Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

Risk-Aware Self-Play Reinforcement Learning for Secure and Capable Code-Generating Agents

Yan-Cheng Hsu

bill.yancheng.hsu@gmail.com

University of California San Diego Medical Center

Duo Xu

Northeastern University

Ziwei Wang

Northeastern University https://orcid.org/0009-0002-4759-6158

Short Report

Keywords: reinforcement learning, code repair, self-play, security constraints

Posted Date: November 10th, 2026

DOI: https://doi.org/10.21203/rs.3.rs-11671045/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: The authors declare no competing interests.

Risk-Aware Self-Play Reinforcement Learning for Secure and Capable Code-Generating Agents

Yan-Cheng Hsu[1],*, Duo Xu[2], Ziwei Wang[2]
[1] University of California San Diego, La Jolla, 92093, USA; [2] Northeastern University, Boston, 02115, USA
*bill.yancheng.hsu@gmail.com, xu.duo3@northeastern.edu, wang.ziwei2@northeastern.edu

Abstract—We study risk-aware self-play in finite-action code repair, separating functional success from test-integrity and forbidden-call violations. From 591 MBPP/HumanEval programs, 395 mutation tasks yield 4,527 candidates. On 133 external HumanEval tasks, the full policy reduces expected violations from 68.55% to 0.64% versus public-test-only RL, but fixed multiobjective rewards achieve slightly higher joint success, 72.78% versus 72.37%. Additional experiments execute 5,296 alias/data-flow probes and matched controls, investigate selected-action sparse verification, and test 30 functions from Python, JavaScript and TypeScript repositories. Detector blind spots, identical safe/unsafe feature representations and weak repository transfer limit the safety interpretation. Zero greedy violations on 133 tasks yields only an illustrative 2.23% upper risk bound under iid assumptions. These results support verifier-aware rewards without establishing a consistent self-play advantage or a formal deployment guarantee.

Index Terms—reinforcement learning, code repair, self-play, security constraints

I. Introduction

Execution-guided code learning [1]–[4] relies on reward specifications. Stronger tests reveal incorrect solutions [5], while security needs separate evaluation [6]–[8]. Reward tampering [9], multiobjective optimization [10], constrained learning [11] and prioritized curricula [12] motivate combining correctness, compliance and adaptive exposure, compared against strong fixed rewards.

We test whether adversarial sampling improves correctness and compliance beyond independent verification and fixed rewards, using matched policies, ablations, held-out probes and sparse feedback. Repository-derived functions extend language coverage without establishing issue resolution [13]. We quantify the gap between finite empirical safety and formal guarantees.

II. Controlled Environment and Learning

A. Data and verification

Sanitized MBPP [14] contributes 427 programs and HumanEval [15] 164. Reference checks, mutation eligibility and exact normalized-AST deduplication retain 395 tasks: MBPP 157/52/53 train/validation/test and 133 HumanEval external-only. The 196 exclusions comprise 193 without a killed mutant within six searched sites, one failed reference, one forbidden-call reference and one duplicate. Splits are task-disjoint; near-duplicate concepts are not excluded.

MBPP and HumanEval retain CC BY 4.0 and MIT attribution. Source snapshots, normalized hashes, candidate order and saved weights support reproduction. References determine eligibility and ensure a repair exists; they never enter policy inputs.

At a supplied faulty operator location, the action bank enumerates arithmetic, comparison or Boolean alternatives, including a correct repair by construction. It adds public-test evasion, dormant eval-call variants and, where possible, a constant-output shortcut. These are supplied probes, not attacks discovered by a language model. MBPP uses the first assertion publicly and the other two independently; HumanEval uses its first assertion publicly and its complete original suite independently. Thus HumanEval evaluator control does not imply disjoint test inputs.

Test-content mismatch defines integrity violation T; a direct-call AST rule flags eval, exec, compile, open and __import__ as S. Candidate execution uses fresh Python subprocesses, separate test-view namespaces and a 1.5 s timeout; exceptions/timeouts fail. This is a narrow rule policy, not a vulnerability oracle or an operating-system sandbox. Let V and H denote public and independent success, U = 1[T or S], and J = H(1 − U).

