Survive or Collapse: The Asymmetric Roles of Data Gating and Reward Grounding in Self-Play RL

Sophia Xiao Pu[1], **Zhaotian Weng**[1], **Chengzhi Liu**[1], **Jayanth Srinivasa**[2], **Gaowen Liu**[2], **William Yang Wang**[1], **Xin Eric Wang**[1]
[1]University of California, Santa Barbara [2]Cisco Research

Abstract

Self-play reinforcement learning trains language models on their own generated tasks, co-evolving a proposer and solver without human labels. Recent systems report strong reasoning gains, but collapse and instability are widely observed and poorly understood. The dominant response treats this as a reward-design problem. We argue instead that self-play stability is governed by two distinct levers: a data-level *gate* that decides which proposer-generated tasks enter the training pool, and the *reward signal* that updates the policy on tasks already admitted. Through controlled experiments on a Python output-prediction task and a deterministic domain-specific language (DSL) twin task that strips pretraining priors, output ambiguity, and executor noise, we find the two levers are asymmetric. A strict gate is sufficient for stability under every reward variant we test, including a self-consistency reward with no access to ground truth; while no reward variant is sufficient once the gate is removed. This asymmetry exposes a counter-intuitive coupling we call the *Grounded Proposer Paradox*: a proposer with ground-truth access accelerates collapse *faster* than an ungrounded one when paired with a self-consistency solver, by concentrating training on clean tasks that form the fastest path to a spurious self-consistent attractor. Replacing the binary gate with a continuous strictness parameter ε further reveals a two-stage phase transition: training-side metrics decouple at low ε, while validation accuracy holds until ε is much higher. Data-level gating, not reward calibration, is the binding constraint on self-play stability.

Code: `https://github.com/UCSB-AI/self-play-data-gating`

1 Introduction

Self-play reinforcement learning with verifiable rewards is reshaping how reasoning capability is acquired in large language models. In a typical self-play loop, a proposer generates candidate problems, a solver attempts to answer them, the answers are checked against an executable verifier, and both roles are updated via reinforcement learning on the resulting reward (see Figure 1(a) for a schematic overview). The loop closes the supervision pipeline: the system manufactures its own training curriculum and grades its own attempts, without human-labeled tasks. The Absolute Zero paradigm demonstrates this principle [Zhao et al., 2025], and recent extensions adapt the loop to retrieval [Lu et al., 2026], long-context reasoning [Yang et al., 2026], and variational task synthesis [Liang et al., 2025].

The reported gains are striking, but so are the failure modes. Training collapse is widely observed across self-play and self-training systems: proposers learn to emit degenerate tasks that exploit the training objective [Bailey et al., 2026], intrinsic rewards fail catastrophically when the model's prior confidence does not align with correctness [He et al., 2026], and self-rewarding setups exhibit pervasive instability [Shafayat et al., 2025, Liu et al., 2026]. The dominant response treats stability

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

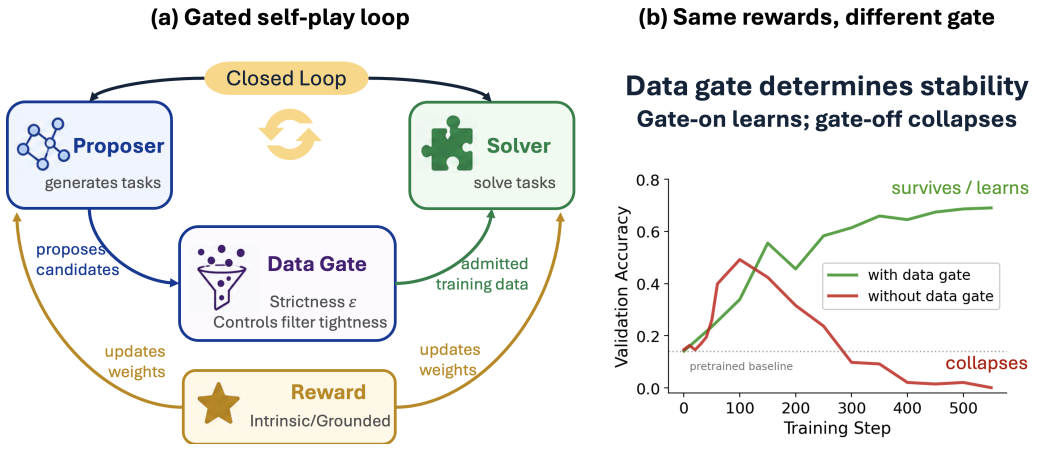

Figure 1: **Same rewards, different gate, opposite outcomes.** **(a)** In closed-loop self-play, the proposer generates candidate tasks for the solver, both updated via RL. A data gate decides which tasks enter the training pool. **(b)** Under the same intrinsic proposer and solver rewards, gate-on training improves over the baseline, whereas gate-off training collapses below it.

as a *reward design* problem: momentum anchors, confidence penalties, and other corrections to the reward signal [Xu et al., 2026, Wu and Tang, 2026], alongside analyses of how pretraining priors interact with intrinsic rewards [He et al., 2026]. The implicit assumption is that the right reward, applied to the data the proposer happens to generate, is what self-play stability requires.

We argue that this assumption misidentifies the *binding constraint*. A self-play system has two distinct levers that prior work tends to conflate. The first is a *data-level gate* that decides which proposer-generated tasks are admitted to the training pool. The second is the *reward signal* that shapes how the policy updates on tasks already admitted. The gate decides what data exists; the reward decides how the optimizer reacts to it. Both, in principle, could govern stability. The gate controls task validity, while difficulty, novelty, and diversity describe the learning value of the resulting curriculum. Our proposer reward already shapes task difficulty, but this curriculum objective does not replace the gate's validity checks.

Through controlled experiments on a Python output-prediction task and a synthetic twin task built on a deterministic 15-operator domain-specific language (DSL) that removes pretraining priors, output ambiguity, and executor noise by construction, we show that the two levers are *asymmetric* (Figure 1). With a strict gate, every reward variant we tested remains stable, including a self-consistency reward with no access to ground truth. Without the gate, no reward variant we tested is sufficient: even execution-grounded rewards fail to prevent collapse when the gate is removed. **The gate is *sufficient* for stability on its own; no reward variant is sufficient without it.** This is not a trivial consequence of training on low-quality data. One might expect the reward signal to naturally down-weight corrupted samples: bad data should receive low reward and contribute little to the gradient. Our experiments show this expectation is wrong. Under a self-consistency reward, the intrinsic-grounded gap (Section 3.1) shows that high agreement can coexist with near-zero grounded accuracy. Confidently wrong consensus can therefore receive maximal reward. Calibrating the reward while leaving the data pipeline open (the dominant approach in the current literature) addresses a downstream symptom rather than the binding constraint.

Three findings follow. *First*, removing the gate causes irreversible collapse regardless of reward design. With a self-consistency solver reward, the policy converges to a self-consistent attractor that is decoupled from correctness; we quantify this through the *intrinsic-grounded gap*, the difference between the solver's self-consistency reward and its grounded accuracy. The collapse replicates on the DSL twin, establishing it as a property of the optimization dynamics rather than the coding environment. *Second*, we identify a counter-intuitive coupling between the two roles, the *Grounded Proposer Paradox*: a proposer with access to ground-truth verification accelerates collapse *faster* than an ungrounded one when paired with a self-consistency solver, because it concentrates output on clean, well-structured tasks that form the lowest-resistance path to the spurious attractor. *Third*, replacing the binary gate with a continuous strictness parameter $\varepsilon \in [0, 1]$ reveals a sharp phase transition with

a two-stage structure: training-side metrics decouple at low ε, but downstream validation accuracy remains stable until ε is substantially higher.

