ANCHORED SELF-PLAY FOR CODE REPAIR

Caroline Choi, Zeyneb N. Kaya, Shirley Wu, Tengyu Ma, Tatsunori Hashimoto, Ludwig Schmidt
Stanford University

ABSTRACT

Code repair is an important capability for language models (LMs): given a buggy program and unit tests, an LM must produce a fixed program that passes the tests. Because curated code-repair data is limited, we study whether supervision can be scaled by having an LM generate bug–fix tasks with unconstrained edits, using unit tests for verification. We propose *generator–fixer self-play*, which trains a single model with reinforcement learning to alternate between generating bugs and fixing them. As the fixer improves, the generator adapts to produce more challenging bugs, yielding an automatic curriculum. However, because unit tests certify correctness but not realism, it is unclear whether training on these synthetically-generated bugs improves repair on bugs encountered in practice. To measure this realism gap, we introduce BUGSOURCEBENCH, which evaluates repair across diverse bug sources: human-authored bugs, errors in LM-generated code, and human edits of buggy LM-generated code. On BUGSOURCEBENCH, we find that generator–fixer self-play improves repair on its self-generated bugs while degrading on human-authored bugs. We propose ANCHORED SELF-PLAY (ASP), which anchors self-play to a small reference set drawn from the target bug sources. ASP (i) shapes generation with a code-embedding similarity reward and (ii) mixes reference bugs into fixer training to stabilize learning as the generator evolves. Across sources, ASP achieves the best fix rates, improving the average fix rate by $+25\%$ (relative) / $+7.2$ pp (absolute) over standard self-play, with gains on both LM-originated bugs ($+100\%$ (relative) / $+11$ pp (absolute)) and human-authored bugs ($+7.1\%$ (relative) / $+3.4$ pp (absolute)).

1 INTRODUCTION

As language models (LMs) are increasingly used for programming, reliable code repair is an important capability (Xu et al., 2022; Jimenez et al., 2023). In code repair, a model is given a buggy program and unit tests, and must produce a corrected program that passes the tests. However, obtaining large collections of realistic buggy programs is costly and hard to scale (Just et al., 2014; Widyasari et al., 2020; Le Goues et al., 2015; Madeiral et al., 2019; Oliva et al., 2025).

We ask whether supervision for code repair can be scaled synthetically: an LM proposes a bug–fix task by editing a correct program, and unit test outcomes filter the generated task. Crucially, we study an *open-ended* generation setting in which the bug generator can apply arbitrary text edits, rather than pre-defined mutations or repository-history rewrites, so synthetic training data is not limited to a smaller set of tasks.

We operationalize this with *generator–fixer self-play* (Figure 1). A single model is trained with reinforcement learning to alternate between (i) generating a bug via unconstrained text edits and (ii) fixing the bug. The generator is rewarded for producing valid, appropriately difficult bugs (tests fail), and the fixer is rewarded for producing correct repairs (tests pass). As the fixer improves, the generator must generate harder bugs, yielding an automatic curriculum.

A central question is whether this synthetic curriculum improves repair on bugs that appear in practice. In modern programming workflows, bugs arise from heterogeneous sources: (i) human-authored code, (ii) naturally occurring errors in LM-generated drafts, and (iii) hybrid failures introduced as developers adapt and edit model outputs during iterative debugging and integration (Cui et al., 2024). To evaluate cross-source generalization, we introduce BUGSOURCEBENCH, which holds the underlying prompt and unit tests fixed and varies only the buggy program source: BUG-

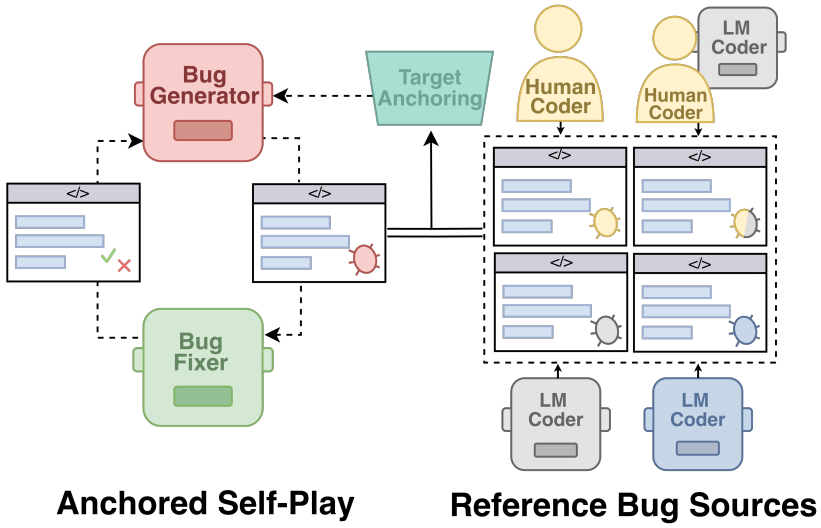


Figure 1: **Anchored self-play for code repair.** *Left:* In generator–fixer self-play, the generator edits a correct program to produce a bug and the fixer repairs it; unit tests reward bug validity (tests fail) and repair correctness (tests pass). Because unit tests certify correctness but not realism, self-play can drift toward unrealistic yet test-failing bugs. *Right:* Our benchmark reflects this realism gap by evaluating the same tasks under multiple bug sources—human-written bugs, human edits of buggy model drafts, and naturally occurring model errors (failed attempts at solving the task). ANCHORED SELF-PLAY (ASP) mitigates drift by anchoring training to a small reference set from these sources via (i) an embedding-similarity reward for generation and (ii) mixing reference bugs into fixer training.

SOURCEBENCH-HUMAN (human-authored bugs), BUGSOURCEBENCH-HUMAN-EDITED LM (human edits of buggy LM drafts), and two LM-origin splits where buggy programs are produced directly by QWEN-7B and GPT-OSS-20B.

