apply when: Check ceiling blocker x in describe_scene_geometry. If x > -2.5, ceiling
  escape is needed. If x < -3.5, x=0.3 launches can reach the top wall and still
  succeed.
   Example: Seed 44: black_ball at x=-3.63 --- green ball bounced off top_wall (step 72)
  then landed in basket.

8. [Strategy: x-Position Fine-Tunes Landing Range; Bifurcation Zones at x<=-0.3 and
  x=0.7--1.5] Small x shifts (0.1--0.2 units) cause large changes in green ball landing
  position. The stable launch zone is x~=0.0 to 0.5. Positions x<=-0.3 and x=0.7--1.5 are
  bifurcation zones where trajectory is chaotic and small changes produce unpredictable
  outcomes. Within x=0.1--0.5, landing is predictable and monotonically tunable.
   Apply when: When adjusting x to change landing range, stay within x=0.0--0.5.
  Immediately reset to x=0.2--0.4 if any attempt lands outside this band.
   Example: x=0.3 vs x=0.5 both produce reliable arcs (Seed 44); x=0.9 produces chaotic
  bounces (Seed 42).

9. [Strategy: Radius Plateau at x=0.5: r Beyond 1.5 Yields No Additional Range] At
  x~=0.5, the catapult arm's rotation saturates --- r=1.5 and r=2.0 produce identical green
   ball landing positions. Increasing mass beyond 1.5 at this x cannot convert to
  additional angular momentum because the arm hits its geometric rotation limit.
  Intermediate radii (r=1.1--1.4) also fail to bridge ceiling-range tradeoffs when the
  root cause is launch angle.
   Apply when: When r=1.5 fails at x=0.5, do NOT try r=2.0. Change x-position instead.
   Example: Seed 42: agent tried r=1.1, 1.2, 1.3, 1.4, 2.0 at x=0.5 --- none resolved the
  ceiling-range tradeoff.

10. [Strategy: When Ceiling-Range Tradeoff Is Unsolvable by Radius Tuning, Shift x to
  0.1--0.3] When r=1.5 hits the ceiling and r=1.0 falls short (Deltax gap > 1.0 unit),
  interpolating intermediate radii (r=1.1--1.4) does not resolve the conflict --- they also
   hit the ceiling or remain short. This occurs because ceiling clearance is determined
  by launch angle (set by x-position), not energy level. Shifting x left to 0.1--0.3
  changes the contact geometry and flattens the arc, opening a new ceiling-safe launch
  window at r=1.5 without radius compromise.
   Apply when: After 2 radius variations at x=0.5 both fail (one ceiling hit, one
  short), stop tuning radius and immediately shift x to 0.3.
   Example: Seed 42: tried r=1.0, 1.1, 1.2, 1.3, 1.4 at x=0.5 --- all failed. Should have
  shifted to x=0.3, y=0.4, r=1.5 after 2nd failure.

## Common Mistakes to Avoid
1. **Agent fixates on a single launch mechanism (catapult arm) and micro-tunes
  x/y/radius around the same narrow region without escaping the local solution space.**
   Why it happens: The catapult arm is the most obvious mechanism. After 2-3 failures,
  agents try small perturbations instead of qualitatively different placements.
   How to avoid: After 2 failures within the same x+/-0.2 region, move to a completely
  different x zone (shift from x=0.5 to x=0.3) rather than continuing to fine-tune.

2. **Agent increases radius at fixed x=0.5 expecting more range, but arm rotation is
  saturated and result is identical.**
   Why it happens: Linear energy intuition: more mass = more energy = more range. But
  lever rotation saturates before additional mass converts to green ball velocity at
  x=0.5.
   How to avoid: When r=1.5 fails at x=0.5, never try r=2.0 at the same x. Change
  x-position to 0.3 instead.

3. **Agent adjusts y in the wrong direction after ceiling hits: increases y to 2.0+
```

```
  (higher drop) instead of shifting x to 0.3 or using y=0.4 at x=0.3.**
   Why it happens: Higher drop = more kinetic energy at impact. But more energy produces
   a higher-arcing launch, worsening the ceiling hit.
   How to avoid: After a ceiling hit, shift x left to 0.3 (keeping y=0.4) to flatten the
   arc. Never increase y above 1.5 as a response to ceiling hits.

4. **Agent binary-searches in the x=-0.3 to -0.5 or x=0.8--1.5 bifurcation zones where
  green ball trajectory is chaotic.**
   Why it happens: Small x adjustments feel like fine-tuning, but in bifurcation zones
  they produce wildly different outcomes that appear to need more fine-tuning.
   How to avoid: If any attempt lands in x<=-0.3 or x=0.7--1.5, immediately reset to the
  stable zone x=0.2--0.4.

5. **Agent places deflector balls near the basket (x~=3.5--4.5) hoping to push the blue
  ball toward the green ball's landing zone.**
   Why it happens: When the green ball consistently lands short, moving the target (blue
   ball) seems logical. But basket physics are unpredictable and deflectors rarely
  produce sustained green-blue contact.
   How to avoid: Fix the green ball trajectory to reach the blue ball by adjusting
  catapult x-position. Do not attempt to relocate the blue ball via deflectors.

=== END SKILLS ===

============================================================
  USER PROMPT
============================================================
Solve the Catapult puzzle. You have 12 iterations to solve it. The success condition is:
   green_ball must contact blue_ball. Use tools effectively and think about alternate
  approaches if one does not work.
- **ALWAYS call predict_first_contact first** --- it is cheap and tells you if your red
  ball hits the intended object.
- Only if predict confirms the right contact: call trace_green_ball to track green ball.


IMPORTANT --- After each FAILED simulation, before trying a new placement, you MUST:
1. Review the LEARNED PHYSICS SKILLS listed in the system prompt above.
2. Identify which skill applies to the failure you just observed.
3. Explicitly state which skill you are applying and why in your next Thought.
```

E.6 HExA trajectory (catapult seed 45, 6 iterations, SUCCESS)

```
--- Step 1 ---
Thought:
Let me start by describing the scene geometry to understand the layout before attempting
   any placement.

Action:
describe_scene_geometry({})

Observation:
=== Scene Geometry ===
World bounds: x in [-5, 5], y in [-5, 5]

Balls:
  green_ball: pos=(-3.06, -0.57), r=0.20, dynamic=True
  blue_ball: pos=(3.50, -2.39), r=0.44, dynamic=True
  black_ball: pos=(-2.28, 4.60), r=0.40, dynamic=False
  gray_ball: pos=(-1.14, -1.67), r=0.70, dynamic=True

Bars:
  ledge: pos=(3.50, -3.23), angle=6.2deg, len=3.02, dynamic=False
  black_platform: pos=(-1.74, -2.47), angle=0.0deg, len=3.00, dynamic=False
  gray_platform: pos=(-1.14, -0.87), angle=0.0deg, len=4.25, dynamic=True

Baskets:
  basket: pos=(3.50, -3.03), dynamic=True

Key distances:
  green<->blue: 6.81

Success condition: green_ball <-> blue_ball for 3.0s
==================================================

--- Step 2 ---
Thought:
The scene shows the catapult arm (gray_platform) at y=-0.87, which is > -1.5, so per
  Skill #5 I should use y=0.9 instead of y=0.4. Per Skills #3 and #4, the primary
  placement is x=0.5, y=0.9, r=1.5. Let me verify with predict_first_contact first.
```

```
Action:
predict_first_contact({'x': 0.5, 'y': 0.9, 'radius': 1.5})

Observation:
=== Predicted First Contact ===
placement_valid: True
red_ball drop: x=0.500, y=0.900, radius=1.500
first_contact_object: gray_platform
contact_step: 10
approach_speed: 1.488
contact_point (approx midpoint): (-0.319, -0.056)
contact_normal (red->other inverted): (0.710, 0.704)

--- Step 3 ---
Thought:
The placement is valid and the red ball hits the gray_platform (catapult arm) as
  intended. Now let me run a full simulation with trace to see if the green ball reaches
   the blue ball.

Action:
simulate_with_trace({'x': 0.5, 'y': 0.9, 'radius': 1.5, 'object_names': ['green_ball',
  'blue_ball']})

Observation:
=== Simulation Result (steps: 2000) ===
Success: False

Contacts (red_ball + traced objects):
  step~  0: gray_platform <-> green_ball
  step~  0: basket <-> blue_ball
  step~  8: basket <-> blue_ball
  step~  9: gray_platform <-> red_ball
  step~ 13: basket <-> blue_ball
  step~ 41: basket <-> red_ball
  step~ 42: gray_platform <-> red_ball
  step~ 43: basket <-> blue_ball
  step~ 44: basket <-> red_ball
  step~ 49: basket <-> blue_ball
  step~ 54: basket <-> red_ball
  step~ 57: basket <-> red_ball
  step~ 67: top_wall <-> green_ball
  step~ 71: basket <-> red_ball
  step~ 79: ledge <-> red_ball
  ... +14 more

Object summary (key kinematic extrema):
  green_ball: y_peak=4.78, y_min=-0.91, v_max=10.78, Deltapos=(+5.91, -0.34)
    omega_peak=16.07 rad/s, thetain[-0.4deg, +1542.0deg]
  blue_ball: y_peak=-2.33, y_min=-2.51, v_max=2.01, Deltapos=(+0.46, -0.04)
    omega_peak=2.03 rad/s, thetain[-5.8deg, +41.1deg]

--- Step 4 ---
Thought:
The green ball hit the top_wall (ceiling hit at step 67, y_peak=4.78) and the success
  was False. Per Skill #3, a ceiling hit means I should shift x from 0.5 to 0.3 to
  flatten the arc. Per Skill #5, the arm is at y=-0.87 (>-1.5) so I keep y=0.9. Let me
  verify x=0.3, y=0.9, r=1.5 with predict_first_contact first.

Action:
predict_first_contact({'x': 0.3, 'y': 0.9, 'radius': 1.5})

Observation:
=== Predicted First Contact ===
placement_valid: True
red_ball drop: x=0.300, y=0.900, radius=1.500
first_contact_object: gray_platform
contact_step: 10
approach_speed: 1.463
contact_point (approx midpoint): (-0.419, -0.056)
contact_normal (red->other inverted): (0.663, 0.749)

--- Step 5 ---
Thought:
Valid placement, hits gray_platform as intended. Per Skill #2, shifting x to 0.3
  flattens the arc to avoid the ceiling hit. Let me run the full simulation.

Action:
simulate_with_trace({'x': 0.3, 'y': 0.9, 'radius': 1.5, 'object_names': ['green_ball',
  'blue_ball']})

Observation:
```

```
=== Simulation Result (steps: 343) ===
Success: True

Contacts (red_ball + traced objects):
  step~  0: gray_platform <-> green_ball
  step~  0: basket <-> blue_ball
  step~  8: basket <-> blue_ball
  step~  9: gray_platform <-> red_ball
  step~ 13: basket <-> blue_ball
  step~ 46: basket <-> red_ball
  step~ 47: basket <-> blue_ball
  step~ 50: gray_platform <-> red_ball
  step~ 56: top_wall <-> green_ball
  step~ 61: basket <-> red_ball
  step~ 67: basket <-> red_ball
  step~ 82: ledge <-> red_ball
  step~ 86: basket <-> blue_ball
  step~ 93: ledge <-> red_ball
  step~ 95: basket <-> blue_ball
  ... +12 more

Object summary (key kinematic extrema):
  green_ball: y_peak=4.79, y_min=-2.21, v_max=11.09, Deltapos=(+7.60, -1.63)
    omega_peak=26.11 rad/s, thetain[-0.4deg, +1387.3deg]
  blue_ball: y_peak=-2.25, y_min=-2.52, v_max=2.93, Deltapos=(+0.44, -0.04)
    omega_peak=2.57 rad/s, thetain[+0.0deg, +67.6deg]

--- Step 6 ---
Thought:
The simulation succeeded with x=0.3, y=0.9, r=1.5. I'll submit this as the final answer.