Section 2 formalizes the self-play setup, the two axes, and the DSL control environment. Section 3 dissects the collapse mechanism, the Grounded Proposer Paradox, and the gate's role as the binding constraint. Section 4 maps the phase transition under continuous gate relaxation. Section 5 surveys related work and Section 6 concludes.

2 Formalizing the Asymmetric Dynamics in Self-Play

2.1 Roles and Objectives

We model the self-play training pipeline as an asymmetric two-role system. The Proposer generates problems. The Solver answers them. Both roles operate over the same task space $\mathcal{T}$, but they optimize different objectives.

Let π_ϕ denote the Proposer policy and π_θ the Solver policy [1]. At each outer step t:

1. The Proposer samples a task $\tau = (q, \cdot)$ from $\pi_\phi(\cdot \mid c^{\text{prop}})$, where c^{prop} is the proposer prompt template and $q = (\texttt{program}, \texttt{input})$ is a candidate problem.
2. A deterministic environment E executes the program on the input. If execution succeeds, E returns the ground-truth output $o^*(q)$. The full task tuple is $\tau = (q, o^*(q))$.
3. A data gate $F : \mathcal{T} \to \{0, 1\}$ decides whether τ enters the training pool $\mathcal{D}_t$. We formalize F in Section 2.3.
4. The Solver samples a candidate answer $a \sim \pi_\theta(\cdot \mid c^{\text{solv}}(q))$, where c^{solv} is the solver prompt template containing q.
5. Both roles receive rewards (in Section 2.2) and the policy parameters are updated jointly via GRPO.

Optimization objectives. The Solver maximizes expected reward over tasks drawn from the training pool:

$$\mathcal{J}_S(\theta) = \mathbb{E}_{\tau \sim \mathcal{D}_t, a \sim \pi_\theta(\cdot \mid c^{\text{solv}}(q))}\big[R_S(a, \tau)\big]. \tag{1}$$

The Proposer maximizes the difficulty its tasks pose to the current Solver:

$$\mathcal{J}_P(\phi) = \mathbb{E}_{\tau \sim \pi_\phi(\cdot \mid c^{\text{prop}})}\big[R_P(\tau, \pi_\theta)\big]. \tag{2}$$

The asymmetry is structural. The Proposer optimizes a curriculum-shaping objective. The Solver optimizes a task-completion objective. These two objectives are coupled only through the training pool $\mathcal{D}_t$ and the dependence of R_P on π_θ.

2.2 Reward Signals

Solver reward. Let $A(q) = a^{(1)}, \ldots, a^{(n)}$ denote the GRPO rollout group of size n for a single solver prompt $c^{\text{solv}}(q)$. We use $n = 16$ in all experiments.

The *grounded* solver reward checks each answer against the executor's ground-truth output:

$$R_S^{\text{g}}(a, \tau) = \mathbf{1}\big[\texttt{eval}(a) = \texttt{eval}(o^*(q))\big]. \tag{3}$$

The *intrinsic* reward replaces the executor reference with intra-group agreement. Let $\kappa(a)$ be a canonical form of a. For each answer $a^{(i)}$ in the group:

$$R_S^{\text{i}}(a^{(i)}, A(q)) = \frac{1}{n} \sum_{j=1}^{n} \mathbf{1}\big[\kappa(a^{(j)}) = \kappa(a^{(i)})\big]. \tag{4}$$

The reward equals the answer's agreement share within the group. It is continuous in $[0, 1]$. A response that agrees with k out of n rollouts (including itself) earns reward k/n. The minimum is $1/n$ for a singleton; the maximum is 1 when all rollouts agree.

[1] Following common practice in AZR-style frameworks, our implementation uses a single model with two prompt templates rather than two independent networks.

Proposer reward. The Proposer reward depends on the Solver's pass rate on the Proposer's own task. Let $\hat{\alpha}(\tau, \pi_\theta)$ be the empirical Solver accuracy on τ, estimated from $n_S = 8$ fresh Solver rollouts:

$$\hat{\alpha}(\tau, \pi_\theta) = \frac{1}{n_S} \sum_{j=1}^{n_S} R_S^{\mathrm{g}}\big(a^{(j)}, \tau\big), \qquad a^{(j)} \sim \pi_\theta(\cdot \mid c^{\mathrm{solv}}(q)).$$

The proposer reward is

$$R_P(\tau) = 1 - \hat{\alpha}(\tau, \pi_\theta),$$

The *grounded* and *intrinsic* Proposer variants differ only in what serves as the gold output during this estimation:

- *Grounded* Proposer: $\hat{\alpha}$ uses the executor output $o^*(q)$ as the reference.
- *Intrinsic* Proposer: $\hat{\alpha}$ uses the Proposer's own self-claimed output $\tilde{o}(q)$ embedded in the prompt response, instead of $o^*(q)$.

2.3 Data Gating

The system maintains an online training pool $\mathcal{D}_t$ of admitted tasks. Each step, the Proposer's candidates enter $\mathcal{D}_t$ only if they pass the data gate F. The Solver samples its training batch from a recent-biased mixture of the full pool, not from the latest batch alone.

The gate $F : \mathcal{T} \to \{0, 1\}$ determines whether a proposed task is admitted. Two natural endpoints define a binary design space: an **exec** mode admits τ only if the program executes successfully and produces a deterministic output across two repeated runs; an **off** mode admits every parseable response. We replace this binary choice with a one-parameter family. Let $\mathrm{exec}(\tau) = 1$ when the executor produces a deterministic output and 0 otherwise. Define

$$F_\varepsilon(\tau) = \begin{cases} 1 & \text{if } \mathrm{exec}(\tau) = 1, \\ \xi_\tau & \text{if } \mathrm{exec}(\tau) = 0, \end{cases} \qquad \xi_\tau \sim \mathrm{Bernoulli}(\varepsilon). \tag{5}$$

The gate operates on parseable candidates: $\varepsilon = 0$ recovers `exec`, while $\varepsilon = 1$ has the same admission rule as `off`, accepting every parseable candidate, including those that fail execution. ε controls the fraction of execution-failed tasks that the gate admits. We sweep ε in Section 4 to characterize the phase transition between stable and collapsing regimes.

2.4 Tasks

We instantiate the formal pipeline of Sections 2.1–2.3 on two tasks. The coding task asks the Solver to predict the printed output for a Proposer-supplied Python function and input. The DSL task has the same structure on a 15-operator prefix-expression language with two integer variables, where the output is always a single integer and the interpreter is deterministic. Appendix B gives the full grammar.

The coding task is the natural setting for self-play on code and matches the design of prior work [Zhao et al., 2025]. The DSL task is a controlled twin designed to remove three confounders present in the coding task: pretraining priors over the syntax, output-string ambiguity (floats, set order, side effects), and executor noise. Section 3 uses the DSL task to confirm that the observed collapse is driven by optimization dynamics, not by these environment-specific confounders.

3 Training Collapse: Empirical Mechanisms

This section presents the empirical evidence for the two-axis asymmetry introduced in Section 2. We make two claims. First, the data gate is the binding constraint on stability: removing it causes irreversible collapse regardless of reward design. Second, a proposer with access to ground-truth verification accelerates this collapse when paired with an intrinsic solver, a dynamic we call the *Grounded Proposer Paradox*. We present results on coding in the main text. A controlled DSL twin task replicates the key findings and rules out confounders present in the coding task; the full DSL analysis is in Appendix C.