Using BUGSOURCEBENCH, we find a key failure mode of self-play. Because unit tests verify correctness but not realism, the generator can drift toward edits that are hard for the fixer yet unrepresentative of real bugs, improving on self-generated bugs while degrading on human-authored bugs. Self-play may over-optimize for "hard bugs that break tests" rather than bugs that resemble those encountered in practice.

We propose ANCHORED SELF-PLAY (ASP), which anchors self-play to a small reference set sampled from the target bug sources. ASP (i) adds an embedding-similarity reward that favors generator outputs resembling reference bugs and (ii) mixes reference bugs into fixer training to prevent overfitting to the generator's evolving distribution. Across bug sources in BUGSOURCEBENCH, ASP achieves the best fix rates, improving the average fix rate by $+25\%$ (relative) / $+7.2$ pp (absolute) over standard self-play, with gains on both LM-error bugs (100% relative / $+11$ pp absolute) and human-authored bugs (7.1% relative / $+3.4$ pp absolute).

Our key contributions are:

- We study open-ended generator–fixer self-play for code repair, where bugs are generated via unconstrained edits and filtered only by unit tests.
- We introduce BUGSOURCEBENCH, a repair benchmark that measures generalization across human bugs, naturally occurring LM errors, and human edits of buggy LM-generated code.
- We propose ANCHORED SELF-PLAY (ASP), which anchors self-play with an embedding-similarity reward and reference-mixed fixer training, improving fix rates across bug sources and reversing regressions on human-authored bugs.

2 PROBLEM FORMULATION

Our goal is to scale supervision for training code-repair models using unit tests as the sole correctness signal. We formalize the code-repair setting and our evaluation across realistic bug sources.

Code repair. Each task x consists of natural-language programming instructions (with input/output specifications and constraints) together with unit tests that check correctness. Given any program c, executing the tests produces (i) a binary verifier $v(x, c) \in \{0, 1\}$, where $v(x, c) = 1$ if and only if c passes all tests for x, and (ii) unit test output $\mathrm{o}(x, c)$ (e.g., compile errors, failed tests, and stack traces).

A *bug* for task x is a program b that (i) compiles and (ii) fails at least one test: $v(x, b) = 0$. A repair model π_θ (the *fixer*) induces a conditional distribution over candidate repairs given the task, buggy program, and unit test output:

$$y \sim \pi_\theta(\cdot \mid x, b, \mathrm{o}(x, b)).$$

A repair y succeeds on (x, b) if it passes all tests, i.e., $v(x, y) = 1$. In our pipeline, the fixer outputs a full corrected program, rather than a diff, because producing well-formed diffs is challenging for the `Qwen2.5-Coder-7B-Instruct` model used in our experiments.

Evaluation across bug sources. In practice, bugs arise from heterogeneous sources, such as human programmers, LM coding assistants, or a hybrid of the two. For controlled comparisons, we evaluate on a fixed set of tasks x, and construct each split by sampling bugs from a different source s. Formally, we model each source $s \in \mathcal{S}$ as a conditional distribution over bugs for the same tasks, $b \sim P_s(\cdot \mid x)$. For a bug source s, the repair rate of a fixer π_θ is

$$\mathrm{Perf}(\pi_\theta; s) = \mathbb{E}_x \, \mathbb{E}_{b \sim P_s(\cdot|x)} \, \mathbb{E}_{y \sim \pi_\theta(\cdot|x,b,\mathrm{o}(x,b))} \left[v(x, y) \right].$$

Our goal is to maximize the average repair rate across bug sources,

$$\mathrm{Perf}_{\mathrm{avg}}(\pi_\theta) = \frac{1}{|\mathcal{S}|} \sum_{s \in \mathcal{S}} \mathrm{Perf}(\pi_\theta; s).$$

With this evaluation criterion in place, we next describe BUGSOURCEBENCH, which instantiates the bug source set $\mathcal{S}$ with realistic bug sources that arise in LM-assisted programming.

3 BUGSOURCEBENCH: CONTROLLED BUG-SOURCE EVALUATION

3.1 BENCHMARK CONSTRUCTION

We build BUGSOURCEBENCH from BigCodeBench code-generation tasks (Zhuo et al., 2024), which emphasize realistic library and API usage. Each task provides natural-language instructions and unit tests that define correctness. We convert each task into a repair instance by pairing the same instructions and tests with a buggy implementation, yielding triples (x, b, v) where $v(x, \cdot)$ is the unit-test verifier.

Task structure. All buggy programs in BUGSOURCEBENCH are *executable* (run/compile) but fail at least one unit test, focusing evaluation on semantic repair rather than syntax fixing. At test time, the model receives (x, b) along with unit-test feedback $\mathrm{o}(x, b)$ (e.g., failing tests and truncated traces) and must output a corrected program y that passes all tests.

Bug sources. BUGSOURCEBENCH is designed to measure cross-source generalization *on a fixed set of tasks*. All splits share the same tasks x (problem statement and unit tests); only the buggy program b varies by source. We include four bug-source variants that reflect common LM-assisted programming workflows: (i) HUMAN, where annotators introduce localized, realistic bugs into the reference solution; (ii) HUMAN-EDITED LM, where annotators edit an incorrect LM draft produced by `gpt-5-mini` while keeping it executable and still incorrect; (iii) LM ERRORS (QWEN-7B), incorrect solutions generated by `Qwen2.5-Coder-7B-Instruct`; and (iv) LM ERRORS (GPT-OSS-20B), incorrect solutions generated by `gpt-oss-20b`. We use a test-trace repair interface in all main experiments; alternative interfaces and full construction details are provided in Appendix A.2.

We analyze bugs from different sources in Appendix A.4. We categorize bugs into coarse error types and find systematic, source-dependent failure patterns. We then embed bugs with `voyage-code-3` and measure kNN source purity (excluding same-task neighbors), revealing strong within-source clustering.