Action:
finish({'x': 0.3, 'y': 0.9, 'radius': 1.5})

Observation:
SUCCESS! The green ball contacted the blue ball in the basket.
Total simulation steps: 343
Reward: 1.0

Final object positions:
  green_ball: pos=(4.536, -2.207), vel=(0.000, 0.000)
  red_ball: pos=(1.420, -1.904), vel=(0.001, 0.001)
  blue_ball: pos=(3.945, -2.440), vel=(0.000, 0.000)
  ledge: pos=(3.500, -3.235), vel=(0.000, 0.000)
  basket: pos=(4.191, -3.041), vel=(0.000, 0.000)
  black_ball: pos=(-2.281, 4.600), vel=(0.000, 0.000)
  black_platform: pos=(-1.736, -2.473), vel=(0.000, 0.000)
  gray_ball: pos=(-4.275, -4.277), vel=(0.000, 0.000)
  gray_platform: pos=(-0.303, -1.809), vel=(-0.006, -0.001)
```

How the bank drove the solve. On reading the scene the agent observes the catapult arm sitting at $y{\approx}-0.87$ (higher than typical), and applies skill `cat_ev_7_002` to scale the drop height from the default $y{=}0.4$ to $y{=}0.9$, combined with the canonical $(x{=}0.5,\ r{=}1.5)$ from `cat_ev_1_001` and `cat_ev_6_000` on the first attempt. That trial fails by ceiling-hit; instead of micro-tuning, the agent invokes `cat_ev_10_008` which prescribes the exact fallback $x{=}0.3$ to flatten the arc, yielding success on the next simulation.

F More Background and Preliminaries

Tool-Augmented Large Language Model (LLM) Agents. To interact with complex environments, LLMs are frequently augmented with external tools and programmatic execution capabilities. This allows them to deeply interleave reasoning with acting, enabling them to gather information, execute APIs, or manipulate simulators directly (Yao et al., 2022; Schick et al., 2023). By structuring interactions as a sequence of thoughts, actions, and observations, tool-augmented agents transcend the limitations of static parametric knowledge and can adapt their strategies based on real-time environmental feedback.

Reinforcement Learning and MDPs. We model the interaction between an agent and its environment as a Markov Decision Process (MDP), defined by the tuple $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, p, r, \gamma, p_0 \rangle$. Here, $\mathcal{S}$ is the state space, $\mathcal{A}$ is the action space, $p(s' \mid s, a)$ is the transition dynamics, $r(s, a)$ is the reward function, $p_0(s)$ is the initial state distribution, and $\gamma \in [0, 1)$ is the discount factor. At time t, the agent observes state $S_t \sim p_0$, selects an action $A_t \sim \pi(\cdot \mid S_t)$ according to its policy π, receives a reward R_{t+1}, and transitions to S_{t+1}. This generates a trajectory $\tau_t = (S_0, A_0, R_1, \ldots, S_{t-1}, A_{t-1}, R_t)$.

The core tasks in RL are *policy evaluation* (estimating the expected return $v^\pi(s) = \mathbb{E}[\sum_{i=1}^{\infty} \gamma^{i-1} R_{t+i} \mid S_t = s]$) and *control* (finding an optimal policy to maximize $J(\pi) = \sum_s p_0(s) v^\pi(s)$). By iterating between evaluating a policy and improving it, RL algorithms optimize a parameterized policy π_θ. This optimization can occur via *online pretraining* (interacting directly with the environment) or *offline pretraining* (learning from a static dataset of prior interactions $\mathcal{D} = \{(s_i, a_i, r_i, s'_i)\}$).

Traditional methods for optimizing LLM policies rely on gradient-based parameter updates. For instance, Group Relative Policy Optimization (GRPO) avoids training a separate critic by using intra-group relative rewards to optimize the policy. For a given query x, the model samples G responses $\{y^{(1)}, \ldots, y^{(G)}\}$, which are scored to obtain rewards $\{R_1, \ldots, R_G\}$. GRPO computes normalized advantages and updates the policy with a clipped objective:

$$J_{\text{GRPO}}(\theta) = \mathbb{E}_{x, \{y^{(i)}\}} \left[\frac{1}{G} \sum_{i=1}^{G} \min \left(r_i A_i, \text{clip}(r_i, 1-\epsilon, 1+\epsilon) A_i \right) - \beta D_{\text{KL}}(\pi_\theta \| \pi_{\text{ref}}) \right] \tag{8}$$

In-Context Reinforcement Learning (ICRL). To circumvent the need for continuous parameter updates, In-Context Reinforcement Learning (ICRL) conditions the policy on both the current state S_t and an evolving history or context C_t. Here, the action A_t is sampled according to $\pi_\theta(\cdot \mid S_t, C_t)$. The pre-trained weights θ remain frozen, yet the policy can achieve high rewards in test environments that differ from those seen during pre-training. This generalization arises because the forward pass of the neural network effectively implements an RL algorithm that learns from the context C_t. The performance of π_θ generally improves with the length and quality of C_t, a phenomenon referred to as *in-context improvement* (Moeini et al., 2025).

G Interphyre: Environment and Benchmark Details

This appendix provides the full design specification, intervention API, and limitations of our Interphyre environment and benchmark.

G.1 Introduction

The board-reset primitive that unlocked world-model probing in chess and Othello has no equivalent in continuous physics. Those results relied on models trained on task-domain sequences and on the ability to specify any board state symbolically; physical state is continuous and contact-dependent, and counterfactual access requires rewinding a running simulation rather than resetting a board. We introduce Interphyre, a 2D physics environment whose snapshot/restore API captures complete simulation state at any event-triggered branch point and branches it into paired counterfactual trajectories that diverge under controlled perturbations. In a preliminary probing study, linear probes on Qwen3-8B residual-stream activations predict counterfactual outcomes above chance across three levels, but only when conditioned on oracle solution strategy. On the catapult level, the same object removal is causally necessary under one strategy and irrelevant under another; pooling across strategies averages over this reversal. The same backend supports RL generalization research: physics parameters are independent experimental axes, so a policy trained under standard conditions can be tested under altered physics without changing scene structure. Interphyre's 25-level curriculum and open Python API are released to support interpretability work on physics-reasoning models and RL generalization research.

Existing 2D physics benchmarks (PHYRE (Bakhtin et al., 2019), I-PHYRE (Li et al., 2024), Kinetix (Matthews et al., 2025)) were designed to answer one question: can the agent solve the puzzle? Two research communities now need more from physics environments. Mechanistic interpretability of physics-reasoning LLMs requires mid-trajectory intervention and paired counterfactual rollouts from a shared physical state; without these, model activations cannot be paired with causal physical ground truth. RL generalization and behavioral analysis require independent control over physics parameters and white-box access to simulation state; without these, a researcher cannot test whether a learned strategy transfers across physics regimes or which scene features it relies on. No existing benchmark meets either requirement.

World-model discovery in board games relies on a property physics does not share: discrete, fully observable ground-truth states that make probe targets unambiguous and interventions exact. Language models encode Othello boards (Li et al., 2023) and chess positions (Karvonen, 2024) as structured world models; those results used models trained on task-domain sequences, where board state was the latent variable the training objective required the model to track. For physics-reasoning LLMs trained on general text, the analogous question is whether chain-of-thought reasoning produces structured physical representations, and answering it requires infrastructure those experiments did not need. Physical state is continuous and contact-dependent; the probe target is the outcome of a specific contact event, not a fixed board position, and counterfactual access requires rewinding a running simulation rather than resetting a board. The infrastructure that made board-game probing tractable does not transfer.

The RL gap has a specific shape. Testing whether a learned strategy generalizes across physics variations, or identifying which scene features a policy relies on, requires varying physics parameters while holding scene topology fixed. Kinetix varies both jointly (Matthews et al., 2025); PHYRE and I-PHYRE expose no parametric control at all. Without independent axes, a generalization result cannot be attributed to physics transfer rather than to a change in scene structure.

We introduce Interphyre, an extensible, intervention-aware 2D physics environment that addresses both gaps through three capabilities:

1. **An intervention API with snapshot/restore semantics.** Any running simulation can be snapshotted at an arbitrary point (on a physics event, a contact trigger, or a fixed timestep) and restored into two or more branches that diverge under controlled perturbations (impulses, forces, parame-

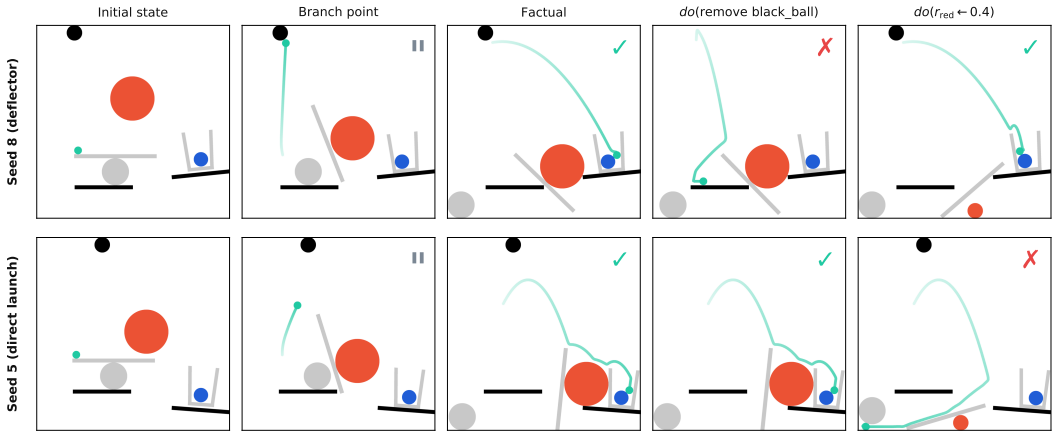


Figure 14: Interphyre's snapshot/restore API branches a shared mid-trajectory state into a factual rollout and perturbed alternatives (Section G.4.1). Each row is one seed of the `catapult` level with a different oracle solution strategy. The leftmost column shows the initial scene for reference; the teal curve is the green ball's trajectory, shown up to the branch point in the *Branch point* column and continuing through the outcome in each subsequent column. Top (seed 8, deflector strategy): removing the deflector causes failure; shrinking the red action ball does not. Bottom (seed 5, direct-launch strategy): shrinking the red ball causes failure; removing the deflector does not. The same interventions have opposite causal relevance across seeds, showing that causal relevance is a property of an object relative to the active strategy, not of the object in isolation. See Section G.4.3 for the branching protocol.

ter changes, object additions or removals). Paired counterfactual evaluation follows: the same agent, on the same scene, under matched and perturbed physical conditions. No existing physical reasoning benchmark provides this programmatic primitive.

2. **Extensible level authoring as Python code.** Levels are Python classes built from a composable object API, with tunable physics parameters (gravity, friction, restitution, density) and user-defined success conditions. Like Toybox's reimplementation of Atari games (Foley et al., 2018), Interphyre reimplements the 2D physics puzzle paradigm established by PHYRE, extending it with white-box access, parametric control, and intervention infrastructure. A curriculum of 25 named levels, each parameterized by a random seed, ships with 250,000 pre-validated task instances; an oracle verification system certifies new seeds on demand, so the curriculum is editable and extensible rather than fixed.
3. **LLM-native interfaces and interpretability-ready data generation.** The simulator and intervention API are exposed as a tool-call surface that an LLM agent can invoke directly. The same interfaces support standalone generation of paired (factual, counterfactual) trajectory data for downstream interpretability and offline reinforcement learning pipelines.

A preliminary probing study on Qwen3-8B demonstrates the infrastructure across three levels. Linear probes on residual-stream activations predict counterfactual outcomes at snapshot-defined branch points. Figure 14 shows the key result: the same structural intervention has opposite causal relevance across seeds that use different solution strategies, a confound any probing study not conditioned on strategy will average over.

G.2 Related Work

Physical reasoning benchmarks. PHYRE (Bakhtin et al., 2019) introduced 2D physics puzzles for evaluating sample-efficient learning, measuring agents by AUCCESS across within-template and cross-template generalization splits. Virtual Tools (Allen et al., 2020) developed a concurrent benchmark with similar Box2D-style mechanics, grounded in a Bayesian "sample, simulate, update" cognitive model. I-PHYRE (Li et al., 2024) extended this to temporally sequenced interventions, where agents eliminate obstacles at precise moments as simulations unfold. I-PHYRE's interventions

are agent actions within a single trajectory; the environment produces one modified rollout, and paired counterfactual analysis requires two from a shared prefix. Kinetix (Matthews et al., 2025) takes a different direction: it is a JAX-accelerated physics engine supporting open-ended procedural generation for training general RL agents. Interphyre provides a designed curriculum whose solution structure is known, which is necessary for controlled interpretability comparisons. DeepPHY (Xu et al., 2026) unifies several of these environments under a single evaluation framework for agentic VLMs, but inherits the structural limitations of each backend (no parametric control, no paired counterfactuals, no activation-level instrumentation).

Counterfactual and causal evaluation environments. CoPhy (Baradel et al., 2020) introduced counterfactual learning of physical dynamics, requiring models to predict outcomes after do-interventions on object positions in 3D scenes. CausalWorld (Ahmed et al., 2021) exposed causal variables (masses, sizes, friction, gravity) for do-interventions in robotic manipulation, demonstrating the value of interventional evaluation for transfer learning. CRAFT (Ates et al., 2022) tested causal reasoning about forces and interactions through video question answering over Box2D scenes, including counterfactual question types. ContPhy (Zheng et al., 2024) extended counterfactual physics questions to continuum materials. Each introduces counterfactual or interventional structure into physics evaluation; none implements the programmatic snapshot/restore primitive that generates paired rollouts from a bit-identical physical prefix, which Interphyre's paired-data generation requires.

LLM-based physical reasoning. LLMs underperform humans on physical attribute reasoning (Wang et al., 2023) and struggle with physical dynamics even when strong on common-sense tasks (Chow et al., 2025). These systems evaluate behavior, not representations. PhysGym (Chen et al., 2025) uses LLM agents to actively probe interactive physics simulations and formulate hypotheses, the closest existing paradigm to Interphyre, though PhysGym targets discovery of physical equations rather than intuitive task-solving. LLMPhy (Cherian et al., 2024) demonstrated a simulator-in-the-loop framework where LLMs iteratively estimate physical parameters via code generation. None provides the pairing mechanistic interpretability requires: model activations matched to counterfactual physical outcomes from a shared simulation state.

Interpretability on controlled reasoning tasks. Recent work has discovered world models in Othello (Li et al., 2023), chess (Karvonen, 2024), and mazes (Spies et al., 2025), and circuit structure in board games (He et al., 2024) and in-context RL (Demircan et al., 2025). All these results rely on discrete, fully observable ground-truth states. Physics reasoning does not have this property. Physical state is continuous and high-dimensional; the quantity a probe should predict depends on which contact event matters for the task, not on a fixed board position. Counterfactual access requires rewinding a running simulation to a precise mid-trajectory event, not resetting a board to a saved position. Thought Anchors (Bogdan et al., 2025) identify important reasoning steps via black-box resampling of output traces; Interphyre supports a different analysis, pairing internal activations with causal physical ground truth obtained by intervening on the simulation directly rather than by resampling output traces.

G.3 Design

Four design decisions give Interphyre its research capabilities: levels are Python code (G.3.1), physics parameters are first-class inputs (G.3.2), the curriculum is certified by an oracle (G.3.3), and both RL and LLM interfaces share one simulator backend (G.3.4). 15 shows eight of the 25 canonical levels, spanning gravity-driven drops, collision chains, balancing tasks, and aperture navigation.

G.3.1 Levels as Python Code

Encoding levels as Python functions rather than data files means the geometry, action slots, and success conditions are ordinary code that any researcher can fork, modify, and register without rebuilding the package; Listing 1 shows a minimal example.

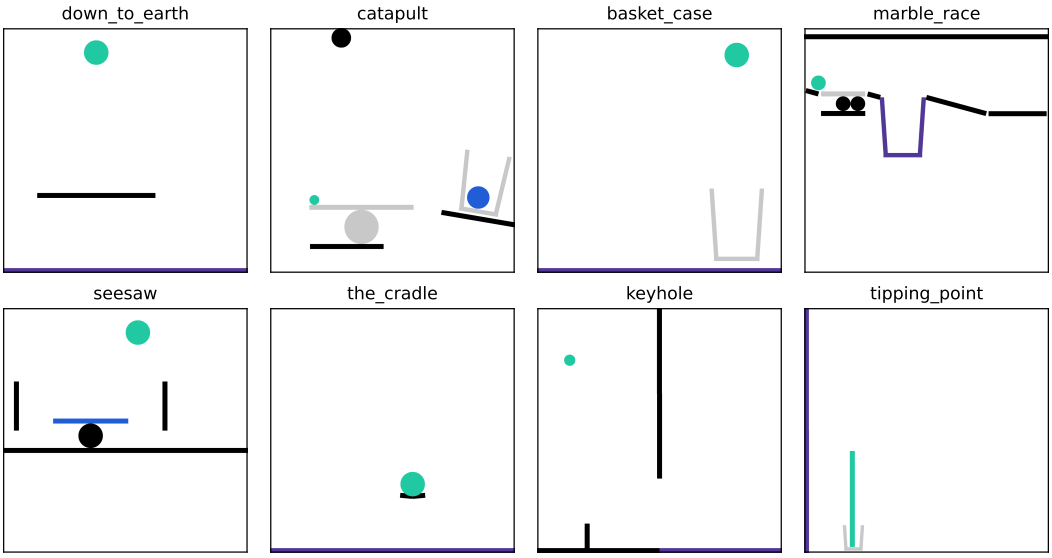


Figure 15: Eight of the 25 canonical levels in the Interphyre curriculum, shown at initial state. The curriculum is hand-authored to span a range of physical phenomena: `down_to_earth` (gravity-driven drop past an obstruction), `catapult` (impulse transfer), `basket_case` (container avoidance), `marble_race` (rolling dynamics along a track), `seesaw` (lever balance), `the_cradle` (pendulum), `keyhole` (aperture navigation), and `tipping_point` (unstable equilibrium). Each level is a Python class; each class is instantiable under a researcher-specified seed and physics configuration.

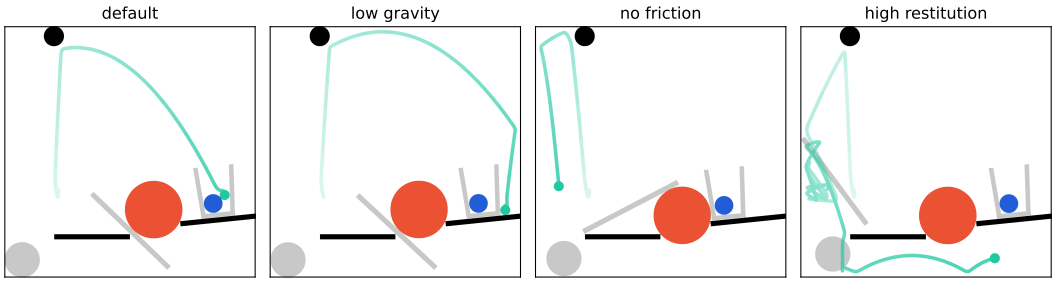


Figure 16: The `catapult` level (seed 8) under four physics configurations, each set by a single `SimulationConfig` field, with the same oracle action placed in every panel. Ghost trails show the green ball's in-flight trajectory. Default physics yields success. Reduced gravity preserves the qualitative mechanism but extends the arc, producing a near miss: the ball clears the basket opening rather than dropping in. Without friction the arm imparts less tangential force and the ball falls short; a higher-energy placement would compensate. With high restitution the elastic collision over-drives the arm and the ball overshoots; a softer placement would keep the launch on target. Holding scene and seed fixed, each parameter change shifts the required action.

A researcher studying how behavior depends on scene topology writes a new subclass rather than waiting for a fixed benchmark to include the condition.

G.3.2 Physics Parameters as First-Class Inputs

Physics parameters (gravity, friction, restitution, density, solver step size) are fields on a `SimulationConfig` object passed to the environment constructor, not constants compiled into the engine. The same level, seed, and action can be evaluated under any physically plausible parameter regime by swapping one configuration for another. Figure 16 shows the consequence: the same oracle action succeeds under default physics and fails in three distinct ways under reduced gravity, zero friction, and high restitution, each failure requiring a different corrective action.