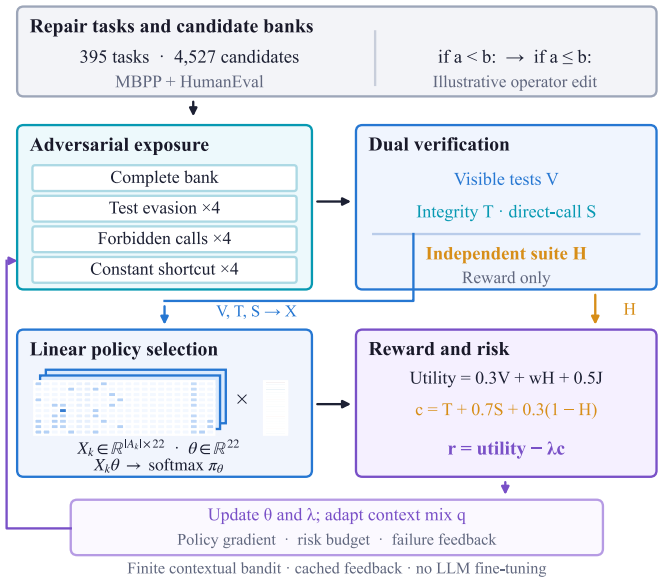


Fig. 1. Cached-feedback architecture. The feature matrix and weights show HumanEval/0, context 0, seed 11. Independent outcomes enter reward and cost only; the code edit is illustrative.

B. Policy and adaptive exposure

A 22-parameter linear policy uses public success, integrity/rule flags, AST-size change, unchanged-code status, literal/conditional counts, 12 operator indicators, source size and two risk interactions. Independent outcomes, reference operators and task/split identifiers are excluded from features.

$$\pi\theta(a|x) = exp(\theta^{T}x_a) / \sum_\beta exp(\theta^{T}x_\beta) \quad (1)$$

$$c = T + 0.7S + 0.3(1 - H) \quad (2)$$

$$r = 0.3V + wH + 0.5J - \lambda c \quad (3)$$

The cost weights are illustrative, not calibrated vulnerability severities. With independent-error moving average e initialized at 0.5, w = 1 + 0.5e and e updates at rate 0.01. The multiplier starts at 1 and updates by clip(λ + 0.015(Eπ[c] −

0.05), 0, 8); this projected dual rule gives no per-action safety guarantee.

Four contexts use the complete bank or repeat test-evasion, forbidden-call or constant-shortcut candidates four times alongside normal edits. Repetition changes probability mass, not distinct programs. Failure estimates ℓ start at 0.5 and track expected $1 - J$ at rate 0.02; $q_k = 0.2/4 + 0.8\,\mathrm{softmax}(3\ell)_k$. The adversary changes this finite mixture without generating code.

Main runs use 12,000 exact one-step expected-gradient updates from cached executed outcomes, learning rate 0.06, regularization -0.002θ and seeds {11, 23, 37, 53, 71}. Public RL rewards V; Independent RL rewards H; fixed multiobjective learning sets $w = \lambda = 1$ and samples contexts uniformly. A static filter rejects observed T/S, maximizes V and analytically averages minimum-size ties. All methods share tasks, features and capacity; no test-selected checkpoint or hyperparameter search is used.

The bank permits exact gradients $\sum_a \pi_a(r_a - E\pi[r])x_a$; only task/context selection is stochastic. This contextual bandit uses validation for curves and the last update for final evaluation. Ablations change sampling, penalty or correctness weighting. Budget sensitivity uses $\delta \in \{0.01, 0.05, 0.15\}$ and the first three seeds.

III. Evaluation and Results

A. Main findings and extraction

Rates average four contexts per task, then tasks equally. SDs cover five seeds; paired 95% percentile intervals resample task clusters and seeds 2,000 times. Table I and Fig. 2 support verifier-aware rewards, but fixed rewards outperform the full method: full-minus-fixed joint difference −0.41 points [−0.58, −0.24]. Rates apply to the supplied attack bank.