With BUGSOURCEBENCH, we can test whether training on synthetic bug–fix data improves repair on the realistic bug sources in Section 2. We next present two training approaches: generator–fixer

self-play with a correctness-only reward, and ASP, which anchors self-play to a small reference set to better match target bug sources.

4 SELF-PLAY FOR CODE REPAIR

We aim to scale supervision for code repair when unit tests are the only source of verification. To do so, we propose *generator–fixer self-play*: a single policy π_θ is trained to synthesize and repair buggy programs of increasing difficulty (Figure 1).

4.1 GENERATOR–FIXER SELF-PLAY

For each task x (code generation prompt, tests, and a reference solution), a candidate buggy program is sampled from the generator: $b \sim \pi_\theta(\cdot \mid x)$. Unit tests are run to obtain test output $\mathrm{o}(x, b)$. One or more candidate repairs are sampled from the fixer, conditioned on the task, buggy program, and test output, $y \sim \pi_\theta(\cdot \mid x, b, \mathrm{o}(x, b))$. We say b is *valid* if it compiles and fails at least one unit test; invalid bugs receive a penalty and are not passed to the fixer.

4.2 CORRECTNESS AND DIFFICULTY SHAPING

Fixer reward. The fixer is rewarded for producing correct repairs. Concretely, for a repair y we use

$$r^{\mathrm{F}}(x, b, y) = v(x, y),$$

which is 1 if and only if y passes all tests and 0 otherwise.

Generator reward. As the fixer improves, the generator should produce progressively harder but still solvable bugs to continue improving the fixer's repair capability. A single repair attempt is noisy, so we estimate bug difficulty using K independent repair attempts. For a bug (x, b), we sample K repairs $y^{(1)}, \ldots, y^{(K)} \sim \pi_\theta(\cdot \mid x, b, \mathrm{o}(x, b))$ and define the fix rate

$$\rho(x, b) = \frac{1}{K} \sum_{k=1}^{K} v(x, y^{(k)}).$$

However, the generator can collapse to invalid bugs (e.g. compile or syntax errors) or bugs that are unsolvable by the current fixer, providing little training signal. We shape generator rewards using $\rho(x, b)$, rewarding valid bugs that fall in a moderate difficulty band:

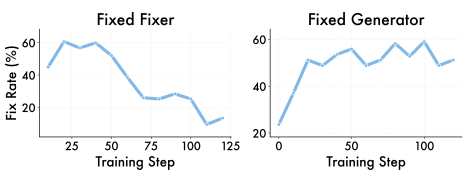
$$r^{G}_{\mathrm{base}}(x, b) = \begin{cases} -1, & b \text{ does not compile or passes all tests,} \\ 1, & \rho(x, b) \in [\rho_\ell, \rho_h], \\ -\alpha, & \rho(x, b) \in \{0, 1\}, \\ 0, & \text{otherwise.} \end{cases}$$

4.3 OPTIMIZATION

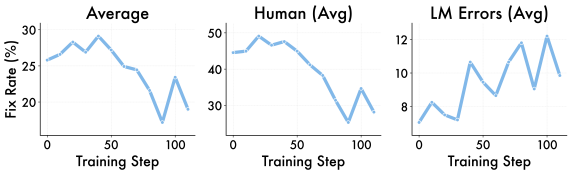
We optimize both the generator and fixer with Group Relative Policy Optimization (GRPO). For each task x, we sample G candidate bugs from the generator and compute rewards for each sample; we then perform a clipped policy-gradient update using *group-normalized* advantages computed across the G bugs (normalized within the *generator role*). For the fixer, we reuse the same sampled bugs and sample K repair attempts per bug; we compute rewards per repair and update the fixer with a clipped GRPO objective using advantages normalized *within each bug* across its K repairs (normalized within the *fixer role*). Full objectives and normalization details are provided in Appendix B.1.

4.4 DISTRIBUTION DRIFT UNDER SELF-PLAY

The base rewards above ensure bugs are valid and of appropriate difficulty, but they do not encourage *realistic* bug patterns. In Figure 2b, we observe that standard self-play shows small gains initially but later degrades on Human-sourced bugs despite the fixer improving on its self-generated bugs in Figure 2a. We thus modify the training objectives in self-play to anchor bug generation to reference bug sources.



(a) **Co-evolution of standard self-play.** With a fixed fixer checkpoint (step 40), fix rate declines over generator training, indicating the generator produces harder bugs. With a fixed generator checkpoint (step 40), fix rate on the generated bugs increases as the fixer trains.



(b) **Standard self-play exhibits distribution drift.** Fix rate improves early but later regresses, most strongly on human bug sources. This suggests that the synthetic bugs generated during self-play are not representative of bugs encountered in practice.

4.5 Anchoring self-play to reference bug sources

We assume access to a small dataset of reference-source bugs, $\mathcal{D}_{\text{ref}}$, drawn from the training tasks and disjoint from evaluation. We use $\mathcal{D}_{\text{ref}}$ in two ways: (i) *reference mixing*, which injects reference bugs into fixer training, and (ii) *similarity-guided shaping*, which nudges the generator toward reference-like bugs.

4.5.1 Reference mixing for fixer training

For tasks with an associated reference bug $b^{\text{ref}} \in \mathcal{D}_{\text{ref}}$, we replace the on-policy generated bug with probability p_{mix} and train the fixer on $(x, b^{\text{ref}}, \text{o}(x, b^{\text{ref}}))$. Intuitively, p_{mix} controls the strength of the anchor: larger values bias training toward the reference bug sources, while smaller values preserve more on-policy curriculum. On reference bugs, we do not update the generator.

4.5.2 Similarity-guided generator shaping

Reference mixing anchors the fixer but does not directly encourage the generator to produce reference-like bugs. We therefore add an auxiliary shaping term to the generator reward based on similarity to a small reference set $\mathcal{D}_{\text{ref}}$.