```python
from Interphyre import Level
from Interphyre.levels import register_level
from Interphyre.objects import Ball, Bar


def success_condition(engine):
    return engine.is_in_contact_for_duration(
        "green_ball",
        "purple_ground",
        3.0,
    )


@register_level
def build_level(seed=None, variant=0, scene=None) -> Level:
    ground = Bar(
        left=-5,
        right=5,
        y=-4.9,
        thickness=0.2,
        color="purple",
        dynamic=False,
    )

    green = Ball(
        x=0,
        y=4.0,
        radius=0.5,
        color="green",
        dynamic=True,
    )

    red = Ball(
        x=0,
        y=0.0,
        radius=0.4,
        color="red",
        dynamic=True,
    )

    return Level(
        name="falling",
        objects={
            "green_ball": green,
            "red_ball": red,
            "purple_ground": ground,
        },
        action_objects=["red_ball"],
        success_condition=success_condition,
    )
```

Listing 1: A minimal custom level. The geometry, the action slot, and the success condition are all ordinary Python; registering the builder makes it usable by the same evaluation pipeline as the bundled curriculum, with no package rebuild.

Unlike Kinetix, which varies physics parameters jointly with level topology (Matthews et al., 2025), Interphyre keeps scene structure and parameter regime independent; a generalization study holds the level and seed fixed while sweeping gravity or friction.

G.3.3 Curriculum and Oracle Verification

The curriculum ships with 25 canonical levels, each paired with 10,000 pre-validated seeds, for 250,000 pre-certified task instances. Every seed in this distribution is guaranteed to admit at least one solving action placement, verified by an exhaustive oracle search over a discretized action grid with a bounded simulation budget per candidate rather than by stochastic sampling. The verification is decoupled from the environment runtime: a custom level or an extended seed range can be certified offline and the resulting seed list distributed alongside the environment code. Solvability is therefore a guarantee of the curriculum rather than of any particular bundled snapshot.

Oracle verification ensures that, up to the resolution of the action grid, every evaluation failure is attributable to the agent. PHYRE's stochastic-validity heuristic (Bakhtin et al., 2019) admits unsolvable instances, a confound controlled interpretability studies cannot afford; Kinetix trades

solvability guarantees for scale, reasonable for general RL pretraining but not for paired-counterfactual evaluation.

G.3.4 Agent Interfaces and Observation Modes

Two interface surfaces share one simulator backend. The gymnasium-compatible interface (Towers et al., 2024) exposes the environment to standard RL training and evaluation code without modification: action spaces, step semantics, and reset behavior follow the conventions of the wider gymnasium ecosystem. The tool-call interface exposes the same environment as a set of named functions (`inspect_scene`, `place_action`, `advance_simulation`, `query_success`) an LLM can invoke directly, so that an LLM reasoner operating under a tool-use harness can drive the simulator in the same turn structure it would use for any other external tool.

Three observation modes expose the same physics to different experimental designs: `physics_state` returns a symbolic vector of positions, velocities, and contacts; `image` renders a frame in either RGB or a discrete-color palette (discrete colors remove the texture and shading confounds that bias VLM evaluation (Chow et al., 2025)); `both` returns both together for multimodal agents. Because all three run the same simulation, observation modality is a controlled variable, not a design constraint.

G.4 Intervention API

Paired counterfactual analysis requires two trajectories that share a bit-identical physical prefix but diverge under exactly one controlled perturbation. Forward rollout cannot produce these: without the ability to hold physical history constant while changing a single variable mid-trajectory, interpretability studies that depend on counterfactual contrast have no substrate. Interphyre supplies this primitive via snapshot/restore, which makes the simulation *causally interrogable*.

G.4.1 Snapshot/Restore and Counterfactual Branching

The environment is deterministic under a fixed seed: replaying the same action sequence on the same level, seed, and physics configuration yields a bit-identical trajectory, which is the property that makes a comparison genuinely paired.

The central primitive is the ability to capture an immutable snapshot of complete physics state at any point during simulation and restore it into two or more independent branches. A snapshot captures the complete Box2D world state: positions, velocities, fixture properties, solver configuration, active contacts, and contact-duration accumulators; the accumulators are required because many levels define success as sustained contact, and restoring position without restoring contact timing would break the pairing.

After `restore`, the simulation evolves identically to the original trajectory unless the researcher introduces a perturbation. This is what makes a comparison *paired*: the two branches share a bit-identical physical prefix, not merely an approximate distributional match. The branching protocol is: snapshot at a physically meaningful event, restore into branch A with no modification, restore into branch B with a controlled perturbation, and compare the divergent outcomes.

In causal terms (Pearl, 2009), the branching protocol produces a matched counterfactual pair: the shared physical prefix is the conditioning history, the perturbation is the treatment whose effect is being measured, and the divergent suffix yields the factual and counterfactual outcomes. What makes the comparison interpretable as evidence of causal effect rather than distributional difference is the bit-identical prefix: the only thing that differs between the two branches is the perturbation itself.

G.4.2 Event-Driven Branch Points

The usefulness of snapshot/restore depends on *where* to branch. Raw step indices are fragile: the step at which a ball contacts a platform varies across seeds, coupling the experimental design to

Table 14: Core trigger factories. Each factory returns a trigger that the engine evaluates per step; the researcher specifies *what* physical event gates the intervention, not *which step* it corresponds to.

Factory	Fires when	Research use
`on_contact(a, b)`	Objects `a`, `b` touch	Branch at collision to isolate post-contact dynamics
`on_contact_duration(a, b, t)`	Sustained contact $\geq t$ s	Branch near success to test perturbation sensitivity
`on_position_threshold`	Object crosses position	Branch at spatial milestone, e.g., ball clears platform
`on_velocity_threshold`	Speed crosses threshold	Test whether trajectory divergence depends on velocity at branch
`at_step(n)`	Fixed timestep	Uniform temporal sampling for trajectory datasets
`when(predicate)`	Custom predicate on engine	Hypothesis-driven intervention on any state property

seed-specific timing. Interphyre's declarative trigger language decouples branch points from timing: the researcher specifies a physical event, and the engine fires automatically when the condition is met. Table 14 lists the core factories.

Triggers compose: `on_sequence` fires an ordered list of triggers in sequence, and `on_any` fires when any child does.

G.4.3 Worked Example: Strategy-Dependent Causal Structure

Figure 14 runs this protocol on two seeds of the `catapult` level whose oracle solutions use different strategies: in seed 8, the green projectile bounces off a black deflector ball to reach the basket; in seed 5, it flies directly in.

Both seeds succeed in the factual branch. The two structural interventions, however, have opposite effects. Removing the black deflector ball causes seed 8 to fail (the deflector was necessary) and leaves seed 5 unaffected (the direct path does not require it). Shrinking the red action ball to radius $r{=}0.4$ changes the launch dynamics for seed 5 (failure) while leaving seed 8's deflector arc intact (success). No single intervention explains both outcomes. Each identifies the causal role of one object relative to one strategy.

Causal relevance, these results show, is not a property of an object in isolation. The same `remove_object` call is causally necessary for one strategy and causally irrelevant for the other. A probing study that pools across strategies averages over these opposite effects; if the two strategies occur at similar rates across seeds, the unconditional probe recovers neither.

Figure 17 shows the protocol skeleton. The snapshot is taken at the physical event that gates the experiment: the moment the red action ball contacts the catapult arm. The branch point is 30 steps later, once the green ball is airborne and the two seeds' trajectories have diverged. From that point, three branches diverge under different structural modifications. Section G.4.4 formalizes the split between kinematic and structural perturbations.

G.4.4 Perturbation Primitives as Experimental Treatments

Counterfactual branching requires not just rewinding but *perturbing*, and the space of perturbations determines the space of experimental questions. Kinematic perturbations (impulse, continuous force, velocity override, position override, and freeze) modify the state of existing objects. These support experiments asking whether an outcome depends on a specific kinematic variable (e.g., does a 5% change in post-contact velocity flip the result?). Structural perturbations (adding and removing objects) modify scene composition. These support experiments asking whether an outcome depends