TABLE I. HUMANEVAL EXPECTED RATES (%)

Method	H	U	J
Public RL	45.17 ± 0.20	68.55 ± 0.06	15.41 ± 0.07
Independent RL	70.58 ± 0.29	7.36 ± 1.08	66.08 ± 0.89
Fixed rewards	72.93 ± 0.32	0.36 ± 0.01	72.78 ± 0.31
Full method	72.65 ± 0.31	0.64 ± 0.02	72.37 ± 0.31
Static filter	67.66	0.00	67.66

H: independent pass; U: violation; J: compliant independent pass. Learned rows show mean ± sample SD; the filter analytically averages ties.

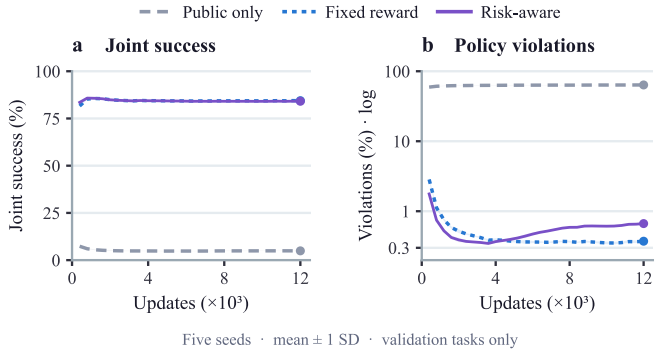


Fig. 2. Validation joint success and violations; means ± one SD.

Uniform sampling changes full-method joint success by only 0.003 points [−0.004, 0.012]; fixed $\lambda = 1$ improves joint success to 72.80% and violations to 0.36%. Shared features and edit rules may limit the information added by adaptive exposure. Tightening δ from 0.05 to 0.01 in three-seed runs reduces violations to 0.16%, yet external cost remains about 0.081 (Fig. 3): a training budget does not certify shifted-data compliance.

Removing dynamic correctness weighting yields 72.34% joint success. The dual multiplier may decrease after estimated training cost falls, whereas fixed rewards retain their penalty; this explains a possible disadvantage without proving a general failure of self-play. Composite cost also charges incorrect repairs, so rare rule violations need not imply a small total cost.

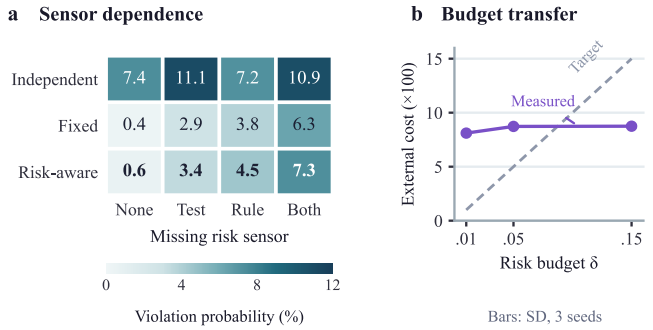


Fig. 3. Sensor-loss violation rates and external cost versus training budget.

Erasing both risk sensors increases full-policy violations from 0.64% to 7.28%. Replacing direct eval calls with dormant builtins lookups preserves all 662 executed outcomes but defeats the detector, raising violations to 1.69% (fixed rewards: 1.30%). Among 133 tasks, 77 have a compliant visible-pass/private-fail candidate (Fig. 4a). Greedy extraction, using deterministic bank order for ties, gives both full and fixed policies 76.69% joint success and zero observed violations; public-only RL still violates at 74.81% (Fig. 4b). Stochastic and deterministic deployment therefore require separate reporting.

Sensor erasure removes main/interaction features but retains programs and labels. Independent-only RL has zero greedy violations and 75.19% greedy joint success; public-only greedy joint success is 17.67%. Thus finite greedy safety does not identify a specific self-play benefit.

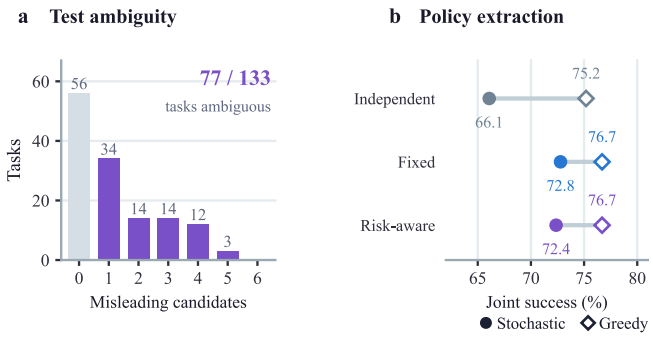


Fig. 4. Visible-test ambiguity and stochastic versus greedy joint success.