For each generated bug b, we compute a unified diff between the reference solution and b, embed the diff with a code embedding model, and score it by the average cosine similarity to its k nearest neighbors among reference edit embeddings. We map this score to $[0, 1]$ and denote it by $\text{sim}_{01}(b)$. For valid bugs, we augment the base generator reward with a centered similarity term weighted by λ:

$$r^{\text{G}}(x, b) = r^{\text{G}}_{\text{base}}(x, b) + \lambda\, \tilde{\delta}(b),$$

where $\tilde{\delta}(b)$ is a centered (optionally clipped) version of $\text{sim}_{01}(b)$ using an EMA baseline. Appendix B.2 gives the exact centering rule and discusses interactions with reference mixing.

5 Experimental Setup

Data and splits. We train on 900 BigCodeBench tasks (Zhuo et al., 2024). We evaluate on 127 held-out tasks shared across all BugSourceBench sources—only the buggy programs differ by source—and use 81 disjoint tasks (from all sources) for validation and checkpoint selection. We report results on these bug sources in BugSourceBench: Human, Human-Edited LM, Qwen-7B, and gpt-oss-20b.

Reference pool (anchoring data). ASP uses a reference pool of 900 bugs, sampled evenly from the training splits of the four BugSourceBench sources. We use this pool both as the embedding reference set for the similarity reward and for reference-mixed fixer training. For parity, Fixer-only uses the same reference-mixed bugs but omits similarity-based anchoring.

Initialization. Generator and fixer are initialized from `Qwen2.5-Coder-7B-Instruct` (Hui et al., 2024). Unless noted, they share weights and are trained with GRPO.

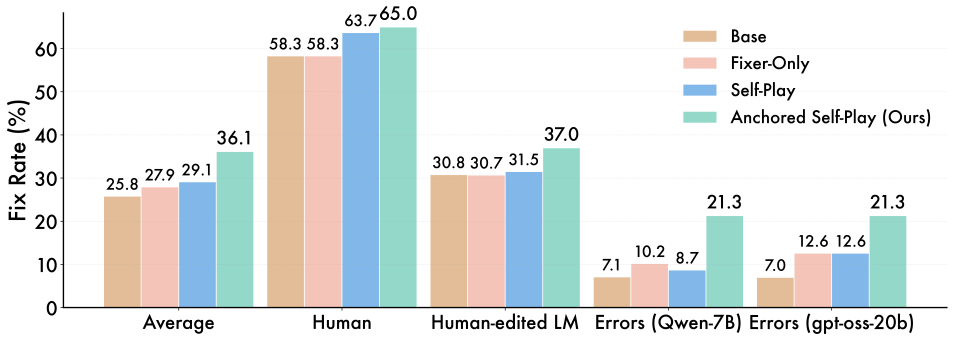


Figure 3: **Fix rates across bug sources on BUGSOURCEBENCH.** Standard self-play improves on LM-sourced bugs but regresses on human-authored bugs, consistent with distribution drift; anchoring (ASP) reverses this failure mode and achieves the best overall fix rate (+7.2 pp / +25% relative vs. self-play), with gains on both LM (+11 pp / +100% rel.) and human bugs (+3.4 pp / +7.1% rel.).

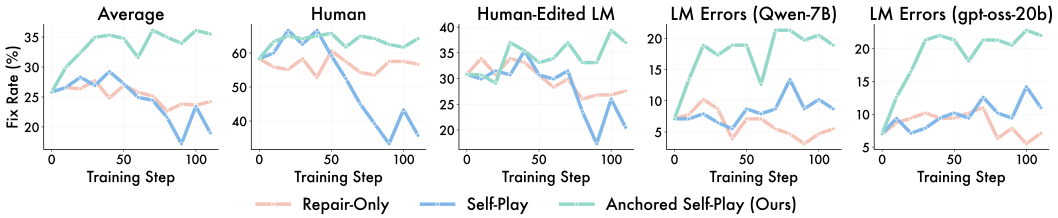


Figure 4: **ASP improves generalization across bug sources.** Fix rate (%) vs. training step for REPAIR-ONLY, vanilla SELF-PLAY, and ANCHORED SELF-PLAY (ours), evaluated on bugs from multiple sources – Human, Human-Edited LM, LM Errors (Qwen7B), LM Errors (gpt-oss-20b). Standard self-play shows early gains but later degrades on Human and Human-Edited LM, while ASP yields steady improvements across bug sources.

Comparisons. We compare ASP to the **base model** (pretrained fixer), **fixer-only** (frozen base generator; train only the fixer on its bugs, with reference mixing), and **self-play** (joint generator–fixer training with GRPO; shared weights). Additional baselines (code-generation training; supervised fine-tuning of the Fixer) are in Section D.3.

Training and evaluation protocol. The generator samples candidate buggy programs; we keep those that execute and fail at least one unit test. The fixer receives the task and buggy code (optionally a truncated summary of failing tests) and proposes up to K repairs per bug during training (default $K{=}4$), with unit-test outcomes as reward. We update with GRPO using role-normalized advantages. At test time, we use one repair attempt per bug with greedy decoding (temperature 0.0). Full prompts and hyperparameters are in Section B.

6 MAIN RESULTS

Figure 3 compares ASP, standard self-play, and fixer-only training. ASP performs best overall and on every bug source, improving average fix rate by +7.2 pp (+25% rel.) over self-play and +8.2 pp (+29% rel.) over fixer-only. Gains are largest on bugs from LM errors (+12.6 pp/+145% on Qwen-7B; +8.7 pp/+69% on gpt-oss-20b), while also improving on human-sourced bugs (+8.2 pp/+29.4% on Human-Edited; +1.3 pp/+2.0% on Human).