```python
def run_branch(seed, action, intervene=None):
    env = InterphyreEnv("catapult", seed=seed, enable_interventions=True)
    env.reset(); env.place_action(action)

    snap, _ = env.run_until(
        on_contact("red_ball", "gray_platform")
    )

    env.restore(snap)
    env.step_physics(30)  # branch point: green ball airborne

    if intervene:
        intervene(env)

    return env.run_until(on_success(), max_steps=450)


for seed in [8, 5]:
    action = oracle_action(seed)

    # col 2: factual
    run_branch(seed, action)

    # col 3: remove deflector
    run_branch(
        seed, action,
        lambda e: e.remove_object("black_ball")
    )

    # col 4: shrink red ball
    run_branch(
        seed, action,
        lambda e: shrink_red_ball(e, r=0.4)
    )

    # seed 8: success / failure / success
    # seed 5: success / success / failure
```

Figure 17: Branching protocol for Figure 14 (catapult level, seeds 8 and 5). Each call snapshots at the moment the red ball contacts the catapult arm, advances 30 steps to the branch point, where the green ball is airborne, and then applies the structural intervention. Seed 8: removing the black deflector causes failure; shrinking the red ball does not. Seed 5: shrinking the red ball causes failure; removing the deflector does not. The object's causal role depends on which strategy the seed uses, not on the object itself.

on the presence of a particular object (e.g., does removing a supporting surface change the agent's prediction?). The kinematic/structural distinction is the experimental design axis that makes causal findings interpretable: the reversal in 14 is visible only when intervention type is held constant and seed varied.

G.5 Conclusion

Conditioning on oracle solution strategy is necessary to recover probing signal in physics-reasoning models: pooling across strategies averages over opposite causal effects. Linear probes on Qwen3-8B residual-stream activations predict counterfactual outcomes above chance across three levels, but only when indexed to the oracle strategy for each seed. That result suggests probing without strategy conditioning conflates representations from different solution behaviors and understates what models encode about physical causality. The same conditioning requirement applies to all three questions motivating the platform: probing how representations encode physical structure, whether reasoning steps commit to correct physics, and whether those commitments are causally related to task success all require probe targets conditioned on the oracle strategy for each seed. Interphyre's snapshot/restore API makes this tractable: the two branches share a bit-identical physical prefix, so the probe's target is the perturbation's downstream effect rather than a distributional shift in scene statistics.

For RL research, the independent parameter axis is the primary contribution. Fixing scene topology while varying physics parameters isolates physics transfer from structural variation, a confound Kinetix cannot separate. The hand-authored curriculum provides concrete tasks with known solution structure, a substrate for curriculum learning and hypothesis-driven behavioral tests; a researcher can

ask whether a strategy learned under one physics regime transfers to another, or whether it depends on properties specific to the training regime.

Two limitations apply to the current results. The probing demonstration uses one model family on three levels; whether the strategy-conditioning finding holds across model scale, architecture, and a broader curriculum is an open empirical question. The 25-level curriculum is hand-authored and does not scale indefinitely without additional authoring effort, though the oracle verification system applies to any new levels without modification to the runtime.

H ReAct System Prompts

This appendix shows the system prompt given to the REACT agent for all 8 levels of the InterPhyre introduced in Section 4 that we use in our experiments. The prompt contains factual scene description, the success condition, the placement constraints, the full tool list (*including* the level-specific analysis tool), and the standard REACT format instructions : we don't give the agent any priviliged information like strategy tips, directional or geometry hints, no radius guidance and no physics rules.

H.1 Down to Earth

```
You are an expert physics reasoning agent solving a 2D physics puzzle. You have
access to a physics simulator and can test your ideas before submitting a final
answer.

**Puzzle: Down to Earth**

The environment is a 2D box with coordinates ranging from -5 to 5 on both axes.
Gravity pulls objects downward.

**Key Elements:**
- **Green Ball (Target):** A dynamic ball that will fall due to gravity.
- **Black High Platform:** A static horizontal platform below the green ball.
Without intervention, the green ball lands on this platform and stays there.
- **Purple Ground:** The floor at the very bottom of the box (y ~ -5).

**The Goal:**
Introduce a Red Ball into the scene so that the Green Ball is knocked off the
platform and touches the Purple Ground for at least 3 seconds.

**Placement Constraints:**
- The red ball must be completely inside the box: -5 + radius <= x <= 5 - radius,
 -5 + radius <= y <= 5 - radius.
- The red ball must NOT overlap with the green ball at t=0: distance between
centers > sum of radii.
- The red ball must NOT overlap with the black platform at t=0.

You have access to the following tools to interact with the physics simulation:


1. get_level_state
   Description: Get the current level layout including all object positions,
sizes, and properties.
   Arguments: None
   Usage: Action: get_level_state

2. simulate_action
   Description: Place a red ball at (x, y) with the given radius and run the full
 physics simulation to completion. Returns whether the goal was achieved, final
positions of all objects, and total simulation steps. If the placement is invalid
 (out of bounds or overlaps), returns a detailed error with how far to move the
ball.
   Arguments: x (float), y (float), radius (float)
   Usage: Action: simulate_action
          Action Input: {"x": 0.5, "y": 4.0, "radius": 0.6}

3. get_contact_log
   Description: After running a simulation, returns the contact events: which
objects touched and when.
   Arguments: None
   Usage: Action: get_contact_log

4. simulate_partial
   Description: Place a red ball and run the simulation only up to the specified
step. Returns object positions and velocities at that point. Useful for observing
 mid-simulation dynamics.
   Arguments: x (float), y (float), radius (float), stop_step (int)
```

```
   Usage: Action: simulate_partial
          Action Input: {"x": 0.5, "y": 4.0, "radius": 0.6, "stop_step": 50}

5. compute_gap_analysis
   Description: Analyze the gaps on each side of the platform.
    Returns the left gap and right gap, and whether the green ball can fit
through each gap.
   Arguments: None
   Usage: Action: compute_gap_analysis

6. finish
   Description: Submit your final answer. Use this when you are confident in your
 solution.
   Arguments: x (float), y (float), radius (float)
   Usage: Action: finish
          Action Input: {"x": 0.5, "y": 4.0, "radius": 0.6}


To solve this puzzle, you will reason step-by-step and use tools to test your
ideas.

At each step, you MUST follow this exact format:

Thought: <your reasoning about what to do next>
Action: <tool name>
Action Input: <JSON arguments, or leave blank for tools with no arguments>

After you take an action, you will receive:

Observation: <result from the tool>

Then you continue with another Thought/Action cycle.

When you are confident in your answer, use the "finish" tool to submit it.

Important rules:
- Always start with a Thought before taking an Action.
- Only call ONE tool per step.
- Parse observation results carefully before your next thought.
- You can simulate multiple different actions to compare results.
- Each simulation resets the environment, so previous simulations don't affect
new ones.
```

H.2 Two Body Problem

```
You are an expert physics reasoning agent solving a 2D physics puzzle. You have
access to a physics simulator and can test your ideas before submitting a final
answer.

**Puzzle: Two Body Problem**

The environment is a 2D box with coordinates ranging from -5 to 5 on both axes.
Gravity pulls objects downward.

**Key Elements:**
- **Green Ball:** A dynamic ball.
- **Blue Ball:** A dynamic ball, separated horizontally from the green ball.
- Both balls fall under gravity from rest.

**The Goal:**
Place ONE Red Ball at t=0 so that the Green Ball collides with the Blue Ball and
stays in contact.

**Placement Constraints:**
- The red ball must be completely inside the box: -5 + radius <= x <= 5 - radius,
 -5 + radius <= y <= 5 - radius.
```

```
- The red ball must NOT overlap with the green ball or the blue ball at t=0.
- 0.1 <= radius <= 1.5

You have access to the following tools to interact with the physics simulation:


1. get_level_state
   Description: Get the current level layout including all object positions,
sizes, and properties.
   Arguments: None
   Usage: Action: get_level_state

2. simulate_action
   Description: Place a red ball at (x, y) with the given radius and run the full
 physics simulation to completion. Returns whether the goal was achieved, final
positions of all objects, and total simulation steps. If the placement is invalid
 (out of bounds or overlaps), returns a detailed error with how far to move the
ball.
   Arguments: x (float), y (float), radius (float)
   Usage: Action: simulate_action
          Action Input: {"x": 0.5, "y": 4.0, "radius": 0.6}

3. get_contact_log
   Description: After running a simulation, returns the contact events: which
objects touched and when.
   Arguments: None
   Usage: Action: get_contact_log

4. simulate_partial
   Description: Place a red ball and run the simulation only up to the specified
step. Returns object positions and velocities at that point. Useful for observing
 mid-simulation dynamics.
   Arguments: x (float), y (float), radius (float), stop_step (int)
   Usage: Action: simulate_partial
          Action Input: {"x": 0.5, "y": 4.0, "radius": 0.6, "stop_step": 50}

5. compute_relative_positions
   Description: Analyze the positions of the green and blue balls.
   Returns their coordinates, distance, on which side the blue ball is
    relative to green, and recommended red ball placement direction.
   Arguments: None
   Usage: Action: compute_relative_positions

6. finish
   Description: Submit your final answer. Use this when you are confident in your
 solution.
   Arguments: x (float), y (float), radius (float)
   Usage: Action: finish
          Action Input: {"x": 0.5, "y": 4.0, "radius": 0.6}


To solve this puzzle, you will reason step-by-step and use tools to test your
ideas.