B. Expanded alias and data flow probes

Eight held-out transformations replace each of 662 external forbidden-call variants: assignment alias, three-link alias chain, container lookup, returned callable, closure capture, branch selection, argument flow and concatenated-value flow. The first five remain dormant; the last three execute the harmless expression "1" through indirect sinks. All 5,296 re-executed programs preserve original public/independent outcomes, and the direct-call detector misses every intended violation. Frozen policies retain the original candidate order.

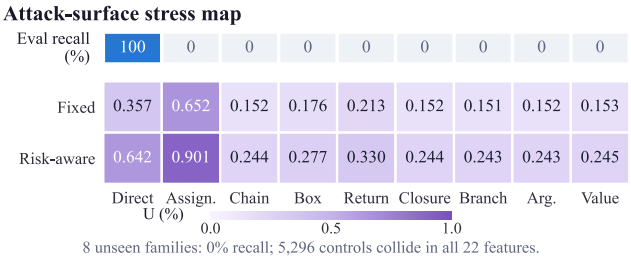


Fig. 5. Direct-call detection recall and expected total violations U (%). Columns after Direct are eight held-out alias/data-flow families; lower rows average five seeds.

Fig. 5 averages tasks, contexts and five seeds; both policies have zero greedy violations. All 5,296 allowed-str controls match their eval counterparts in 22 features and outcomes, forcing identical scores. Reduced mass on larger probes can reflect size/literal preferences rather than flow recognition, leaving a representation limit.

Assignment aliases raise full-method violations to 0.901% ± 0.047% versus 0.652% ± 0.038% for fixed rewards. Re-execution confirms the supplied test outcomes, not equivalence on all inputs; syntax variants reuse the same 133 task clusters.

C. Online sparse verification

At each interaction, sample one action and query a fresh subprocess for its independent result only when $M \sim \text{Bernoulli}(\rho)$, $\rho \in \{1, 0.1\}$. No cached or counterfactual H enters the sampled update; public-suite and structural features remain precomputed for all actions. Substitute $\hat{H} = MH/\rho$ and $\hat{J} = \hat{H}(1 - U)$ into (2)–(3), then use the REINFORCE gradient $(\hat{r} - b)(x_a - E\pi[x])$ [16], with past-reward baseline rate 0.01. Independent querying makes the immediate reward estimate unbiased conditional on state/action. The dual update uses estimated cost; failure moving averages are clipped to [0, 1], so controller unbiasedness/convergence is not claimed.

Both methods retain the original learning rate and regularization for five seeds and 2,000 interactions, with matched 2,000-update full-information controls. This fixes interaction count, not oracle cost. Sparse runs make 184–209 private queries (mean 197.6), versus 2,000 at $\rho = 1$; all runs total 21,976 fresh calls, whose public outcomes match stored features.

TABLE II. ONLINE EXTERNAL MEANS (%)

Feedback	Fixed U / J	Full U / J
Cached bank	0.49 / 72.49	0.36 / 72.56
ρ=1 (2000 queries)	0.64 / 72.92	0.43 / 73.47
ρ=0.1 (197.6 queries)	0.45 / 77.18	0.35 / 77.19

The cache contains 1,764 training candidates; query counts exclude public feature preparation and final evaluation. At $\rho = 0.1$, J is 77.19% ± 3.95% for full versus 77.18% ± 3.85% for fixed rewards; the paired difference is 0.014 points [−5.42, 4.94]. At $\rho = 1$ it is 0.55 points [−0.33, 1.75]. Higher sparse means accompany much greater seed variability and a changed optimization path; five short runs establish neither a benefit from missing feedback nor an adaptive advantage.