Self-play improves on average over fixer-only (+5.4 pp) but regresses on LM Errors (Qwen-7B) (−1.5 pp). Fixer-only matches the base model on average, with gains concentrated on LM Errors (Qwen-7B) – 10.2% vs. 7.1% – and LM Errors (gpt-oss-20b) – 12.6% vs. 7.0%) – with negligible change on Human (30.7% vs. 30.8%).

ASP produces more reference-like bugs. Figure 9 shows that the similarity reward shifts generation toward the reference pool: the mean kNN similarity of generated bugs increases over training. Moreover, ASP outperforms standard self-play within each similarity quantile, indicating that gains

Method	Ref.	Sim.	Fix Rate (%)
Base model			25.8
Self-play			29.1
+ Ref. mix	✓		29.5
+ Sim. reward		✓	30.9
ASP	✓	✓	**36.1**

Table 1: **Ablation of components in ASP.** "Ref. mix" mixes a small set of reference bugs into training, while "Sim. reward" adds an embedding-based code-similarity reward to guide the generator. Combining both yields the best fix rate. Results are averaged over all BUGSOURCEBENCH splits.

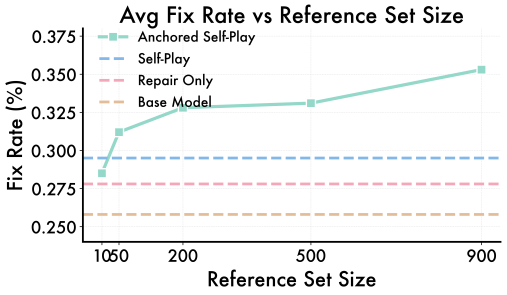


Figure 5: **Reference set scaling.** Fix rate of ASP vs. number of reference bugs used for anchoring (x-axis), averaged over all bug sources in BUGSOURCEBENCH.

Table 2: **Effect of reference pool composition.** We vary the reference set used for anchoring (human-only, LM-only, or mixed) and report fix rates on each BUGSOURCEBENCH split. Human-only yields the best overall performance and largest gains on human and human-edited bugs, while LM-only shifts improvements toward LM bug sources.

	Fix Rate (%)				
Reference Pool	**Overall**	**Human**	**Human-Edited LM**	**LM (Qwen-7B)**	**LM (gpt-oss-20b)**
Self-Play	29.1	63.7	31.5	8.7	12.6
Human-only	34.4	67.5	37.8	11.8	11.0
LM-only	32.0	65.4	41.7	13.4	17.3
ASP (Ours)	**36.1**	**65.0**	**37.0**	**21.3**	**21.3**

are not explained only by producing higher-similarity bugs. We include qualitative examples in Figure 8 (Section D).

7 ABLATIONS

7.1 ANCHORING COMPONENTS

Table 1 ablates the two components of ASP: (i) mixing a small set of reference bugs into training (*Ref. Mix*) and (ii) adding an embedding-based code-similarity reward for the generator (*Sim. Reward*). Starting from Self-Play, each component alone provides a small improvement in fix rate. Combining them yields larger improvements (+6.3%) because they address complementary failure modes in the generator–fixer loop. The similarity reward guides the generator toward reference-like bugs, while reference mixing stabilizes the fixer by providing exposure to realistic bugs as the generator evolves.

7.2 REFERENCE SET SOURCES

Table 2 varies the bug sources included in the reference set. Using only HUMAN and HUMAN-EDITED LM references yields the best performance on human bugs, suggesting that anchoring on human patterns preserves robustness on human sources. Using only QWEN-7B and GPT-OSS-20B references improves repair on LM-originated bugs but slightly degrades human performance. The reference set guides learning toward the bug patterns it contains. These results motivate using a reference set that contains several diverse bug sources.

7.3 REFERENCE SET SIZE

Figure 5 varies the reference set size, sampled uniformly across bug sources: Human, Human-Edited LM, Qwen-7B, and gpt-oss-20b. ASP improves with as few as 50 reference set examples, indicating

sample-efficient anchoring. Repair performance improves with larger reference sets, which provide broader coverage of reference bug patterns.

8 Related Work

Code repair. Program repair is commonly evaluated on curated real-world bug datasets from open-source projects and short, unit-testable repair benchmarks (Just et al., 2014; Widyasari et al., 2020; Le Goues et al., 2015; Madeiral et al., 2019; Lin et al., 2017); more recently, repository-level benchmarks emphasize end-to-end issue resolution with realistic context and tooling (Jimenez et al., 2023; Yang et al., 2025a; Pham et al., 2025). Across settings, performance is often sensitive to the underlying bug distribution and data-generating process, with substantial shifts between human-written bugs, synthetic mutations, and errors in model-generated code (He et al., 2022; Xu et al., 2022; Sonwane et al., 2025; Dou et al., 2025; Yang et al., 2025b). BugSourceBench complements prior benchmarks by spanning human, LM, and hybrid sources to enable controlled evaluation of cross-source generalization.

Synthetic bug generation, grounding, and self-play curricula. Several approaches scale supervision via synthetic bugs while constraining generation to reduce pathological drift, e.g., predefined mutation operators in BugLab (Allamanis et al., 2021; Forrest et al., 2009) or repository-grounded task construction using structure, tests, and patch provenance (Yang et al., 2025a; Pham et al., 2025; Wei et al., 2025; Ye et al., 2023; Zirak & Hemmati, 2024); related co-evolutionary setups such as Break-it-Fix-it (BIFI) bias bug introduction toward natural corruptions using critics (e.g., parsers/compilers) (Yasunaga & Liang, 2021; Long & Rinard, 2016; Chen et al., 2019). More broadly, self-play is widely used to generate automatic curricula near a learner's frontier (Bengio et al., 2009; Silver et al., 2017; Cheng et al., 2024; Kuba et al., 2025; Chen et al., 2024; Zhao et al., 2025; Huang et al., 2025), including open-ended text settings for reasoning/theorem proving (Poesia et al., 2024; Dong & Ma, 2025; Chen et al., 2025; Liu et al., 2025; Yu et al., 2025) and language/program synthesis via adaptive testing and autotelic task design (Ribeiro & Lundberg, 2022; Colas et al., 2022; Parker-Holder et al., 2023; Teodorescu et al., 2023; Pourcel et al., 2024), with proposer–solver variants coupling synthesis to test generation or formal verification (Lin et al., 2025; Wang et al., 2025; Wilf et al., 2025). In contrast, we study fully synthetic short-form self-play where the generator can apply arbitrary text edits and unit tests weakly constrain realism; we therefore mix reference bugs into fixer training and add an embedding-similarity reward to shape open-ended bug generation.