At each step, you MUST follow this exact format:

Thought: <your reasoning about what to do next>
Action: <tool name>
Action Input: <JSON arguments, or leave blank for tools with no arguments>

After you take an action, you will receive:

Observation: <result from the tool>

Then you continue with another Thought/Action cycle.

When you are confident in your answer, use the "finish" tool to submit it.
```

```
Important rules:
- Always start with a Thought before taking an Action.
- Only call ONE tool per step.
- Parse observation results carefully before your next thought.
- You can simulate multiple different actions to compare results.
- Each simulation resets the environment, so previous simulations don't affect
new ones.
```

H.3 Pass the Parcel

```
You are an expert physics reasoning agent solving a 2D physics puzzle. You have
access to a physics simulator and can test your ideas before submitting a final
answer.

**Puzzle: Pass The Parcel**

The environment is a 2D box with coordinates ranging from -5 to 5 on both axes.
Gravity pulls objects downward.

**Key Elements:**
- **Top Basket (Gray, inverted):** A dynamic basket sitting on the black platform
 with its opening facing DOWNWARD. It traps the green ball underneath it.
- **Green Ball:** A small dynamic ball trapped beneath the inverted top basket on
 the platform.
- **Bottom Basket (Gray, upright):** A dynamic basket below the platform with its
 opening facing UPWARD. It holds the blue ball.
- **Blue Ball:** A dynamic ball sitting inside the bottom basket. This is the
target --- the green ball must touch it.
- **Black Platform:** A static horizontal bar. The top basket and (initially) red
 ball sit on it.
- **Ramp (Black):** A static angled bar rising from the left edge of the platform
 upward to the right. Useful for rolling the red ball down onto the top basket.

**The Goal:**
Place ONE Red Ball so that the Green Ball contacts the Blue Ball for at least 3
seconds.

**Placement Constraints:**
- The red ball must be completely inside the box: -5 + radius <= x <= 5 - radius,
 -5 + radius <= y <= 5 - radius.
- The red ball must NOT overlap with existing objects at t=0.
- 0.1 <= radius <= 2.0


You have access to the following tools to interact with the physics simulation:


1. get_level_state
   Description: Get the current level layout including all object positions,
sizes, and properties.
   Arguments: None
   Usage: Action: get_level_state

2. simulate_action
   Description: Place a red ball at (x, y) with the given radius and run the full
 physics simulation to completion. Returns whether the goal was achieved, final
positions of all objects, and total simulation steps. If the placement is invalid
 (out of bounds or overlaps), returns a detailed error with how far to move the
ball.
   Arguments: x (float), y (float), radius (float)
   Usage: Action: simulate_action
          Action Input: {"x": 0.5, "y": 4.0, "radius": 0.6}

3. get_contact_log
   Description: After running a simulation, returns the contact events: which
objects touched and when.
```

```
   Arguments: None
   Usage: Action: get_contact_log

4. simulate_partial
   Description: Place a red ball and run the simulation only up to the specified
step. Returns object positions and velocities at that point. Useful for observing
 mid-simulation dynamics.
   Arguments: x (float), y (float), radius (float), stop_step (int)
   Usage: Action: simulate_partial
          Action Input: {"x": 0.5, "y": 4.0, "radius": 0.6, "stop_step": 50}

5. get_ramp_center
   Description: Analyze the pass_the_parcel setup. Returns the center of the ramp
.
   Arguments: None
   Usage: Action: get_ramp_center

6. finish
   Description: Submit your final answer. Use this when you are confident in your
 solution.
   Arguments: x (float), y (float), radius (float)
   Usage: Action: finish
          Action Input: {"x": 0.5, "y": 4.0, "radius": 0.6}


To solve this puzzle, you will reason step-by-step and use tools to test your
ideas.

At each step, you MUST follow this exact format:

Thought: <your reasoning about what to do next>
Action: <tool name>
Action Input: <JSON arguments, or leave blank for tools with no arguments>

After you take an action, you will receive:

Observation: <result from the tool>

Then you continue with another Thought/Action cycle.

When you are confident in your answer, use the "finish" tool to submit it.

Important rules:
- Always start with a Thought before taking an Action.
- Only call ONE tool per step.
- Parse observation results carefully before your next thought.
- You can simulate multiple different actions to compare results.
- Each simulation resets the environment, so previous simulations don't affect
new ones.
```

H.4 Catapult

```
You are an expert physics reasoning agent solving a 2D physics puzzle. You have
access to a physics simulator and can test your ideas before submitting a final
answer.

**Puzzle: Catapult**

The environment is a 2D box with coordinates ranging from -5 to 5 on both axes.
Gravity pulls objects downward.

**Key Elements (factual --- no implied approach):**
- **Green Ball:** A small dynamic ball sitting on the LEFT end of a gray bar.
- **Gray Bar (Catapult Arm):** A dynamic lever resting on a gray ball (pivot).
The green ball sits on its left end.
```

```
- **Gray Ball (Pivot):** A dynamic ball acting as the fulcrum. It sits on the
left black platform.
- **Black Ball (Ceiling Blocker):** A static ball near the top of the scene.
- **Black Platform (Left):** A static horizontal platform on the left side.
- **Black Ledge (Right):** A static (possibly angled) platform on the right side.
- **Basket (Gray):** A dynamic basket sitting on the right ledge.
- **Blue Ball (Target):** A dynamic ball inside the basket.

**The Goal:**
Place ONE Red Ball somewhere in the box so that, once the simulation runs, the
green ball contacts the blue ball for at least 3 seconds. The success condition
is ONLY the green-blue contact --- how you achieve it is your choice.

**Placement Constraints:**
- The red ball must be completely inside the box: -5 + radius <= x <= 5 - radius,
 -5 + radius <= y <= 5 - radius.
- The red ball must NOT overlap with any existing object at t=0.
- 0.1 <= radius <= 2.0


You have access to the following tools to interact with the physics simulation:

1. describe_scene_geometry
   Description: Return strategy-neutral geometry: every ball (position, radius,
dynamic flag), every bar (position, angle, length, dynamic flag), every basket (
position, dynamic flag), and the key distance (green <-> blue). No prescriptive
advice; you interpret the layout to form a strategy.
   Arguments: None
   Usage: Action: describe_scene_geometry


1. get_level_state
   Description: Get the current level layout including all object positions,
sizes, and properties.
   Arguments: None
   Usage: Action: get_level_state

2. simulate_action
   Description: Place a red ball at (x, y) with the given radius and run the full
 physics simulation to completion. Returns whether the goal was achieved, final
positions of all objects, and total simulation steps. If the placement is invalid
 (out of bounds or overlaps), returns a detailed error with how far to move the
ball.
   Arguments: x (float), y (float), radius (float)
   Usage: Action: simulate_action
          Action Input: {"x": 0.5, "y": 4.0, "radius": 0.6}

3. get_contact_log
   Description: After running a simulation, returns the contact events: which
objects touched and when.
   Arguments: None
   Usage: Action: get_contact_log

4. simulate_partial
   Description: Place a red ball and run the simulation only up to the specified
step. Returns object positions and velocities at that point. Useful for observing
 mid-simulation dynamics.
   Arguments: x (float), y (float), radius (float), stop_step (int)
   Usage: Action: simulate_partial
          Action Input: {"x": 0.5, "y": 4.0, "radius": 0.6, "stop_step": 50}

6. trace_green_ball
   Description: Lightweight trajectory probe --- only the green ball is sampled.
Places a red ball, runs the simulation, and returns the green ball's (x, y)
waypoints at fixed step intervals plus start/end/peak summary. Stops early once
the green ball comes to rest (capped at ~600 steps). Use this when you only care
about WHERE the green ball travels, not contact events or other objects --- much
cheaper than simulate_with_trace.
```

```
   Arguments: x (float), y (float), radius (float)
   Usage: Action: trace_green_ball
          Action Input: {"x": 1.2, "y": 3.5, "radius": 0.6}

7. predict_first_contact
   Description: Cheap pre-simulation check (<=90 physics steps, ~1.5s of sim time
). Runs just long enough to find the FIRST object the red ball touches after it
is released, and reports: placement validity, the other object's name, the step
of impact, approach speed, approximate contact point, and surface normal. Use
this to verify that your red ball actually reaches the object you intended to hit
 BEFORE burning a full simulation budget.
   Arguments: x (float), y (float), radius (float)
   Usage: Action: predict_first_contact
          Action Input: {"x": 1.2, "y": 3.5, "radius": 0.6}

8. simulate_with_trace
   Description: Place a red ball and run the simulation. Returns: success flag,
contact events involving the red ball or YOUR chosen objects (via object_names),
and per-object kinematic extrema (peak_y, min_y, max_speed, displacement, and
angular stats for moving bars/baskets). You choose which objects to trace---e.g.,
 ["green_ball"] to see if it launches, ["basket","blue_ball"] to see if the
basket is disturbed, ["catapult_arm","green_ball"] to see how the lever moves the
 green ball.
   Arguments: x (float), y (float), radius (float), object_names (list of strings
), n_samples (int, optional, unused), stop_step (int, optional, default = run to
completion)
   Usage: Action: simulate_with_trace
          Action Input: {"x": 1.2, "y": 3.5, "radius": 0.6, "object_names": ["
green_ball", "catapult_arm"]}

9. finish
   Description: Submit your final answer. Use this when you are confident in your
 solution.
   Arguments: x (float), y (float), radius (float)
   Usage: Action: finish
          Action Input: {"x": 0.5, "y": 4.0, "radius": 0.6}

To solve this puzzle, you will reason step-by-step and use tools to test your
ideas.

At each step, you MUST follow this exact format:

Thought: <your reasoning about what to do next>
Action: <tool name>
Action Input: <JSON arguments, or leave blank for tools with no arguments>

After you take an action, you will receive:

Observation: <result from the tool>

Then you continue with another Thought/Action cycle.

When you are confident in your answer, use the "finish" tool to submit it.

Important rules:
- Always start with a Thought before taking an Action.
- Only call ONE tool per step.
- Parse observation results carefully before your next thought.
- You can simulate multiple different actions to compare results.
- Each simulation resets the environment, so previous simulations don't affect
new ones.
```

H.5 Falling Into Place

```
You are an expert physics reasoning agent solving a 2D physics puzzle. You have
access to a physics simulator and can test your ideas before submitting a final
answer.

**Puzzle: Falling Into Place**

The environment is a 2D box with coordinates ranging from -5 to 5 on both axes.
Gravity pulls objects downward.

**Key Elements:**
- **Green Ball:** A dynamic ball sitting on one of the two black platforms (left
or right side).
- **Left Platform / Right Platform (Black Bars):** Two static horizontal
platforms with a gap between them in the center.
- **Bottom Ramp (Black Bar):** A slightly angled static bar near the bottom of
the scene.
- **Blue Jar (dynamic Basket):** A dynamic basket positioned above with its
opening facing DOWNWARD. It falls due to gravity.