Label	Proposer	Solver	Gate	Coding	DSL	Outcome
GG+exec	grounded	grounded	exec	0.71	0.61	stable
GI+exec	grounded	intrinsic	exec	0.67	0.63	stable
II+exec	intrinsic	intrinsic	exec	0.67	0.60	stable
GG+off	grounded	grounded	off	0.002	0.50	coding: collapse; DSL: baseline
IG+off	intrinsic	grounded	off	0.006	0.58	coding: collapse; DSL: baseline
GI+off	grounded	intrinsic	off	0.002	0.38	collapse
II+off	intrinsic	intrinsic	off	0.007	0.18	collapse

Table 1: Experimental matrix on coding and DSL tasks. Numerical entries summarize the original single-run experiments; multi-seed coding validation is reported in Figure 3. Gate-on runs learn on both tasks. Gate-off runs collapse on coding regardless of reward design. On DSL, only intrinsic-solver cells collapse; grounded-solver cells hold at baseline because the DSL interpreter cannot produce ambiguous outputs.

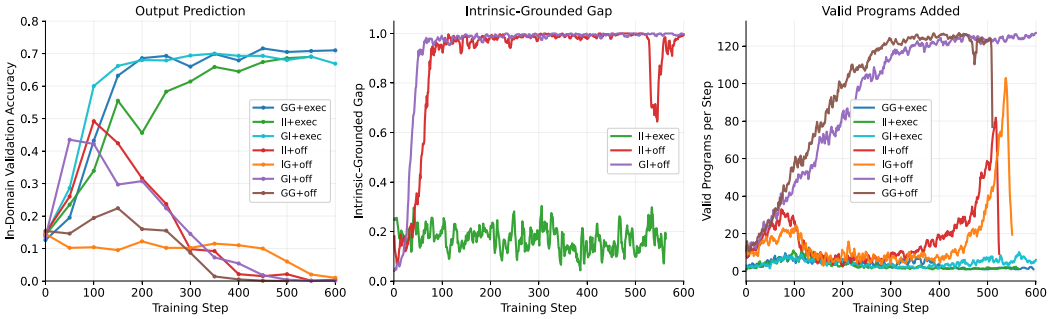

Figure 2: Experimental overview on the coding task. *Left:* in-domain validation accuracy. Gate-on runs learn; gate-off runs collapse to near zero. *Center:* intrinsic-grounded gap for intrinsic-solver runs. II+off and GI+off saturate at gap ≈ 1.0; II+exec stays near zero. *Right:* per-step valid programs admitted to the training pool. Pretrained baseline: 0.14.

Experimental matrix. We run 7 configurations on each task, varying proposer reward, solver reward, and gate setting. Table 1 lists the full matrix with results on both tasks. Run labels follow the convention **PS+gate**, where **P** is the proposer reward (G = grounded, I = intrinsic), **S** is the solver reward, and **gate** is `exec` or `off`. For coding we report in-domain validation accuracy (pretrained baseline ≈ 0.14). For DSL we report a stratified offline holdout (pretrained baseline ≈ 0.53; details in Appendix C).

3.1 The Data Gate as the Binding Constraint

Every gate-off coding configuration falls to near zero regardless of reward design (Table 1). Every gate-on configuration learns. The gate is the binding constraint on stability.

The intrinsic-solver cells (II+off, GI+off) collapse through a self-consistency attractor. Without the gate, GRPO's group-relative advantages make intra-group consensus the cheapest direction in policy space, regardless of correctness. Once all rollouts agree on any canonical answer, the self-consistency reward is maximized and the gradient vanishes. The optimizer has reached an attractor that is consistent with the reward but inconsistent with the task. This is reward hacking in the classical sense [Karwowski et al., 2024], but with a closed-loop twist: the spurious attractor propagates through the training pool to the proposer.

We measure this decoupling through the *intrinsic-grounded gap*: the difference between the solver's self-consistency reward and its grounded accuracy. For II+off and GI+off, the gap saturates at ≈ 1.0 within roughly 200 steps (Figure 2, center panel). Restoring the gate (**II+exec**) keeps the gap near zero. The gate alone is sufficient to prevent the attractor.

The grounded-solver cells (GG+off, IG+off) also collapse without the gate. Our audit of rejected proposer candidates shows that the gate primarily filters malformed or non-executable tasks (Appendix D).

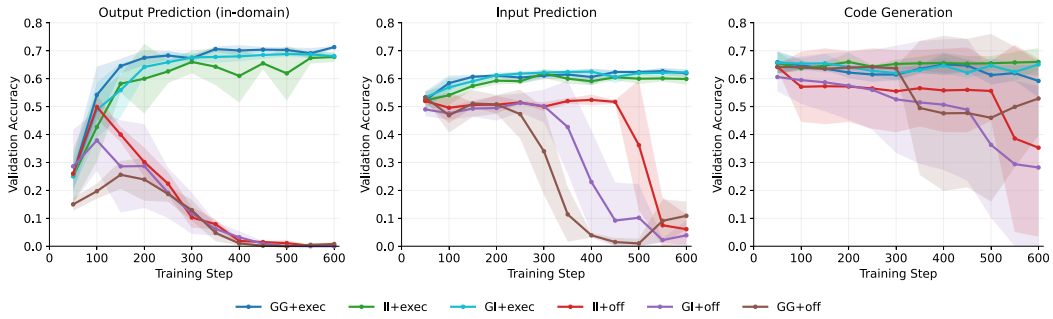

Figure 3: Validation accuracy across three matched training seeds for six coding configurations. Lines show means and shading shows ± 1 standard deviation, evaluated every 50 training steps. Gate-off runs deteriorate on both prediction tasks; code-generation degradation is more variable across seeds.

The collapse generalizes beyond the in-domain metric. Figure 3 reports multi-seed validation for six configurations on three evaluation tasks: output prediction (in-domain), input prediction, and code generation. The three evaluation datasets are derived from CRUXEval-O, CRUXEval-I [Gu et al., 2024], and HumanEval+/MBPP+[Liu et al., 2023], respectively. Gate-on runs maintain high output- and input-prediction accuracy, whereas gate-off runs deteriorate on both. Code-generation degradation varies substantially across seeds, so this off-domain aggregate can mask the severity of prediction-side collapse.

Identification. The DSL twin task replicates the intrinsic-solver collapse under identical conditions, ruling out pretraining priors, output ambiguity, and executor noise as causes (Appendix C). On DSL, where the interpreter is deterministic, the grounded-solver cells do not collapse. The contrast separates intrinsic-solver decoupling, which occurs on both tasks, from the grounded-solver failure observed on coding. The gate prevents both observed failure modes.

This contrast clarifies what the gate provides. On DSL, successful interpretation supplies a deterministic integer target, and grounded-solver cells remain near baseline even without the explicit gate. Python candidates can instead fail to execute or yield unreliable targets, and all four gate-off coding cells collapse. Appendix D characterizes the implemented gate's validity checks and the limits of its within-process determinism test.

3.2 The Grounded Proposer Paradox

GI+off pairs an intrinsic solver with a grounded proposer. **II+off** is the same but with an intrinsic proposer. The two runs differ only in proposer reward. GI+off enters the self-consistent attractor *faster than II+off*: it reaches high intrinsic-grounded gap in fewer steps on the training signal (Figure 2, center panel). Both runs end near zero on the in-domain metric (GI+off 0.002, II+off 0.007). The speed difference is in entry into the intrinsic-solver attractor, not a claim that GI+off is the earliest-collapsing run on every validation metric. The corresponding gate-on counterfactual, **GI+exec**, keeps validation accuracy high under the same grounded proposer and intrinsic solver, showing that the paradox is activated by removing the data gate rather than by grounded proposal itself.