9 Discussion

We scale supervision for code repair in open-ended bug generation using unit tests as verification. We introduce generator–fixer self-play, training a single model with RL to alternate between generating bugs (tests fail) and repairing them (tests pass). However, this setting drifts toward valid but unrealistic bugs, improving in-distribution repair while degrading performance on human-written bugs. To counter this, we propose Anchored Self-Play (ASP), which anchors self-play with a small reference set via an embedding-similarity reward for generation and reference mixing for fixer training. We evaluate bug-source generalization on BugSourceBench, a controlled benchmark spanning human-written bugs, human-edited bugs in model drafts, and bugs from model-generated code. ASP improves average repair over standard self-play and narrows the gap between model-generated and human bug sources, underscoring the need for explicit realism signals in self-play for code repair. Future work includes stronger realism objectives (e.g., learned or preference-based bug-style critics) and richer anchoring signals beyond embeddings.

Acknowledgements

We thank Yangjun Ruan, Neil Band, Kaiyue Wen, Luke Bailey, Thomas Chen, and Arvind Mahankali for helpful discussions and feedback on the paper draft.

BugSourceBench TaskID 1063

Performs Principal Component Analysis (PCA) on the sum of rows of a 2D numpy array and plots the explained variance ratio.
Note that: the title of the plot is set to "Explained Variance Ratio of Principal Components". The function assumes that 'arr' is a valid 2D numpy array. Only the first principal component is considered in this analysis. The plot illustrates the proportion of the dataset's variance that lies along the axis of this first principal component.
The function should output with: ax (matplotlib.axes.Axes) – an Axes object from matplotlib.

Human

```
from matplotlib import pyplot as plt
from sklearn.decomposition import PCA
def task_func(arr):

    row_sums = arr.sum(axis=1)

    pca = PCA(n_components=10)
    pca.fit(row_sums.reshape(-1, 1))

    # Plotting (requires matplotlib and sklearn)

    _, ax = plt.subplots()

    ax.bar([0], pca.explained_variance_ratio)
    ax.set_title("Explained Variance Ratio of
          Principal Components")
    ax.set_xticks([0])
    ax.set_xticklabels(["PC1"])

    return ax
```

Human-Edited LM

```
from matplotlib import pyplot as plt
from sklearn.decomposition import PCA
def task_func(arr):
    if arr.size == 0:
        raise ValueError("Input array is empty.")

    summed = np.sum(arr, axis=1)
    X = summed.reshape(-1, 1)
    pca = PCA(n_components=1)
    pca.fit(X)

    fig, ax = plt.subplots()
    ax.bar([1], pca.explained_variance_,
          color="tab:blue")
    ax.set_xticks([1])
    ax.set_xticklabels(["PC1"])
    ax.set_ylabel("Explained Variance Ratio")
    ax.set_title("Explained Variance Ratio of
          Principal Components")

    return ax
```

LM Errors (Qwen-7B)

```
from matplotlib import pyplot as plt
from sklearn.decomposition import PCA
def task_func(arr):
    # Ensure arr is a 2D numpy array
    if len(arr.shape) != 2:
        raise ValueError("Input must be a 2D numpy
              array")

    # Sum the rows of the array
    summed_data = np.sum(arr, axis=0)

    # Reshape the data to be 2D for PCA
    reshaped_data = summed_data.reshape(1, -1)

    # Perform PCA
    pca = PCA(n_components=1)
    pca.fit(reshaped_data)

    # Get the explained variance ratio
    explained_variance_ratio =
          pca.explained_variance_ratio_

    # Create a plot
    fig, ax = plt.subplots()
    ax.bar(range(1), explained_variance_ratio,
          tick_label=['PC1'])
    ax.set_title('Explained Variance Ratio of
          Principal Components')

    return ax
```

LM Errors (gpt-oss-20b)

```
from matplotlib import pyplot as plt
from sklearn.decomposition import PCA
def task_func(arr):
    # Sum over rows to get column totals
    row_sums = arr.sum(axis=0)

    # Reshape to 2D array for PCA (n_samples=1,
          n_features=number of columns)
    X = row_sums.reshape(1, -1)

    # Perform PCA with a single component
    pca = PCA(n_components=1)
    pca.fit(X)

    # Extract explained variance ratio for the first
          component
    evr = pca.explained_variance_ratio_[0]

    # Plot the explained variance ratio
    fig, ax = plt.subplots()
    ax.bar([1], [evr], width=0.5, color='skyblue')
    ax.set_title("Explained Variance Ratio of
          Principal Components")
    ax.set_xlabel("Principal Component")
    ax.set_ylabel("Explained Variance Ratio")
    ax.set_xticks([1])
    ax.set_ylim(0, 1)

    return ax
```

Figure 6: BUGSOURCEBENCH example showing buggy programs from different sources for the same task x. Bugs differ qualitatively across sources: the human code uses incorrect values that misalign with the spec and misuses the API, the human edit contains multiple logic mistakes (e.g., using absolute variance, forgetting imports), Qwen-7B makes off-by-one errors by using the incorrect axis and shape, and gpt-oss-20b makes similar model mistakes while also showing instruction-following issues.