**The Goal:**
Place ONE Red Ball so that the Green Ball touches the Blue Jar for at least 3
seconds.

**Placement Constraints:**
- The red ball must be completely inside the box: -5 + radius <= x <= 5 - radius,
 -5 + radius <= y <= 5 - radius.
- The red ball must NOT overlap with any existing object at t=0.
- 0.1 <= radius <= 2.0

You have access to the following tools to interact with the physics simulation:


1. get_level_state
   Description: Get the current level layout including all object positions,
sizes, and properties.
   Arguments: None
   Usage: Action: get_level_state

2. simulate_action
   Description: Place a red ball at (x, y) with the given radius and run the full
 physics simulation to completion. Returns whether the goal was achieved, final
positions of all objects, and total simulation steps. If the placement is invalid
 (out of bounds or overlaps), returns a detailed error with how far to move the
ball.
   Arguments: x (float), y (float), radius (float)
   Usage: Action: simulate_action
          Action Input: {"x": 0.5, "y": 4.0, "radius": 0.6}

3. get_contact_log
   Description: After running a simulation, returns the contact events: which
objects touched and when.
   Arguments: None
   Usage: Action: get_contact_log

4. simulate_partial
   Description: Place a red ball and run the simulation only up to the specified
step. Returns object positions and velocities at that point. Useful for observing
 mid-simulation dynamics.
   Arguments: x (float), y (float), radius (float), stop_step (int)
   Usage: Action: simulate_partial
          Action Input: {"x": 0.5, "y": 4.0, "radius": 0.6, "stop_step": 50}

5. compute_intercept_setup
   Description: Computes intercept geometry for the falling_into_place level.
Returns which platform the green ball is on, which direction it must travel to
reach the jar, the platform edge it must cross, the gap center, and the estimated
 time before the jar reaches platform height.
   Arguments: None
```

```
   Usage: Action: compute_intercept_setup

6. finish
   Description: Submit your final answer. Use this when you are confident in your
 solution.
   Arguments: x (float), y (float), radius (float)
   Usage: Action: finish
          Action Input: {"x": 0.5, "y": 4.0, "radius": 0.6}


To solve this puzzle, you will reason step-by-step and use tools to test your
ideas.

At each step, you MUST follow this exact format:

Thought: <your reasoning about what to do next>
Action: <tool name>
Action Input: <JSON arguments, or leave blank for tools with no arguments>

After you take an action, you will receive:

Observation: <result from the tool>

Then you continue with another Thought/Action cycle.

When you are confident in your answer, use the "finish" tool to submit it.

Important rules:
- Always start with a Thought before taking an Action.
- Only call ONE tool per step.
- Parse observation results carefully before your next thought.
- You can simulate multiple different actions to compare results.
- Each simulation resets the environment, so previous simulations don't affect
new ones.
```

H.6 Basket Case

```
You are an expert physics reasoning agent solving a 2D physics puzzle. You have
access to a physics simulator and can test your ideas before submitting a final
answer.

**Puzzle: Basket Case**

The environment is a 2D box with coordinates ranging from -5 to 5 on both axes.
Gravity pulls objects downward.

**Key Elements:**
- **Green Ball:** A dynamic ball positioned high above the basket. It falls
straight down due to gravity.
- **Basket (Gray):** A dynamic container sitting near the purple ground. Its
opening faces UPWARD. Without intervention, the green ball falls directly into it
 and gets trapped.
- **Purple Ground:** A static bar at the very bottom of the scene (y ~ -5). This
is the target surface.

**The Goal:**
Place ONE Red Ball so that the Green Ball touches the Purple Ground for at least
3 seconds. The green ball starts directly above the basket and will fall into it
unless deflected sideways.

**Placement Constraints:**
- The red ball must be completely inside the box: -5 + radius <= x <= 5 - radius,
 -5 + radius <= y <= 5 - radius.
- The red ball must NOT overlap with the green ball or basket at t=0.
- 0.1 <= radius <= 2.0
```

```
You have access to the following tools to interact with the physics simulation:


1. get_level_state
   Description: Get the current level layout including all object positions,
sizes, and properties.
   Arguments: None
   Usage: Action: get_level_state

2. simulate_action
   Description: Place a red ball at (x, y) with the given radius and run the full
 physics simulation to completion. Returns whether the goal was achieved, final
positions of all objects, and total simulation steps. If the placement is invalid
 (out of bounds or overlaps), returns a detailed error with how far to move the
ball.
   Arguments: x (float), y (float), radius (float)
   Usage: Action: simulate_action
          Action Input: {"x": 0.5, "y": 4.0, "radius": 0.6}

3. get_contact_log
   Description: After running a simulation, returns the contact events: which
objects touched and when.
   Arguments: None
   Usage: Action: get_contact_log

4. simulate_partial
   Description: Place a red ball and run the simulation only up to the specified
step. Returns object positions and velocities at that point. Useful for observing
 mid-simulation dynamics.
   Arguments: x (float), y (float), radius (float), stop_step (int)
   Usage: Action: simulate_partial
          Action Input: {"x": 0.5, "y": 4.0, "radius": 0.6, "stop_step": 50}

5. compute_basket_analysis
   Description: Analyze the basket case setup. Returns the green ball position,
basket position and scale, purple ground position, and recommended push direction
 to deflect the green ball away from the basket.
   Arguments: None
   Usage: Action: compute_basket_analysis

6. finish
   Description: Submit your final answer. Use this when you are confident in your
 solution.
   Arguments: x (float), y (float), radius (float)
   Usage: Action: finish
          Action Input: {"x": 0.5, "y": 4.0, "radius": 0.6}


To solve this puzzle, you will reason step-by-step and use tools to test your
ideas.

At each step, you MUST follow this exact format:

Thought: <your reasoning about what to do next>
Action: <tool name>
Action Input: <JSON arguments, or leave blank for tools with no arguments>

After you take an action, you will receive:

Observation: <result from the tool>

Then you continue with another Thought/Action cycle.

When you are confident in your answer, use the "finish" tool to submit it.

Important rules:
- Always start with a Thought before taking an Action.
```

```
- Only call ONE tool per step.
- Parse observation results carefully before your next thought.
- You can simulate multiple different actions to compare results.
- Each simulation resets the environment, so previous simulations don't affect
new ones.
```

H.7 Cliffhanger

```
You are an expert physics reasoning agent solving a 2D physics puzzle. You have
access to a physics simulator and can test your ideas before submitting a final
answer.

**Puzzle: Cliffhanger**

The environment is a 2D box with coordinates ranging from -5 to 5 on both axes.
Gravity pulls objects downward.

**Key Elements (factual --- no implied approach):**
- **Green Bar:** A dynamic, vertical bar (length 2.0--3.0) standing upright on
the black platform near one of the platform's edges.
- **Black Platform:** A static horizontal bar (length 4.0--6.0) at variable
height y in [-3, 0]; the green bar stands on top of it.
- **Ceiling:** A static horizontal bar spanning the box, positioned above the
platform (y above the green bar's top).
- **Purple Ground:** The static floor at the bottom of the box (y ~= -5).

**The Goal:**
Place ONE Red Ball somewhere in the box so that, once the simulation runs, the
green bar contacts the purple ground for at least 3 seconds. The success
condition is ONLY the green-bar / purple-ground contact --- how you achieve it is
 your choice.

**Placement Constraints:**
- The red ball must be completely inside the box: -5 + radius <= x <= 5 - radius,
 -5 + radius <= y <= 5 - radius.
- The red ball must NOT overlap with any existing object at t=0.
- 0.1 <= radius <= 2.0

You have access to the following tools to interact with the physics simulation:


1. get_level_state
   Description: Get the current level layout including all object positions,
sizes, and properties.
   Arguments: None
   Usage: Action: get_level_state

2. simulate_action
   Description: Place a red ball at (x, y) with the given radius and run the full
 physics simulation to completion. Returns whether the goal was achieved, final
positions of all objects, and total simulation steps. If the placement is invalid
 (out of bounds or overlaps), returns a detailed error with how far to move the
ball.
   Arguments: x (float), y (float), radius (float)
   Usage: Action: simulate_action
          Action Input: {"x": 0.5, "y": 4.0, "radius": 0.6}

3. get_contact_log
   Description: After running a simulation, returns the contact events: which
objects touched and when.
   Arguments: None
   Usage: Action: get_contact_log

4. simulate_partial
```

```
   Description: Place a red ball and run the simulation only up to the specified
step. Returns object positions and velocities at that point. Useful for observing
 mid-simulation dynamics.
   Arguments: x (float), y (float), radius (float), stop_step (int)
   Usage: Action: simulate_partial
          Action Input: {"x": 0.5, "y": 4.0, "radius": 0.6, "stop_step": 50}

5. compute_cliffhanger_analysis
   Description: Analyse the cliffhanger geometry. Returns the green bar's centre,
 length, and the (x, y) coordinates of its bottom point (resting on the platform)
 and top point (opposite end); the platform's left/right extents and top-surface
y; the ceiling y and purple-ground y; the bar's distance to each platform edge;
which edge is closer (LEFT or RIGHT) --- i.e. the edge the bar must fall past ---
 and the width of the falling gap on that side.
   Arguments: None
   Usage: Action: compute_cliffhanger_analysis

6. finish
   Description: Submit your final answer. Use this when you are confident in your
 solution.
   Arguments: x (float), y (float), radius (float)
   Usage: Action: finish
          Action Input: {"x": 0.5, "y": 4.0, "radius": 0.6}


To solve this puzzle, you will reason step-by-step and use tools to test your
ideas.

At each step, you MUST follow this exact format:

Thought: <your reasoning about what to do next>
Action: <tool name>
Action Input: <JSON arguments, or leave blank for tools with no arguments>

After you take an action, you will receive:

Observation: <result from the tool>

Then you continue with another Thought/Action cycle.

When you are confident in your answer, use the "finish" tool to submit it.

Important rules:
- Always start with a Thought before taking an Action.
- Only call ONE tool per step.
- Parse observation results carefully before your next thought.
- You can simulate multiple different actions to compare results.
- Each simulation resets the environment, so previous simulations don't affect
new ones.
```

H.8 Tipping Point

```
You are an expert physics reasoning agent solving a 2D physics puzzle.

Puzzle: Tipping Point

The environment is a 2D box with coordinates ranging from -5 to 5 on both axes.
Gravity pulls objects downward.

Key Elements (factual):
- Green Bar: dynamic vertical bar (length 2.0--5.0) resting upright in a small
gray basket on the ground.
- Gray Basket: small dynamic basket holding the green bar near the bottom of the
box.
- Purple Wall: static vertical bar flush against the LEFT (x ~= -4.9) or RIGHT (x
 ~= 4.9) side of the box; spans top to bottom.
```

```
- Box bounds: x in [-5, 5], y in [-5, 5].

Goal: Place ONE Red Ball so that green_bar contacts purple_wall and maintains
contact for the success-time duration. The path to that outcome is yours to
design.

Placement Constraints:
- Red ball must be inside the box: -5 + radius <= x <= 5 - radius, -5 + radius <=
 y <= 5 - radius.
- No overlap with existing objects at t=0.
- 0.1 <= radius <= 2.0

When your simulation shows green_bar in continuous contact with purple_wall for
the required duration, call finish.

You have access to the following tools to interact with the physics simulation:


1. get_level_state
   Description: Get the current level layout including all object positions,
sizes, and properties.
   Arguments: None
   Usage: Action: get_level_state

2. simulate_action
   Description: Place a red ball at (x, y) with the given radius and run the full
 physics simulation to completion. Returns whether the goal was achieved, final
positions of all objects, and total simulation steps. If the placement is invalid
 (out of bounds or overlaps), returns a detailed error with how far to move the
ball.
   Arguments: x (float), y (float), radius (float)
   Usage: Action: simulate_action
          Action Input: {"x": 0.5, "y": 4.0, "radius": 0.6}

3. get_contact_log
   Description: After running a simulation, returns the contact events: which
objects touched and when.
   Arguments: None
   Usage: Action: get_contact_log

4. simulate_partial
   Description: Place a red ball and run the simulation only up to the specified
step. Returns object positions and velocities at that point. Useful for observing
 mid-simulation dynamics.
   Arguments: x (float), y (float), radius (float), stop_step (int)
   Usage: Action: simulate_partial
          Action Input: {"x": 0.5, "y": 4.0, "radius": 0.6, "stop_step": 50}

5. compute_tipping_point_analysis
   Description: Analyse tipping_point geometry. Returns the green bar's centre,
length, angle, and the (x, y) coordinates of its top and bottom endpoints; the
basket's centre and floor; the purple wall's x position and its top/bottom y; the
 purple wall's side relative to the green bar (LEFT or RIGHT); the horizontal
distance from the bar's centre to the wall; the angle the bar must tip through to
 touch the wall (assuming it pivots near the basket); and the suggested tip
direction (LEFT or RIGHT).
   Arguments: None
   Usage: Action: compute_tipping_point_analysis

6. finish
   Description: Submit your final answer. Use this when you are confident in your
 solution.
   Arguments: x (float), y (float), radius (float)
   Usage: Action: finish
          Action Input: {"x": 0.5, "y": 4.0, "radius": 0.6}
```

```
To solve this puzzle, you will reason step-by-step and use tools to test your
ideas.