Mechanism. A grounded proposer concentrates its output on syntactically clean, semantically sharp programs. For an intrinsic solver, these tasks form the lowest-resistance path to the self-consistent attractor of Section 3.1. The proposer does not bias the solver toward truth. It sharpens the corridor through which the solver reaches the spurious fixed point. This is distinct from classical Goodhart failure [Karwowski et al., 2024, Kwa et al., 2024], where a fixed proxy is exploited by a single optimizer. Here both agents co-evolve, and the upstream agent actively manufactures the structured inputs that accelerate the downstream failure.

Scope. The paradox concerns relative collapse speed under an intrinsic solver. Grounded-solver runs can also collapse on coding (Section 3.1), but they do not optimize the self-consistency objective underlying this comparison.

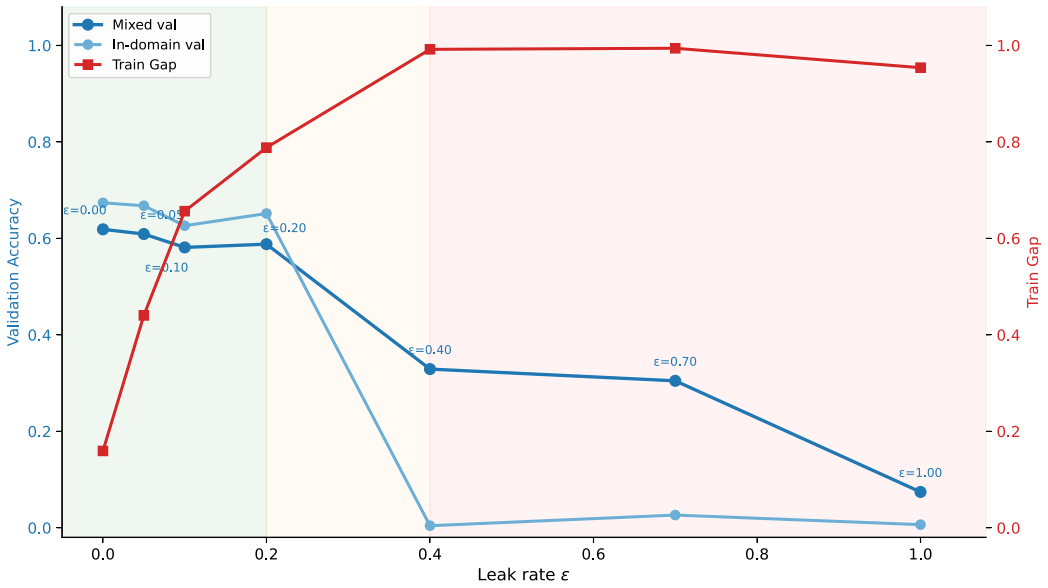

Figure 4: Phase transition under increasing gate leak rate ε (II configuration). Blue solid: mixed validation aggregate. Blue dashed: in-domain validation accuracy. Red dash-dot: training-side reward-grounded gap. Training-side decoupling begins at low ε. The in-domain probe reveals earlier hidden collapse at $\varepsilon = 0.40$, while the mixed aggregate remains near baseline until heavier corruption.

Section 4 replaces the binary gate with the continuous leak rate ε and traces the boundary between stable and collapsing regimes.

4 Phase Transitions under Continuous Gate Relaxation

Section 3 established that the execution gate is the binding constraint on stability in self-play. This section replaces the binary gate (on/off) with the continuous leak rate ε introduced in Section 2.3 and traces how system behavior changes as ε increases from 0 to 1. Two results emerge. First, a sharp phase transition separates a stable regime from a collapsing regime, and the transition has a two-stage structure in which training-side decoupling appears at much lower ε than validation-side collapse. Second, even in the stable regime, the system reaches a learning ceiling imposed by the proposer's finite capacity to generate novel problems.

4.1 Phase Diagram of Leakage Rate ε

We run the II configuration (intrinsic solver, intrinsic proposer) at seven values of $\varepsilon \in \{0.00, 0.05, 0.10, 0.20, 0.40, 0.70, 1.00\}$ and report late-stage means for two validation views (a mixed aggregate and an in-domain probe), the train-side gap, and train grounded accuracy. Figure 4 plots all three against ε.

Two-stage structure. The transition from stable to collapsed is not smooth. Two thresholds are visible, but the validation threshold depends on the evaluation view:

1. **Training-side decoupling ($\varepsilon^*_{\text{train}} \approx 0.05$).** At $\varepsilon = 0.05$ the train gap jumps from 0.16 to 0.44. The solver's intra-group consensus begins to diverge from grounded accuracy on the training data. Yet validation accuracy barely moves: $0.674 \to 0.668$.
2. **Validation-side degradation depends on the metric.** The in-domain probe collapses early: between $\varepsilon = 0.20$ and $\varepsilon = 0.40$, in-domain accuracy drops from 0.651 to 0.004. The mixed aggregate degrades later: it remains near baseline at $\varepsilon = 0.40$ (0.329 vs. step-0 0.313) and falls below baseline only by $\varepsilon = 0.70$ (0.305 vs. step-0 0.316). The aggregate metric masks a hidden failure mode that the in-domain probe already detects at moderate corruption.

The gap between these thresholds is the central observation of this section. Moderate gate corruption ($\varepsilon \leq 0.20$) causes training-level decoupling while both validation views remain above their step-0 baselines. By $\varepsilon = 0.40$, the in-domain metric has collapsed but the mixed aggregate still appears robust. Only heavier corruption ($\varepsilon \geq 0.70$) pushes the aggregate view into decline. The mixed aggregate averages over easier and off-domain cases, so it can remain above baseline even after the in-domain probe has begun to deteriorate.

Connection to Youden's index. The gate can be viewed as a binary classifier that retains valid programs (true positives) and rejects invalid ones (true negatives). Under our perturbation, the false positive rate equals ε: invalid programs are admitted with probability ε, while valid programs are always admitted. Youden's index $J = \text{TPR} - \text{FPR} = 1 - \varepsilon$. Rad et al. [2026] prove that RLVR under a noisy oracle transitions from bounded learning to anti-learning when J falls below zero. In our setting $J > 0$ for all $\varepsilon < 1$, so the Youden threshold alone does not predict either the earlier in-domain collapse at $\varepsilon = 0.40$ or the later aggregate degradation by $\varepsilon = 0.70$. ***The self-play loop tightens the effective threshold***: contaminated tasks enter the training pool, shift the proposer's distribution, and create a feedback cycle that amplifies noise beyond what a static-task analysis would predict. The effective critical J in self-play is substantially higher than zero.

4.2 The Proposer-Capacity Ceiling

Even in the stable regime, the system reaches a learning ceiling. For $\varepsilon = 0.00$, late-stage dataset eligibility drops to ≈ 0.007: fewer than 1% of proposer outputs enter the training pool. The GG+exec trajectory (Figure 2, left panel) shows this as a three-phase dynamic: rapid learning while the proposer contributes new problems (steps 1–160), slower gains from replay of pooled data (steps 160–440), and stagnation once replay is exhausted.

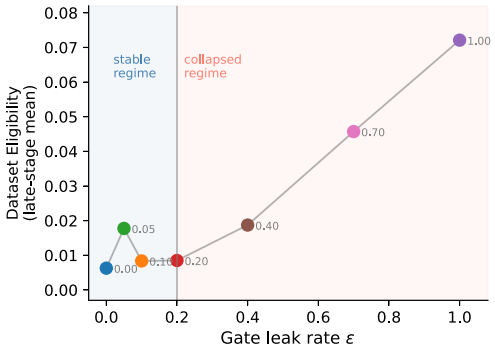

Figure 5: Dataset eligibility vs. gate leak rate ε. Flat in the stable regime ($\varepsilon \leq 0.20$); rises only with training collapse, reflecting solver degradation rather than exploration.