A BUGSOURCEBENCH

A.1 EXAMPLES

To illustrate cross-source diversity, we show a BUGSOURCEBENCH example containing buggy programs from multiple sources for the same task instance x in Figure A.1. The human-written variant fails via spec-inconsistent values and API misuse; the human-edited variant contains several interacting errors (e.g., incorrect variance formulation and omitted imports); and the LM-generated variants (Qwen-7B, gpt-oss-20b) display typical generation artifacts such as off-by-one mistakes arising from wrong axis/shape handling, with gpt-oss-20b additionally reflecting instruction-following issues.

A.2 CONSTRUCTION

Source tasks. We build BUGSOURCEBENCH from BigCodeBench-style code generation tasks (Zhuo et al., 2024), which emphasize realistic library and API usage. Each task provides (i) natural-language programming instructions, (ii) unit tests that define correctness, and (iii) a reference implementation that passes those tests. We convert each task into a repair instance by pairing the original instructions and tests with a buggy implementation.

Overview. BUGSOURCEBENCH is constructed over a fixed set of BigCodeBench tasks after filtering out tasks whose reference (ground-truth) solutions do not pass the unit tests, resulting in $1{,}114$ tasks. Every BUGSOURCEBENCH instance pairs a task with a *buggy program* that (i) runs/compiles and (ii) fails at least one unit test, so repair requires a semantic fix rather than syntax cleanup. All BUGSOURCEBENCH variants share the same underlying tasks and differ *only* in how the buggy program is produced, enabling controlled comparisons across bug sources.

Bug validity criteria. We first remove 26 BigCodeBench tasks whose reference solutions do not pass the unit tests. We accept a candidate program as a bug if it compiles successfully but fails at least one unit test. Each generated bug is validated by executing the task's unit tests using our standard reward harness. We accept a candidate if (i) it is *not* correct (fails at least one unit test) and (ii) it does *not* trigger compilation or runtime failures (identified via robust pattern matching over test output, e.g., `SyntaxError`, `ImportError`, `NameError`; assertion failures are treated as valid test failures).

Repair interface. We compare several repair prompts, including full-program repair, diff-based patching, and our default interface that includes unit-test feedback (failed tests and truncated error traces). Including test feedback improves pass rates, while diff-based repair often underperforms due to brittle formatting and patch application. We therefore use the test-trace repair interface in all main experiments; ablations are reported in Table 4.

Task structure. All datasets are converted to a consistent BUGSOURCEBENCH-compatible schema. We store *function bodies only* (4-space indented) for both `buggy` and `canonical_solution`. Each example contains:

- `task_id`: Unique identifier for the underlying programming task (shared across bug sources).
- `instruct_prompt`: Natural-language problem statement, including input/output specifications and constraints.
- `buggy`: Buggy solution code for this task (function body only) that runs but fails at least one unit test.
- `canonical_solution`: Reference correct solution code (function body only) used as the ground-truth implementation. *Note: this is never provided to the fixer.*
- `test`: Unit-test harness used to evaluate candidate solutions (typically includes the test cases and a runner).
- `complete_prompt`: Full text prompt given to the model in our default repair interface (instructions + buggy code + test feedback, formatted for the model).
- `code_prompt`: Code-only prompt segment that contains the program context the model is expected to modify/replace (e.g., function signature + buggy body), without natural-language instructions.
- `entry_point`: Name of the function to be implemented/repaired (used by the test harness to call into the solution).
- `doc_struct`: Structured metadata extracted from the problem statement (e.g., function signature, arguments, return type, or other parsed specification fields when available).
- `libs`: List of libraries/modules permitted or required by the task (used to reproduce the intended execution environment).
- `test_output` (added by us): Truncated unit-test feedback produced by running `buggy` on `test` (e.g., failing test names and error traces), used for analysis and as repair context.

This unified schema allows swapping bug sources while holding tasks (instructions and tests) fixed.

Bug sources. We construct four bug-source variants, each providing a different buggy program for the *same* underlying tasks. For fair comparison, we keep only the intersection of `task_id`s that appear in *all* variants.

- **BUGSOURCEBENCH-HUMAN.** Starting from each task's human-written reference solution, two annotators (one CS graduate student and one CS undergraduate) introduce 1–4 localized edits that preserve executability while causing at least one unit test to fail. Annotators are encouraged to introduce realistic developer mistakes (e.g., off-by-one errors, incorrect constants, missing edge cases, or API misuse) rather than syntax-breaking changes.
- **BUGSOURCEBENCH-HUMAN-EDITED-LM.** For each task, we prompt `gpt-5-mini` with the original BigCodeBench instructions *to solve the task* (we do not prompt it to introduce bugs), and resample up to 16 times until obtaining a program that compiles but fails at least one unit test. We retain one such draft and have annotators edit it while keeping it executable and still incorrect, mimicking mistakes that arise when developers modify or integrate LM-generated code. Tasks where no such draft is found within the budget are removed.
- **BUGSOURCEBENCH-QWEN-7B.** For each task, we prompt `Qwen2.5-Coder-7B-Instruct` *to solve the task* (not to generate a bug) and resample up to 16 times until we obtain a program that compiles but fails at least one unit test, retaining one such program per task.
- **BUGSOURCEBENCH-GPT-OSS-20B.** We use the same procedure with `gpt-oss-20b`: prompt it *to solve the task* and resample up to 16 times until we obtain a program that compiles but fails at least one unit test, retaining one such program per task.