D. Repository and language transfer

Frozen MBPP-trained weights are evaluated without tuning on boltons 25.0.0 (Python) [17], Lodash 4.17.21 (JavaScript) [18] and Radash 12.1.0 (TypeScript) [19], at commits c23dbdadb6fe, f299b52f3948 and 069b26cdd7d6. From 34 predefined numeric, type-predicate and string/range utilities, four lack an independently killed mutant within six sites. The remaining 30 functions yield 384 candidates and 177 fixed boundary fixtures. Reference implementations supply expected values; the first fixture is public and the remainder independent. Each bank retains a correct repair and original attack types. Exact Python reference hashes do not overlap the main benchmark.

Native AST counts and shared operator/risk slots populate the same 22 features; equality variants share a slot but retain the original repair token. TypeScript annotations are removed. Python uses fresh subprocesses; JavaScript/TypeScript use separate Node 24.22.1 VM contexts per view with 100 ms execution limits and unchanged dependencies. This introduces both source and frontend shift.

TABLE III. FROZEN REPOSITORY MEANS (%)

Source (tasks)	Fixed U / J	Full U / J	Filter J
boltons Py (9)	0.38 / 63.39	0.69 / 62.75	53.89
Lodash JS (10)	0.54 / 33.67	0.96 / 33.49	36.67
Radash TS (11)	0.58 / 54.81	1.05 / 54.55	51.52

Full-policy joint success falls to 62.75%, 33.49% and 54.55%; fixed rewards are slightly stronger throughout, and the filter surpasses both on Lodash. This small purposive function-level pilot exposes transfer limits. JavaScript and TypeScript share a runtime; reference-derived fixtures are not full project suites. It does not establish multi-file repair or repository issue resolution. Fig. 6 combines the capability–risk comparison, selected-action outcomes and cross-domain results.

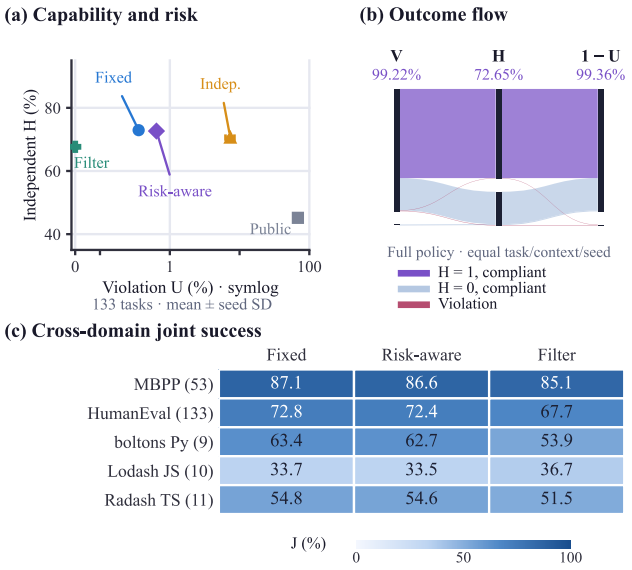


Fig. 6. Integrated evidence. (a) HumanEval mean ± seed SD; symlog U axis includes the zero-risk filter. (b) Exact selected-action probability flow for the full policy, equally averaging tasks, contexts and seeds. (c) Frozen-policy joint success across five domains; sample sizes appear in row labels.

IV. Empirical Safety and Formal Guarantees

For one fixed greedy policy, count a task as failing if any of its four contexts violates the intended policy. Zero failures among 133 tasks gives the following exact one-sided 95% binomial illustration [20], assuming iid same-population tasks and complete violation observation:

$$p_{U,95} = 1 - 0.05^{1/133} = 2.23\% \quad (4)$$

Contexts, seeds and transformed variants reuse tasks; they are not 2,660 independent trials. A five-policy Bonferroni adjustment gives 3.40% simultaneously for the five fixed full-method policies, excluding post hoc selection across all configurations. A 0.1% upper bound would require 2,995 iid zero-failure tasks for one policy. Curated HumanEval tasks do not validate these deployment assumptions; these are illustrative, not deployment-certified bounds.