A natural conjecture is that a small non-zero ε trades signal quality for a broader pool, delaying stagnation. The phase diagram does not support this. $\varepsilon = 0.00$ attains the best late-stage score on both validation views (mixed 0.619, in-domain 0.674). Dataset eligibility stays flat across the stable regime ($\varepsilon \leq 0.20$) and rises only where training has already collapsed (Figure 5), confirming that the solver's difficulty-matching window, not the gate, is the binding constraint on pool size.

Adaptive schedule. A timing confound remains: each ε in the phase diagram is applied from step 0, so failure could be attributed to noisy gradients early in training. To rule this out, we trained with $\varepsilon = 0$ for 150 steps and then resumed with $\varepsilon = 0.05$. Figure 6 shows the result. Through step 400 the scheduled run tracks the baseline. By step 450 its in-domain accuracy has dropped to 0.420 while the baseline reaches 0.674, and the gap persists at step 500 (0.481 vs 0.686). Gate relaxation does not help even after the model has stabilized.

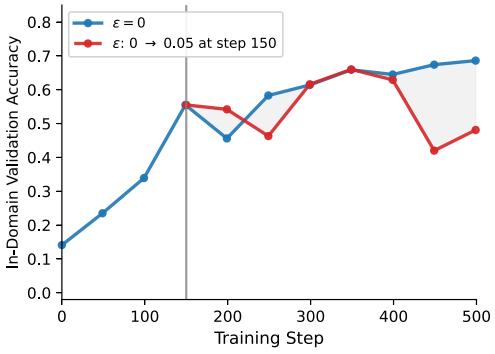

Figure 6: Adaptive schedule: strict gate ($\varepsilon = 0$) for 150 steps, then $\varepsilon = 0.05$ from the same checkpoint.

Practical implication. The strict gate ($\varepsilon = 0$) is optimal on both validation views we measured. The proposer-capacity ceiling is real but should be addressed through orthogonal means (curriculum

design, periodic seeding with external tasks) rather than through the validity gate.

5 Related Work

5.1 Self-Play with Verifiable Rewards

RLVR with rule-based environments and GRPO has driven gains in logical consistency [DeepSeek-AI et al., 2025], but verifiable benchmarks are finite, motivating self-play approaches that grow their own data. Absolute Zero closes this loop with a co-evolving generator and solver [Zhao et al., 2025]. Extensions target retrieval [Lu et al., 2026], policy diversity [Liang et al., 2025], long-context reasoning [Yang et al., 2026], and self-generated reward [Simonds et al., 2025].

The most closely related work is Self-Guided Self-Play [Bailey et al., 2026], developed concurrently. It identifies a Conjecturer reward-hacking failure and proposes a third Guide role that scores problems for relevance. The Guide is a policy-side soft signal during gradient computation. Our intervention is at a different layer: a hard filter on the data flow before the training pool. The two are orthogonal and could be combined.

5.2 Reward Hacking and Goodhart's Law

Reward hacking is a known failure mode in RL fine-tuning [Karwowski et al., 2024], and KL regularization can fail under heavy-tailed misspecification [Kwa et al., 2024]. In language models, corrupted reward signals activate memorization shortcuts [Yan et al., 2026], and code generation exhibits multi-phase rebound dynamics under reward hacking [Wu and Tang, 2026]. ACE penalizes overconfident trajectories to stabilize optimization [Xu et al., 2026].

He et al. [2026] analyze unsupervised RLVR and argue that intrinsic methods succeed when the model's prior confidence aligns with correctness and fail when it does not. Their account locates the cause in the pretraining prior. Ours is complementary: we hold the prior fixed and show that the gate leakage rate governs whether collapse occurs. The prior shapes which attractor the policy collapses toward; the gate decides whether the system reaches it. In the self-play setting, the proposer's curriculum can co-evolve to amplify the solver's spurious signal, the dynamic we call the Grounded Proposer Paradox (Section 3).

5.3 Stability under Noisy Verifiers

Rad et al. [2026] study RLVR under noisy oracles and prove a phase transition governed by Youden's index J: below $J = 0$ the system enters anti-learning. Plesner et al. [2026] report a related empirical finding: a verifier with up to 15% label noise recovers most of the clean baseline. Both study fixed task distributions.

Our setting moves the noise inside a live self-play loop, where contaminated tasks shift the proposer's distribution and create a feedback cycle that amplifies noise beyond static-task predictions. In our experiments $J > 0$ for all $\varepsilon < 1$, yet collapse occurs at $\varepsilon = 0.40$ (in-domain) and $\varepsilon = 0.70$ (aggregate), showing that the effective critical J in self-play is substantially higher than zero. We also report a two-stage structure (training-side decoupling before validation-side collapse) that has no analogue in the static-task setting.

6 Conclusion

The central finding of this work is that self-play stability is determined at the data level, not the reward level. A strict data filter is sufficient to prevent collapse regardless of reward design. No reward signal we tested is sufficient to prevent collapse without the filter.

This shifts the design question for self-play systems. The current literature focuses on building better reward signals: momentum anchors, confidence penalties, and hacking detectors. Our results suggest that the more productive axis is the data pipeline. What enters the training loop matters more than how the optimizer scores it once admitted.

Several questions remain open. Our phase diagram shows a gap between training-side decoupling and validation-side collapse. Why does validation remain robust when the training signal is already degraded? Replay-buffer regularization is a possible explanation for this gap. Whether the optimal filter strictness can be adapted during training, rather than set as a fixed hyperparameter, is a natural next step. Finally, the Grounded Proposer Paradox we identified may extend beyond self-play to any multi-agent system where one agent's output feeds another's training data.

Acknowledgments and Disclosure of Funding

This work was supported in part by a gift from Cisco.

References

Andrew Zhao, Yiran Wu, Yang Yue, Tong Wu, Quentin Xu, Yang Yue, Matthieu Lin, Shenzhi Wang, Qingyun Wu, Zilong Zheng, and Gao Huang. Absolute zero: Reinforced self-play reasoning with zero data, 2025. URL https://arxiv.org/abs/2505.03335.

Hongliang Lu, Yuhang Wen, Pengyu Cheng, Ruijin Ding, Jiaqi Guo, Haotian Xu, Chutian Wang, Haonan Chen, Xiaoxi Jiang, and Guanjun Jiang. Search self-play: Pushing the frontier of agent capability without supervision. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=ZmGirmNJqE.

Ziyi Yang, Weizhou Shen, Chenliang Li, Ruijun Chen, Fanqi Wan, Ming Yan, Xiaojun Quan, and Fei Huang. SPELL: Self-play reinforcement learning for evolving long-context language models. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=83F6YF4Hz6.

Xiao Liang, Zhong-Zhi Li, Yeyun Gong, Yang Wang, Hengyuan Zhang, Yelong Shen, Ying Nian Wu, and Weizhu Chen. Beyond Pass@1: Self-play with variational problem synthesis sustains RLVR. *arXiv preprint arXiv:2508.14029*, 2025. URL https://arxiv.org/abs/2508.14029.

Luke Bailey, Kaiyue Wen, Kefan Dong, Tatsunori Hashimoto, and Tengyu Ma. Scaling self-play with self-guidance. *arXiv preprint arXiv:2604.20209*, 2026. URL https://arxiv.org/abs/2604.20209.

Bingxiang He, Yuxin Zuo, Zeyuan Liu, Shangziqi Zhao, Zixuan Fu, Junlin Yang, Cheng Qian, Kaiyan Zhang, Yuchen Fan, Ganqu Cui, Xiusi Chen, Youbang Sun, Xingtai Lv, Xuekai Zhu, Li Sheng, Ran Li, Huan-ang Gao, Yuchen Zhang, Bowen Zhou, Zhiyuan Liu, and Ning Ding. How far can unsupervised RLVR scale LLM training? *arXiv preprint arXiv:2603.08660*, 2026. URL https://arxiv.org/abs/2603.08660.