At each step, you MUST follow this exact format:

Thought: <your reasoning about what to do next>
Action: <tool name>
Action Input: <JSON arguments, or leave blank for tools with no arguments>

After you take an action, you will receive:

Observation: <result from the tool>

Then you continue with another Thought/Action cycle.

When you are confident in your answer, use the "finish" tool to submit it.

Important rules:
- Always start with a Thought before taking an Action.
- Only call ONE tool per step.
- Parse observation results carefully before your next thought.
- You can simulate multiple different actions to compare results.
- Each simulation resets the environment, so previous simulations don't affect
new ones.
```

I Evolver Prompts for HEXA and Cross-Level Transfer

This appendix reproduces, verbatim, the prompts presented to the evolver that drive every skill-distillation result in this paper. Placeholder fields shown in curly braces (`{level_name}`, `{success_block}`, `{n_sources}`, etc.) are filled in at run time from the level identifier, the rendered trajectory blocks, and other context. Double-braces `{{...}}` appear as single braces at run time and are used to denote literal JSON braces inside the prompt body.

I.1 Pass 1 Contrastive Skill Distillation

Every HEXA round invokes this prompt to extract *strategy skills* by contrasting high-reward against low-reward trajectories within the same level batch. The reward function from Eq. 6 is communicated to the teacher as a per-trajectory `Reward` annotation embedded in the rendered success and failure blocks.

```
You are an expert physics analyst distilling agent behavior into concise,
actionable skills.

**Environment**: 2D physics simulation (Box2D). Gravity = -9.8 m/s^2. World
bounds [-5, 5] on both axes. The agent places a red ball at (x, y) with a given
radius, then the simulation runs to completion.

**Level: {level_name}**
{level_description}

Below are SUCCESSFUL and FAILED trajectories from the same level. Each shows the
agent's reasoning (Thought), actions taken, and simulation observations. Each
trajectory has a **Reward** score:
- Successes: +1.0 (solved fast, 1-3 iters) to +0.25 (solved slowly, 16-25 iters)
- Failures: -0.5 (tried all iterations) to -0.75 (gave up early)

**Weight your analysis by reward** -- skills from high-reward trajectories (fast
solves) are more reliable than skills from low-reward ones (barely solved).

=== SUCCESSFUL TRAJECTORIES ({n_successes}) ===
{success_block}

=== FAILED TRAJECTORIES ({n_failures}) ===
{failure_block}

---

Your task: By CONTRASTING the successes and failures, extract the KEY PHYSICS
SKILLS that distinguish solving from failing. Focus especially on high-reward
successes -- what insight let the agent solve it quickly? What did the failed
agents miss?

For each skill, provide:
- **title**: Short name (3-7 words)
- **principle**: The physics insight (2-3 sentences). What mechanism was
exploited? Why does it work?
- **when_to_apply**: Specific trigger condition (1 sentence)
- **source_seeds**: List of seed numbers from the trajectories above that this
skill was primarily derived from. Include only the seeds whose behavior directly
demonstrates or motivates this skill.

Output a JSON array:
```json
[
  {{
    "title": "...",
    "principle": "...",
    "when_to_apply": "...",
    "source_seeds": [1, 5, 16]
  }}
```
```

```
]
```

Extract 4-6 skills. Each should capture a DISTINCT insight from the success/
failure contrast. Avoid redundancy.

I.2 Pass 2 Mistake & Partial-Skill Extraction

Run on the failure trajectories $\mathcal{T}_\ell^{(n),-}$ alone. Produces (i) structured *mistake* records describing recurring failure modes, and (ii) *partial skills* extracted from individual correct steps inside otherwise-failed trajectories.

```
You are an expert at analyzing agent failures and distilling them into avoidable
mistake patterns.

**Environment**: 2D physics simulation (Box2D). Gravity = -9.8 m/s^2. World
bounds [-5, 5].

**Level: {level_name}**
{level_description}

Below are FAILED trajectories. Each shows the agent's reasoning, actions, and
simulation results.

{failure_block}

---

Your task has TWO parts:

**Part 1 -- Mistakes**: Identify the COMMON MISTAKE PATTERNS across these
failures. For each mistake, analyze:
1. What exactly the agent did wrong
2. WHY the agent made this error (what broken causal belief led to it)
3. A concrete actionable fix

**Part 2 -- Partial insights**: Even in failed trajectories, some individual
steps show CORRECT physics reasoning or useful discoveries (e.g., the agent found
 a valid placement region but then abandoned it, or correctly identified a
mechanism but applied it with wrong parameters). Extract 1-2 skills from these "
good steps within bad trajectories". These should be genuine physics insights,
not just restating what went wrong.

Format as JSON object with two arrays:
```json
{{
  "mistakes": [
    {{
      "description": "What the mistake is (1 sentence)",
      "why_it_happens": "The broken belief or reasoning error that causes this (1
 sentence)",
      "how_to_avoid": "Concrete actionable fix -- what to do instead (1-2
sentences)"
    }}
  ],
  "partial_skills": [
    {{
      "title": "Short name (3-7 words)",
      "principle": "The physics insight from the failed trajectory (2-3 sentences
)",
      "when_to_apply": "Specific trigger condition (1 sentence)",
      "source_seeds": [5, 11]
    }}
  ]
}}
```
```

```
```

Extract 3-5 mistakes and 1-2 partial skills. For mistakes, group similar failures
 into one and focus on ROOT CAUSES. For partial skills, only extract genuinely
useful insights -- do not force it if no good steps exist.
```

### I.3 Skill-Bank Evolution

HeXA employs the Skill Evolution method in each round $n \geq 1$, with the current bank $\mathcal{K}_\ell^{(n)}$ and the new trajectory batch $\mathcal{T}_\ell^{(n)}$ as inputs, which yields a new bank $\mathcal{K}_\ell^{(n+1)}$ bounded by the skill limit. The skill evolver determines which previously learned skills should be kept (with ~~confidence~~[Varun: reward] values maintained), which should be discarded, and which should be acquired (with confidence value calculated using Eq. 7).

```
You are a physics teacher evolving a skill bank for puzzle-solving agents.

LEVEL: {level_name}
LEVEL DESCRIPTION: {level_description}

CURRENT SKILL BANK (from previous rounds):
{existing_skills_block}

NEW TRAJECTORIES (from the latest round):
Successes: {n_successes}
Failures: {n_failures}

{new_trajectories_block}

YOUR TASK: Evolve the skill bank by merging the existing skills with insights
from the new trajectories.

RULES:
1. Output the COMPLETE FINAL skill bank (not a diff) -- include both retained
existing skills and any new ones.
2. Hard constraints:
   - Maximum {max_skills} total skills for this level
   - Maximum {max_mistakes} total mistakes for this level
3. For each skill you include:
   - If it's a RETAINED skill from the existing bank: set "is_new": false
   - If it's a NEW skill extracted from the new trajectories: set "is_new": true
   - Include "source_seeds" listing seed numbers where this skill was observed (
required for confidence calibration)
   - Include "confidence": a float in [0.1, 1.0] representing your confidence in
this skill
4. For retained skills, preserve their existing confidence values (they've been
validated).
5. For new skills, estimate confidence based on:
   - Success rate among source trajectories (high success = high confidence)
   - Universality (applies to multiple seed conditions = higher confidence)
   - Clarity and actionability of the principle
6. Do not include duplicate skills. If a new trajectory confirms an existing
skill, keep the existing one (possibly with slightly higher confidence).
7. Remove skills that are:
   - Redundant or subsumed by other skills.
   - Contradicted by the new trajectories
   - Too specific or rarely applicable
   - Low confidence (< 0.3) and not directly observed in new trajectories
8. Do not remove mistakes unless the new trajectories show they're no longer
common.

OUTPUT JSON OBJECT:
{{
  "skills": [
    {{
      "title": "<short name of skill>",
```
```

```
      "principle": "<2-3 sentence physics insight>",
      "when_to_apply": "<condition for applicability>",
      "example": "<optional concrete coordinate example>",
      "source_seeds": [<seed numbers>],
      "confidence": <float in [0.1, 1.0]>,
      "is_new": <true|false>
    }}
    ...
  ],
  "mistakes": [
    {{
      "description": "<what the mistake is>",
      "why_it_happens": "<why agents make this mistake>",
      "how_to_avoid": "<actionable fix>",
      "is_new": <true|false>
    }}
    ...
  ],
  "removed_skill_titles": ["<title of removed skill 1>", ...],
  "reasoning": "<brief explanation of key changes: what was removed, what was
added, why>"
}}

Be concise but precise. Focus on physics insights that directly help puzzle-
solving.
```

I.4 Catapult-Specific Variants (Claude)

Here we provide an example of Skill Evolution prompts specific to the catapult level.

Shared catapult scene description.

```
**Scene (factual -- no approach implied):**
- Green Ball: small dynamic ball on the LEFT end of a gray bar.
- Gray Bar (Catapult Arm): dynamic lever resting on a gray ball (pivot); the
green ball sits on its left end.
- Gray Ball (Pivot): dynamic ball acting as the fulcrum; sits on the left black
platform.
- Black Ball (Ceiling Blocker): static ball near the top of the scene.
- Black Platform (Left): static horizontal platform on the left side.
- Black Ledge (Right): static (possibly angled) platform on the right side.
- Basket (Gray): dynamic basket sitting on the right ledge.
- Blue Ball (Target): dynamic ball inside the basket.

**Success condition:** The green ball must contact the blue ball for at least 3
seconds after the red ball is placed and the simulation runs.

**Placement constraints:**
- The red ball must be completely inside the box: -5 + r <= x <= 5 - r, -5 + r <=
 y <= 5 - r.
- The red ball must NOT overlap with any existing object at t=0.
- 0.1 <= radius <= 2.0.
```

Pass 1 contrastive (catapult).

```
You are an expert physics analyst distilling agent behavior into concise,
actionable skills.

**Environment**: 2D physics simulation (Box2D). Gravity = -9.8 m/s^2. World
bounds [-5, 5] on both axes. The agent places a red ball at (x, y) with a given
radius, then the simulation runs to completion.

**Level: {level_name}**
[catapult scene description injected here -- see above]

Below are SUCCESSFUL and FAILED trajectories from this level. Each shows the
agent's reasoning (Thought), actions taken, and simulation observations. Each
trajectory has a **Reward** score:
- Successes: +1.0 (solved fast, 1-3 iters), +0.75 (4-7 iters), +0.5 (8-15 iters),
 +0.25 (solved slowly, 16-25 iters)
- Failures: -0.5 (tried all 25 iters), -0.75 (gave up early, <10 iters)

**Weight your analysis by reward** -- skills extracted from high-reward
trajectories (fast solves) are more reliable than those from low-reward ones.

=== SUCCESSFUL TRAJECTORIES ({n_successes}) ===
{success_block}

=== FAILED TRAJECTORIES ({n_failures}) ===
{failure_block}

---

Your task: By CONTRASTING the successes and failures, extract the KEY PHYSICS
SKILLS that distinguish solving from failing on this catapult level. Focus on
what high-reward successes DID that low-reward failures missed. Let the
trajectories -- not your prior expectations -- determine the mechanisms that work
.

For each skill, provide:
- **title**: Short name (3-7 words)
- **principle**: The physics insight (2-3 sentences). What mechanism was
exploited? Why does it work?
- **when_to_apply**: Specific trigger condition (1 sentence)
- **source_seeds**: Seed numbers from the trajectories above that this skill was
primarily derived from.

Output a JSON array:
```json
[
  {{
    "title": "...",
    "principle": "...",
    "when_to_apply": "...",
    "source_seeds": [1, 5, 16]
  }}
]
```

Extract 4-6 skills. Each should capture a DISTINCT insight from the success/
failure contrast. Avoid redundancy.
```

Pass 2 mistakes & partial skills (catapult).

```
You are an expert at analyzing agent failures and distilling them into avoidable
mistake patterns.

**Environment**: 2D physics simulation (Box2D). Gravity = -9.8 m/s^2. World
bounds [-5, 5].

**Level: {level_name}**
```

```
[catapult scene description injected here -- see above]

Below are FAILED trajectories. Each shows the agent's reasoning, actions, and
simulation results.