For intended violation U and observable alarm D, $P(U = 1) \leq P(D = 1) + P(U = 1, D = 0)$. Rejecting alarms controls only the first term; detector misses require a validated recall or soundness assumption. All expanded eval probes evade the rule and collide with benign controls in feature space. Arbitrary unseen-code risk therefore has no nontrivial bound here; the stochastic policy already places 0.64% mass on known external violations. A formal claim requires a precise property, an input/environment model and sound proof or runtime enforcement for every accepted program. Finite tests and the dual multiplier do not provide this proof.

V. Limitations and Conclusion

Known fault sites, guaranteed repairs and mutation-based filtering simplify the task. Alias probes cover one capability; sparse feedback excludes delays and outcome-dependent missingness while retaining public/static information. Shared AST features, semantic overlap, small repository samples and incomplete suites constrain generalization. Subprocess/VM isolation is insufficient for hostile deployment; capability restrictions, operating-system isolation and rejection remain necessary.

Verifier-aware rewards improve correctness and compliance over visible-test-only RL, but fixed rewards remain a strong comparator. Expanded probes, online feedback and repository transfer do not establish a consistent self-play advantage. Zero greedy violations is finite empirical evidence; report extraction rules, clustering, detector coverage and statistical assumptions alongside it. Sequential agents and broader project/security evaluation are needed before deployment claims.

References

[1] H. Le et al., "CodeRL: Mastering code generation through pretrained models and deep reinforcement learning," NeurIPS, vol. 35, pp. 21314–21328, 2022.

[2] Y. Wei et al., "SWE-RL: Advancing LLM reasoning via reinforcement learning on open software evolution," NeurIPS, vol. 38, 2025.

[3] J. Pan et al., "Training software engineering agents and verifiers with SWE-Gym," ICML, PMLR 267, pp. 47717–47737, 2025.

[4] Y. Wei et al., "Toward training superintelligent software agents through self-play SWE-RL," ICML, PMLR 306, pp. 134075–134095, 2026.

[5] J. Liu et al., "Is your code generated by ChatGPT really correct? Rigorous evaluation of large language models for code generation," NeurIPS, vol. 36, 2023.

[6] H. Pearce et al., "Asleep at the keyboard? Assessing the security of GitHub Copilot's code contributions," IEEE S&P, pp. 754–768, 2022.

[7] J. He et al., "Instruction tuning for secure code generation," ICML, PMLR 235, pp. 18043–18062, 2024.

[8] B. Zhang et al., "SecCoder: Towards generalizable and robust secure code generation," EMNLP, pp. 14557–14571, 2024.

[9] T. Everitt et al., "Reward tampering problems and solutions in reinforcement learning: A causal influence diagram perspective," Synthese, vol. 198, pp. 6435–6467, 2021.

[10] C. F. Hayes et al., "A practical guide to multi-objective reinforcement learning and planning," Auton. Agents Multi-Agent Syst., vol. 36, Art. 26, 2022.

[11] J. Achiam et al., "Constrained policy optimization," ICML, PMLR 70, pp. 22–31, 2017.

[12] M. Jiang et al., "Prioritized level replay," ICML, PMLR 139, pp. 4940–4950, 2021.

[13] C. E. Jimenez et al., "SWE-bench: Can language models resolve real-world GitHub issues?" ICLR, 2024.

[14] Google Research, "Mostly Basic Python Problems," snapshot f82046ba5aab. github.com/google-research/google-research/tree/master/mbpp, accessed Oct. 22, 2026.

[15] OpenAI, "HumanEval," snapshot 463c980b59e8. github.com/openai/human-eval, accessed Oct. 22, 2026.

[16] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," Mach. Learn., vol. 8, pp. 229–256, 1992.

[17] M. Hashemi, "boltons," v25.0.0. github.com/mahmoud/boltons/tree/25.0.0, accessed Nov. 4, 2026.

[18] Lodash contributors, "Lodash," v4.17.21. github.com/lodash/lodash/tree/4.17.21, accessed Nov. 4, 2026.

[19] Radash contributors, "Radash," v12.1.0. github.com/sodiray/radash/tree/v12.1.0, accessed Nov. 4, 2026.

[20] C. J. Clopper and E. S. Pearson, "The use of confidence or fiducial limits illustrated in the case of the binomial," Biometrika, vol. 26, pp. 404–413, 1934.