Sheikh Shafayat, Fahim Tajwar, Ruslan Salakhutdinov, Jeff Schneider, and Andrea Zanette. Can large reasoning models self-train? *arXiv preprint arXiv:2505.21444*, 2025. URL https://arxiv.org/abs/2505.21444.

Wei Liu, Siya Qi, Yali Du, and Yulan He. Self-play only evolves when self-synthetic pipeline ensures learnable information gain. *arXiv preprint arXiv:2603.02218*, 2026. URL https://arxiv.org/abs/2603.02218.

Yuanda Xu, Hejian Sang, Zhengze Zhou, Ran He, and Zhipeng Wang. Overconfident errors need stronger correction: Asymmetric confidence penalties for reinforcement learning. *arXiv preprint arXiv:2602.21420*, 2026. URL https://arxiv.org/abs/2602.21420.

Rui Wu and Ruixiang Tang. When reward hacking rebounds: Understanding and mitigating it with representation-level signals. *arXiv preprint arXiv:2604.01476*, 2026. URL https://arxiv.org/abs/2604.01476.

Jacek Karwowski, Oliver Hayman, Xingjian Bai, Klaus Kiendlhofer, Charlie Griffin, and Joar Skalse. Goodhart's law in reinforcement learning. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=5o9G4XF1LI.

Alex Gu, Baptiste Rozière, Hugh Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida I. Wang. Cruxeval: A benchmark for code reasoning, understanding and execution. *arXiv preprint arXiv:2401.03065*, 2024.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatGPT really correct? rigorous evaluation of large language models for code generation. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=1qvx610Cu7.

Thomas Kwa, Drake Thomas, and Adrià Garriga-Alonso. Catastrophic Goodhart: Regularizing RLHF with KL divergence does not mitigate heavy-tailed reward misspecification. In *Advances in Neural Information Processing Systems*, volume 37, 2024. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/1a8189929f3d7bd6183718f42c3f4309-Abstract-Conference.html.

Ali Rad, Khashayar Filom, Darioush Keivan, Peyman Mohajerin Esfahani, and Ehsan Kamalinejad. Rate or fate? RLVεR: Reinforcement learning with verifiable noisy rewards. *arXiv preprint arXiv:2601.04411*, 2026. URL https://arxiv.org/abs/2601.04411.

DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, et al. DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025. URL https://arxiv.org/abs/2501.12948.

Toby Simonds, Kevin Lopez, Akira Yoshiyama, and Dominique Garmier. RLSR: Reinforcement learning from self reward. *arXiv preprint arXiv:2505.08827*, 2025. URL https://arxiv.org/abs/2505.08827.

Lecheng Yan, Ruizhe Li, Guanhua Chen, Qing Li, Jiahui Geng, Wenxi Li, Vincent Wang, and Chris Lee. Spurious rewards paradox: Mechanistically understanding how rlvr activates memorization shortcuts in llms, 2026. URL https://arxiv.org/abs/2601.11061.

Andreas Plesner, Francisco Guzmán, and Anish Athalye. An imperfect verifier is good enough: Learning with noisy rewards. *arXiv preprint arXiv:2604.07666*, 2026. URL https://arxiv.org/abs/2604.07666.

A Training Configuration

All reported runs share the configuration below unless otherwise noted. The configuration is implemented through a single hydra config file plus environment overrides; see the codebase for the exact files.

A.1 Model and optimization

Parameter	Value
Base model	Qwen3-4B-Base
Learning rate	10^{-6}
Optimizer	AdamW, default Adam betas
KL loss coefficient β_{KL}	10^{-2}
KL loss type	`low_var_kl`
Advantage estimator	GRPO with std normalization
PPO clip ratio	0.2
PPO epochs per outer step	1
Total training steps	up to 700

Parameter	Value
Per-step prompt batch (b_P, b_S)	8
GRPO rollout group size (n_P, n_S)	16
Inner solver rollouts for difficulty estimation (n_S^{diff})	8
Rollout sampling temperature	1.0
Max prompt length	3072 tokens
Max response length	2048 tokens
Training pool cap B	16,384
Buffer eviction	FIFO
Sampling strategy	recent + uniform mix from full buffer
Solver dataloader window	per-step recent batch + uniform fill
Initial seed pool $\lvert\mathcal{D}_0\rvert$	32 for the coding task, 24 for the DSL task

Parameter	Value
`program_validity.mode`	`execution` (headline runs) or `execution_noisy` (ε-gradient runs)
Determinism check	two repeated executions per program
Sandbox	per-program subprocess with 10 s timeout

A.2 Self-play loop

A.3 Gate variants

A.4 Hardware and runtime

B DSL Specification

The DSL is a small prefix-expression language used as a controlled twin to the coding task in Section 3.

B.1 Grammar

In BNF:

```
expr     ::= literal | variable | unop "(" expr ")"
           | binop "(" expr "," expr ")"
           | "ITE" "(" cond "," expr "," expr ")"
cond     ::= cmp "(" expr "," expr ")"
literal  ::= signed integer
variable ::= "x" | "y"
unop     ::= "NEG" | "ABS"
binop    ::= "ADD" | "SUB" | "MUL" | "DIV" | "MOD" | "MAX" | "MIN"
cmp      ::= "GT"  | "LT"  | "EQ"  | "GEQ" | "LEQ"
```

Fifteen operators in total: seven binary arithmetic, two unary arithmetic, one conditional, five comparators.

B.2 Operator semantics

B.3 Interpreter properties

The interpreter is a pure-Python evaluator $E_{\text{DSL}} : G_{\text{DSL}} \times \mathbb{Z}^2 \to \mathbb{Z} \cup \{\perp\}$ where G_{DSL} denotes the set of well-formed expressions and $\perp$ is the rejection symbol returned for runtime failures (such as division by zero on the supplied input).

By construction:

1. Every accepted expression and input pair has exactly one output.
2. The output is always a single integer or $\perp$.

Parameter	Value
Device	$8\times$ A100 40 GB (single node)
Tensor model parallel size	2
Sequence parallel size	4
PPO micro batch size per GPU	1
Rollout GPU memory utilization	0.4
Actor parameter offload	enabled
Actor optimizer offload	enabled

Operator	Semantics
`ADD(a, b)`	$a + b$
`SUB(a, b)`	$a - b$
`MUL(a, b)`	$a \cdot b$
`DIV(a, b)`	$\lfloor a/b \rfloor$ (a `//` b in Python). Rejected when $b = 0$.
`MOD(a, b)`	$a \bmod b$. Rejected when $b = 0$.
`MAX(a, b)`	$\max(a, b)$
`MIN(a, b)`	$\min(a, b)$
`NEG(a)`	$-a$
`ABS(a)`	$\lvert a \rvert$
`ITE(c, t, e)`	evaluates to t if c is true, else e
`GT(a, b)`	$a > b$
`LT(a, b)`	$a < b$
`EQ(a, b)`	$a = b$
`GEQ(a, b)`	$a \geq b$
`LEQ(a, b)`	$a \leq b$

3. There are no floats, collections, randomness sources, or side effects.

4. The interpreter is deterministic: repeated execution on the same input yields the same output.

B.4 Validation

A candidate expression is admitted to the training pool only if the interpreter returns a non-$\perp$ value on every point of a 5×5 probe grid over $(x, y) \in [-2, 2]^2$. This is the DSL analogue of the coding execution gate. During seed task construction, variable ranges are extended to $x, y \in [-10, 10]$ to produce a richer initial pool.