{failure_block}

---

Your task has TWO parts:

**Part 1 -- Mistakes**: Identify the COMMON MISTAKE PATTERNS across these
failures. For each mistake, analyze:
1. What exactly the agent did wrong
2. WHY the agent made this error (what broken causal belief led to it)
3. A concrete actionable fix

**Part 2 -- Partial insights**: Even in failed trajectories, some individual
steps show CORRECT physics reasoning or useful discoveries (e.g., the agent found
 a promising placement region but then abandoned it, or correctly identified a
mechanism but applied it with wrong parameters). Extract 1-2 skills from these "
good steps within bad trajectories". These should be genuine physics insights,
not just restating what went wrong.

Format as a JSON object with two arrays:
```json
{{
  "mistakes": [
    {{
      "description": "What the mistake is (1 sentence)",
      "why_it_happens": "The broken belief or reasoning error that causes this (1
 sentence)",
      "how_to_avoid": "Concrete actionable fix -- what to do instead (1-2
sentences)"
    }}
  ],
  "partial_skills": [
    {{
      "title": "Short name (3-7 words)",
      "principle": "The physics insight from the failed trajectory (2-3 sentences
)",
      "when_to_apply": "Specific trigger condition (1 sentence)",
      "source_seeds": [5, 11]
    }}
  ]
}}
```

Extract 3-5 mistakes and 1-2 partial skills. For mistakes, group similar failures
 into one and focus on ROOT CAUSES. For partial skills, only extract genuinely
useful insights -- do not force it if no good steps exist.
```

Skill-bank evolution (catapult).

```
You are a physics teacher evolving a skill bank for a catapult-puzzle-solving
agent.

**Environment**: 2D physics simulation (Box2D). Gravity = -9.8 m/s^2. World
bounds [-5, 5].

**Level: {level_name}**
[catapult scene description injected here -- see above]

CURRENT SKILL BANK (from previous rounds):
{existing_skills_block}

NEW TRAJECTORIES (from the latest round):
```

```
Successes: {n_successes}
Failures: {n_failures}

{new_trajectories_block}

YOUR TASK: Evolve the skill bank by merging the existing skills with insights
from the new trajectories.

RULES:
1. Output the COMPLETE FINAL skill bank (not a diff) -- include both retained
existing skills and any new ones.
2. Hard constraints:
   - Maximum {max_skills} total skills for this level.
   - Maximum {max_mistakes} total mistakes for this level.
3. For each skill you include:
   - If it's a RETAINED skill from the existing bank: set "is_new": false.
   - If it's a NEW skill extracted from the new trajectories: set "is_new": true.
   - Include "source_seeds" listing seed numbers where this skill was observed (
required for confidence calibration).
   - Include "confidence": a float in [0.1, 1.0] representing your confidence in
this skill.
4. For retained skills, preserve their existing confidence values (they've been
validated).
5. For new skills, estimate confidence based on:
   - Success rate among source trajectories (high success = high confidence).
   - Universality (applies across multiple seed conditions = higher confidence).
   - Clarity and actionability of the principle.
6. Do not include duplicate skills. If a new trajectory confirms an existing
skill, keep the existing one (optionally raising its confidence slightly).
7. Remove skills that are:
   - Redundant or subsumed by another skill.
   - Contradicted by the new trajectories.
   - Too specific or rarely applicable.
   - Low confidence (< 0.3) AND not observed in the new trajectories.
8. Do not remove mistakes unless the new trajectories show they're no longer
common.

OUTPUT JSON OBJECT:
{{
  "skills": [
    {{
      "title": "<short name of skill>",
      "principle": "<2-3 sentence physics insight>",
      "when_to_apply": "<condition for applicability>",
      "example": "<optional concrete coordinate example>",
      "source_seeds": [<seed numbers>],
      "confidence": <float in [0.1, 1.0]>,
      "is_new": <true|false>
    }}
  ],
  "mistakes": [
    {{
      "description": "<what the mistake is>",
      "why_it_happens": "<why agents make this mistake>",
      "how_to_avoid": "<actionable fix>",
      "is_new": <true|false>
    }}
  ],
  "removed_skill_titles": ["<title of removed skill 1>", ...],
  "reasoning": "<brief explanation of key changes: what was removed, what was
added, why>"
}}

Be concise but precise. Focus on physics insights grounded in the trajectories.
```

I.5 Skill Transfer Prompts

We use this prompt for skill transfer from easier levels to a harder level on Claude Sonnet 4.6 where the skill banks of (`down_to_earth`, `two_body_problem`, `pass_the_parcel`) are given to the model and prompted to frame reusable skills on catapult. The eight constraints—citation necessity, no fictional coordinates, entities only on the target side, no platitudes, transfer explanation necessity, confidence correction, no redundancies, mistakes constraints—are all listed in detail to ensure that each synthesized skill can be traced back to its source skill IDs.

```
You are an expert physics analyst performing CROSS-LEVEL OFFLINE SKILL TRANSFER.

You will be given expert skill banks from {n_sources} SOURCE physics levels.
Each source bank was distilled and evolved over many rounds of an agent solving
that level. You will also be given a factual description of a TARGET level
that the agent has NEVER attempted yet -- no target-level trajectories,
no target-level skills, no target-level success/failure data exist.

Your task: synthesise a TARGET skill bank by extracting transferable physics
PRINCIPLES from the source banks, re-grounding them in the target scene, and
predicting which principles the target level will reward.

============================================================================
**Environment (shared across all levels)**: 2D physics simulation (Box2D).
Gravity = -9.8 m/s^2. World bounds [-5, 5] on both axes. The agent places ONE
red ball at (x, y) with a chosen radius (0.1 <= r <= 2.0) at t=0; the simulation
then runs to completion. Mass scales as r^3; momentum and impact force scale
with both mass and the ball's velocity at first contact.
============================================================================

=== SOURCE LEVELS ===

{source_blocks}

=== TARGET LEVEL: {target_level} ===

{target_block}
{structural_hint_block}
=== TASK ===

Produce a SKILL BANK for the target level. The bank must contain:
- 6 to 10 SKILLS (physics principles you predict will help on the target).
- 2 to 4 MISTAKES (anti-patterns transferable from source failures).

=== HARD CONSTRAINTS ===

1. **Every skill MUST cite source skills.** For each output skill, include
   `source_skills`: a non-empty list of objects of the form
   `{{"source_level": "<level>", "skill_id": "<id>"}}`, listing the source-bank
   entries that motivated this skill. If you cannot cite at least one source
   skill, do NOT emit the skill -- synthesise something else instead.

2. **No invented coordinates.** You have NEVER seen the target level solved.
   Do NOT predict specific (x, y, r) values that "work" for the target. The
   `example` field, if used, must describe a *qualitative* placement
   (e.g. "place near the end of the lever opposite the green ball, with
   r large enough to dominate the lever's mass") -- never specific numbers.

3. **Use target-level entities.** Phrase each skill in terms of the entities
   that exist in the TARGET scene description above. Do NOT mention source-
   level-only entities (e.g. "ramp", "inverted basket") in the principle or
   when_to_apply, even if those entities motivated the transfer. The
   `transfer_rationale` field is where you may name source entities to
   explain the analogy.

4. **No platitudes.** Skills like "use physics intuition", "consider gravity",
   "be careful" are forbidden. Each skill must name a SPECIFIC mechanism
   (e.g. "torque about a pivot scales with moment arm x force"; "the line of
   centres at first contact determines the post-collision velocity direction").

5. **Each skill must include a `transfer_rationale`** (1-2 sentences) explaining
   what physics primitive bridges the source observation to the target
   prediction. This is the audit trail for your synthesis.

6. **Confidence calibration.** Weight a skill's confidence by:
   - How many source banks corroborate the underlying primitive (more = higher).
   - How directly the primitive applies to the target scene (direct = higher).
```

```
   - How load-bearing the source skill was (high source confidence = higher).
   Confidence is a float in [0.1, 1.0]. Assign 0.7+ only when the primitive
   appears in >=2 source banks AND the target scene clearly invokes it.

7. **Avoid redundancy.** Each skill must capture a DISTINCT mechanism. Do not
   emit two skills that are paraphrases of the same idea.

8. **Mistakes**: derive each from source-bank mistakes or from the failure
   modes those mistakes imply. Apply the same hard constraints (cite sources,
   no invented coordinates, target-side entities, specific mechanism).

=== OUTPUT FORMAT (single JSON object) ===

```json
{{
  "skills": [
    {{
      "title": "<3-7 word name>",
      "principle": "<2-3 sentence physics insight, in target-level terms>",
      "when_to_apply": "<specific condition, in target-level terms>",
      "example": "<qualitative placement description, NO numbers -- or omit>",
      "confidence": <float in [0.1, 1.0]>,
      "source_skills": [
        {{"source_level": "<level>", "skill_id": "<id>"}}
      ],
      "transfer_rationale": "<1-2 sentences naming the bridging physics primitive
>"
    }}
  ],
  "mistakes": [
    {{
      "description": "<1 sentence in target-level terms>",
      "why_it_happens": "<1 sentence root cause>",
      "how_to_avoid": "<1-2 sentence concrete fix>",
      "source_skills": [
        {{"source_level": "<level>", "skill_id": "<id>"}}
      ],
      "transfer_rationale": "<1-2 sentences>"
    }}
  ],
  "synthesis_reasoning": "<2-4 sentences: which source levels contributed most
heavily, which primitives you treated as load-bearing, which source skills you
deliberately did NOT transfer and why>"
}}
```

Be precise and specific. Do not pad. Do not produce extra fields. Output ONLY
the JSON object inside a single fenced ```json block.
```

I.6 Cross-Level Synthesis on Qwen-7B

Used for the Falling Into Place and Two Body Problem cross level rows of Table 13, where the teacher is Qwen 2.5 7B. The slim variant drops audit fields (`source_skills`, `transfer_rationale`), reduces to three rules, and tightens form constraints on `principle` / `when_to_apply` / `example` to prevent the platitude-and-hallucination failure mode the long prompt produces on smaller models (e.g. inventing a lever on a level with no pivot). Output budget is reduced to 4–6 skills and 1–3 mistakes.

```
You are extracting transferable physics PRINCIPLES from one or more source level
skill banks for a NEW target level.

ENVIRONMENT: 2D physics (Box2D), gravity = -9.8 m/s^2, bounds [-5, 5]. Agent
places ONE red ball (x, y, radius 0.1-2.0) at t=0; simulation runs to completion.

=== SOURCE LEVELS ===
```

```
{source_blocks}

=== TARGET LEVEL: {target_level} ===

{target_block}

=== TASK ===

Output a skill bank for the target level. Produce 4-6 skills and 1-3 mistakes.

RULES (FOLLOW STRICTLY):

1. SKILLS ARE PRINCIPLES, NOT COMMANDS. Each skill states a physics observation,
mechanism, or relationship the agent can REASON FROM. Skills MUST NOT tell the
agent what to do. Each field has a strict FORM:
   - `principle`: starts with a condition or relationship ("When X holds, Y
occurs", "X scales with Y", "A and B differ in Z"). NEVER an imperative verb.
FORBIDDEN openings: place / position / put / drop / move / set / use / adjust.
   - `when_to_apply`: describes a REASONING context the agent enters ("When
evaluating whether...", "When comparing alternatives that..."). NEVER a target
outcome to achieve ("when you need to tip...", "when knocking the bar off...").
   - `example`: contrasts two physical scenarios or illustrates the principle
abstractly. NEVER a placement, never a specific scene action.

2. Each skill names a SPECIFIC mechanism. FORBIDDEN platitudes: "use physics", "
place strategically", "consider gravity", "adjust as needed", "be careful".

3. Use ONLY entities present in the TARGET scene above. Do not mention source-
only entities anywhere. If the target scene has no pivot, do NOT write skills
about levers or torque-around-a-pivot.

4. NO coordinates and NO placements anywhere in any field.

5. Mistakes describe MISCONCEPTIONS the agent might form (wrong mental models),
NOT actions to avoid. `how_to_avoid` is a CORRECTED UNDERSTANDING, not a
placement instruction.

OUTPUT FORMAT (single JSON object, fenced ```json block, no extra fields):
```json
{{
  "skills": [
    {{"title": "...", "principle": "...", "when_to_apply": "...", "example":
"...", "confidence": 0.X}}
  ],
  "mistakes": [
    {{"description": "<misconception, NOT an action>", "why_it_happens": "...", "
how_to_avoid": "<corrected understanding, NOT a placement command>"}}
  ],
  "synthesis_reasoning": "<1-2 sentences: which source primitive transferred>"
}}
```

Be concise. Output ONLY the JSON object.
```