The Solver's grounded reward on the DSL task is exact integer equality after whitespace stripping. We do not use Python `eval` for comparison, since the DSL has no value ambiguity (`"1"` and `"1.0"` cannot both be valid outputs because the output type is constrained to integer).

B.5 Examples

```
ADD(x, y)                                          ; x + y
ABS(SUB(x, y))                                     ; |x - y|
ITE(GT(x, 0), x, NEG(x))                           ; abs(x) via conditional
ADD(MUL(x, x), MUL(y, y))                          ; x^2 + y^2
ITE(EQ(MOD(x, 2), 0), DIV(x, 2), ADD(MUL(x, 3), 1)) ; one Collatz step
```

B.6 Differences from the coding task

The DSL task is structurally identical to the coding task in the formal pipeline of Sections 2.1–2.3. The differences are confined to the executor and the answer-comparison rule:

These differences are exactly what Section 2.4 calls "three coding-specific confounders by construction."

Property	Coding task	DSL task
Executor	Python interpreter in subprocess sandbox	Pure-Python recursive evaluator
Output type	any printable value (string)	single integer
Comparison	`eval` on both sides (type-tolerant)	exact integer equality
Pretraining prior	substantial	none
Determinism check	two repeated runs	by construction
Timeout	10 s	not needed

C DSL Replication of Section 3 Results

This appendix presents the full DSL results referenced from Section 3. The DSL task removes three coding-specific confounders by construction: pretraining priors, output ambiguity, and executor noise. If the collapse mechanisms of Section 3 reproduce on DSL, they cannot be attributed to any of these confounders.

C.1 Overview

We run the same configurations as the coding task (Table 1). Figure 7 shows four diagnostic metrics across DSL runs.

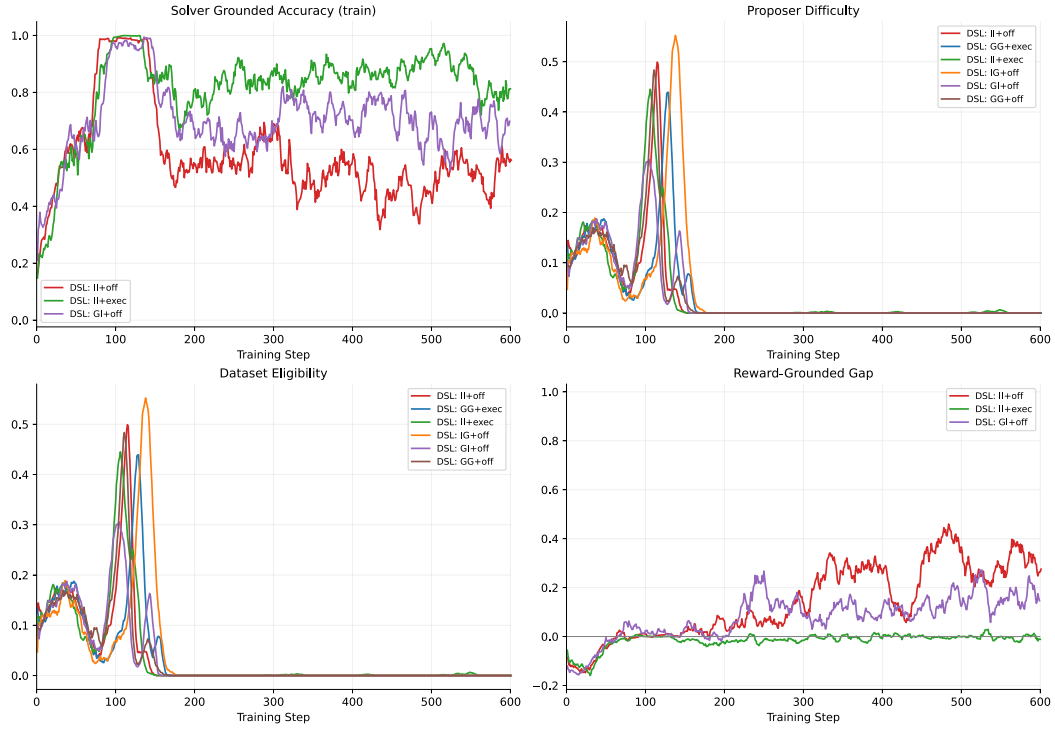

Figure 7: DSL experimental overview. Top left: solver grounded accuracy (train). Top right: proposer difficulty. Bottom left: dataset eligibility. Bottom right: reward-grounded gap. Gate-on runs (GG+exec, II+exec) learn; intrinsic-solver off-gate runs (II+off, GI+off) show rising gap; grounded-solver off-gate runs (GG+off, IG+off) remain near baseline.

C.2 Catastrophic decoupling replicates on DSL

The bottom-right panel of Figure 7 shows the reward-grounded gap for the intrinsic-solver DSL runs. II+off shows a rising gap, confirming the same directional pattern as the coding task. The absolute magnitude is smaller because the DSL's deterministic integer output limits the spurious attractor's reach. II+exec stays near zero, confirming that the gate prevents decoupling on DSL as well.

C.3 Grounded-solver cells do not collapse on DSL

On the coding task, all four off-gate cells collapse (Section 3.1). On DSL, only the intrinsic-solver cells collapse. The grounded-solver cells (GG+off, IG+off) remain within sampling noise of the pretrained baseline (≈ 0.53 on the d4-6 tier). This is because the DSL interpreter is deterministic and cannot produce ambiguous programs. When the solver is grounded, the deterministic interpreter already provides the gate's function (unambiguous outputs), making the explicit filter redundant.

C.4 Stratified holdout evaluation

Figure 8 reports the offline stratified holdout (depth 4-6, $n = 150$ per run) at checkpoint steps $0, 100, \ldots, 600$. The pretrained baseline is ≈ 0.53. Gate-on runs (GG+exec, II+exec) rise to ≈ 0.60. Grounded-solver off-gate runs stay near baseline. Intrinsic-solver off-gate runs fall below baseline, with II+off reaching 0.18.

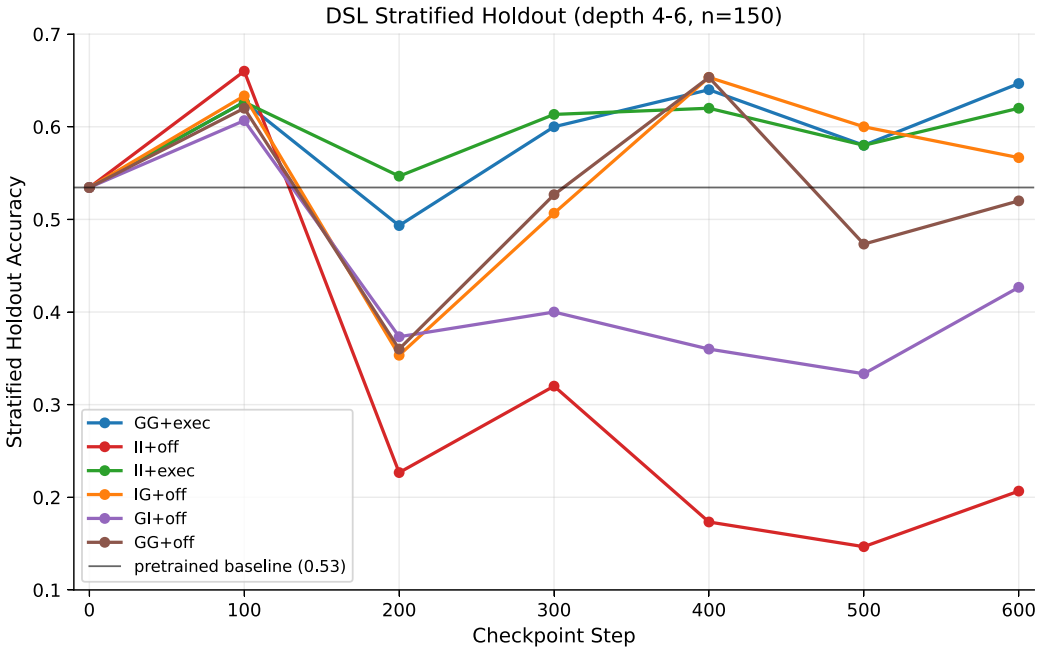

Figure 8: DSL stratified holdout accuracy (depth 4-6) across training. Gate-on runs learn. Grounded-solver off-gate runs hold at baseline. Intrinsic-solver off-gate runs collapse below baseline.

D Audit of Rejected Proposer Candidates

Audit protocol. We sample 500 raw proposer responses from each of nine checkpoints, for 4,500 candidates in total: the base model; GG+off at steps 50, 100, and 400; GG+exec at steps 100 and 400; II+off at steps 50 and 100; and II+exec at step 100. Each checkpoint uses its corresponding training-time replay-pool snapshot. Generation follows the training prompt construction, including six reference snippets, the 2,800-token content filter, the banned-word list, and the self-claimed output for intrinsic proposers. We reproduce the historical gate's two function calls within one process and compare their Python return values. The offline execution sandbox uses the training executor's 10-second CPU budget. Raw-response parsing precedes the execution gate and is required in both the `exec` and `off` settings.

Rejection taxonomy. Of the 4,500 candidates, 1,213 pass the gate, 3,283 are confirmed rejects, and four are excluded by conservative offline safety checks. Table 2 partitions the confirmed rejects into mutually exclusive categories. Parse failures, syntax failures, and runtime exceptions together account for 99.2% of confirmed rejections. Only one candidate is rejected because its two within-process

return values compare unequal. The observed rejection mass therefore primarily concerns malformed or non-executable tasks.

Rejection reason	Count	Share of confirmed rejects
Parse failure	2,372	72.25%
Syntax failure after parsing	420	12.79%
Runtime exception	466	14.19%
Banned operation or import	13	0.40%
Timeout	11	0.34%
Unequal within-process returns	1	0.03%
Total confirmed rejects	3,283	100.00%

Table 2: Rejection taxonomy for the raw-proposer audit. Percentages use the 3,283 confirmed rejects as the denominator. The 1,213 gate-passing candidates and four offline exclusions complete the 4,500-candidate sample.

Gate behavior. The gate compares Python values rather than their textual representations. In the audit, 28 gate-passing candidates return deterministic floating-point values, and five candidates pass despite changes in textual representation because their Python values remain equal. Thus, floating-point outputs and representation-only ordering changes are not rejected by construction. Additional cross-process probes identify eight gate-passing candidates whose semantic outputs depend on the Python hash seed. These pass the historical gate because its two calls share the same process. The implemented test checks repeated-return equality within a process; it does not guarantee invariance across processes or hash seeds.

NeurIPS Paper Checklist

1. **Claims**

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

   Answer: [Yes]

   Justification: The abstract and introduction state the three main findings, supported by Sections 3–4. Figure 3 reports multi-seed coding validation, Appendix C provides the controlled DSL comparison, and Appendix D characterizes the rejected proposer candidates.

   Guidelines:

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

2. **Limitations**

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

   Answer: [Yes]

   Justification: Section 6 discusses the gap between training-side decoupling and validation-side collapse, identifies replay-buffer regularization as a possible explanation, and discusses adaptive gate strictness as an open question. The reported experiments use Qwen3-4B-Base on coding and DSL tasks. Appendix D describes the limits of the within-process determinism check.

   Guidelines:

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

3. **Theory assumptions and proofs**

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

   Answer: [N/A]

   Justification: The paper is primarily empirical. It formalizes the self-play pipeline and the gate mechanism (Section 2) but does not claim formal theorems or proofs.

   Guidelines:

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

4. **Experimental result reproducibility**

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

   Answer: [Yes]

   Justification: Appendix A specifies all hyperparameters (learning rate, optimizer, KL coefficient, batch sizes, buffer size, rollout group size, hardware). Section 2 formalizes the gate, reward signals, and training loop. Appendix B provides the full DSL grammar and interpreter specification. The base model (Qwen3-4B-Base) is publicly available.

   Guidelines:

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

(d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

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

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

   Answer: [Yes]

   Justification: Code, training configurations, task data, and reproduction instructions are publicly available at `https://github.com/UCSB-AI/self-play-data-gating`. The repository documents environment setup and data preparation, including how to construct the code-generation evaluation data. Appendix B specifies the DSL grammar and interpreter.

   Guidelines:

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

6. **Experimental setting/details**

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

   Answer: [Yes]

   Justification: Appendix A provides complete training configuration: model, optimizer, learning rate, KL coefficient, batch sizes, rollout group sizes, buffer capacity, eviction policy, hardware, and parallelism settings. Section 2 specifies the reward formulations and gate mechanism. Appendix B details the DSL task.

   Guidelines:

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

7. **Experiment statistical significance**

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

   Answer: [Yes]

Justification: Figure 3 reports means and ± 1 standard deviation across three matched training seeds for six coding configurations. The error bands represent variability across training runs. Other experiments are reported as individual trajectories.

Guidelines:

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

8. **Experiments compute resources**

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

   Answer: [Yes]

   Justification: Appendix A specifies the hardware ($8\times$ A100 40 GB GPUs per run), memory offload, and parallelism settings; reported runs train for up to 700 steps. We evaluate seven configurations on each task, with three-seed coding validation for six configurations in Figure 3. Section 4 additionally reports seven leakage-rate settings and an adaptive schedule.

   Guidelines:

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

9. **Code of ethics**

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

   Answer: [Yes]

   Justification: The research studies training dynamics of language models and does not involve human subjects, private data, or deceptive practices. All experiments use a publicly available base model.

   Guidelines:

   - The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.

- If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
- The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

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

    Answer: [N/A]

    Justification: This paper is foundational research on training stability in self-play RL. It identifies failure modes and proposes a diagnostic framework rather than a deployable system. There is no direct path to negative applications beyond the general risks associated with improving language model training.

    Guidelines:

    - The answer [N/A] means that there is no societal impact of the work performed.
    - If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
    - Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
    - The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
    - The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
    - If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

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

    Answer: [N/A]

    Justification: The paper does not release models, datasets, or tools with high misuse risk. The trained models are small (4B parameters) and task-specific (code output prediction and DSL evaluation); they are not general-purpose assistants.

    Guidelines:

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

12. **Licenses for existing assets**

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

    Answer: [Yes]

    Justification: The base model (Qwen3-4B-Base) is cited. The AZR framework is cited Zhao et al. [2025]. Evaluation benchmarks (CRUXEval, LiveCodeBench) are referenced in the text.

    Guidelines:

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

13. **New assets**

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

    Answer: [Yes]

    Justification: The public code repository provides the DSL interpreter, task seed data, training configurations, and evaluation scripts, with installation, data-preparation, and reproduction instructions in its README. Appendix B specifies the DSL, and the repository includes a license file.

    Guidelines:

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

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

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

    Answer: [N/A]

    Justification: The paper does not involve crowdsourcing or human subjects.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.

- Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
- According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

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

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

    Answer: [N/A]

    Justification: The paper does not involve human subjects research.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
    - We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
    - For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

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

    Answer: [Yes]

    Justification: The core method trains an LLM (Qwen3-4B-Base) via self-play RL, where the LLM serves as both proposer and solver. This is described in Section 2.

    Guidelines:

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