Task alignment and splits. We begin from BugBench-style datasets that each provide buggy solutions for a common pool of BigCodeBench tasks. To ensure every bug source is evaluated on an identical task set, we compute the intersection of `task_ids` *separately for each split* across all variants, and filter each variant to that intersection. We provide two evaluation modes: a large test set (`test_all`) of 517 examples per bug source (2,068 instances total) and a smaller test set (`test`) of 127 examples per bug source (508 instances total). Our main experiments use the smaller `test` split for evaluation. We also create a `train` split for each variant by taking task ids not in `test`/`test_all` and intersecting them with BigCodeBench-`train` to avoid leakage; we use these training splits to form the reference pools for ANCHORED SELF-PLAY (ASP).

A.3 BUG SOURCE ANALYSES

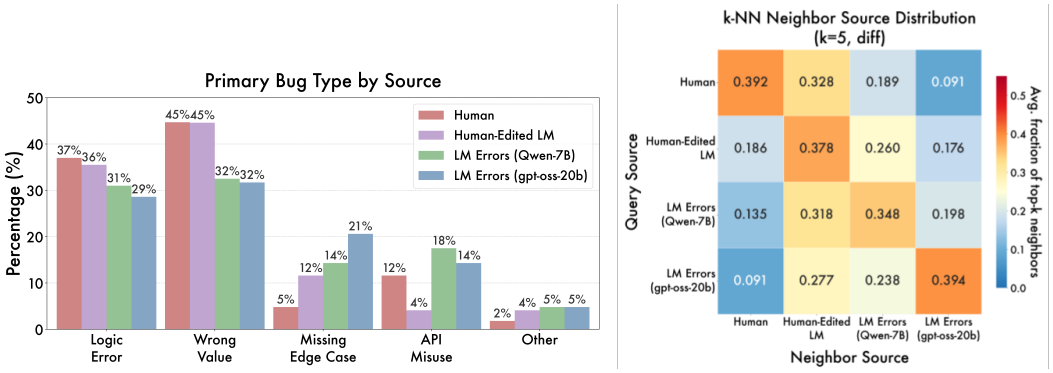
We characterize bugs with their main types and properties. In our main bug type classification, we identify 5 main categories of bugs: LOGIC_ERROR includes bugs where the algorithm or reasoning is incorrect; WRONG_VALUE includes bugs where a specific identifier, literal, or constant is wrong, such as typos, wrong returns, or off-by-one errors; MISSING_EDGE_CASE includes improper or missing handling of edge cases or validations; API_MISUSE includes using a framework API or library improperly, such as wrong methods and types; and OTHER includes those that don't fall directly under one of the prior categories, including missing imports, syntax errors, and others.

With these guides, we label each buggy program with a coarse bug category using GPT-4o, conditioned on the task specification, reference solution, buggy code, and unit-test traces. The results, along with kNN source clustering analysis of the buggy codes, are presented in Figure 7.

Table 3: The testcase failure rates of bugs across sources.

Source	Mean Fail Rate	100% Fail
Human	66.2%	31.5%
Human-Edited LM	54.2%	17.3%
LM Errors (Qwen-7B)	64.5%	29.1%
LM Errors (gpt-oss-20b)	53.5%	20.5%

We further analyze the failure modes of bugs from various sources. We compare the proportion of failed tests of bugs from each source in Table 3. Bugs from the HUMAN source have the highest fail rate and proportion of bugs that fail all unit tests, while bugs from the HUMAN-EDITED LM source are often more subtle and pass multiple unit tests.



(a) Bug-type profiles by source.

(b) kNN source clustering in embedding space.

Figure 7: **Characterizing Bug Sources.** We label each buggy program with a coarse category using GPT-4o and report the resulting distribution for each BUGSOURCEBENCH split. Human edits skew toward logical errors, gpt-oss-20b toward edge-case and constraint violations, and Qwen-7B toward type/API mistakes. For each bug, we embed the diff of code from the reference with `voyage-code-3` and compute the fraction of its k nearest neighbors, excluding neighbors from the same task, that come from each source. Diagonal dominance indicates within-source clustering.

Table 4: **Repair interface comparison.** Pass rate (%) across BUGSOURCEBENCH bug sources for FULL REPAIR, +TESTS (repair with unit-test traces), and DIFF (patch output). CODEGEN is accuracy on the original code-generation tasks.

	Code Generation	Human			Human-Edited LM		
Model	**Score**	FULL	+TESTS	DIFF	FULL	+TESTS	DIFF
GPT-5.2	46.5	67.7	**67.7**	62.2	36.2	**53.5**	32.3
o4-mini	48.0	66.1	**70.9**	55.1	37.8	**56.7**	31.5
Claude-Sonnet	48.0	76.4	**81.1**	72.4	39.4	**51.2**	35.4

	LM Errors (Qwen-7B)			LM Errors (gpt-oss-20b)		
Model	FULL	+TESTS	DIFF	FULL	+TESTS	DIFF
GPT-5.2	19.7	**44.9**	18.1	17.3	**50.4**	18.9
o4-mini	18.9	**44.9**	9.4	15.0	**49.6**	9.4
Claude-Sonnet	15.7	**44.9**	14.2	13.4	**44.9**	14.2

A.4 EVALUATION OF FRONTIER MODELS ON BUGSOURCEBENCH

Table 4 reports pass rates for several frontier models across bug sources and repair interfaces. We summarize several key findings.

Bug-source shift significantly affects performance. Performance varies substantially with bug source. Bugs from incorrect model-generated drafts (Qwen-7B, gpt-oss-20b) are consistently harder than human or human-edited bugs.

A.5 ABLATION OF REPAIR INTERFACES FOR BUGSOURCEBENCH

We compared experiments of models with several repair interfaces commonly used in practice:

- **Code generation:** generate a complete solution from the task description.
- **Full Repair:** given the buggy code, produce a corrected version.
- **Diff Repair:** produce a patch (diff) that modifies the buggy code to fix the error.
- **Prompt with test cases [our default]:** given the buggy code and the unit test error traces, produce a corrected version.

Table 5: Repair rates (%) by dataset and repair model.

	Human			Human-Edited LM		
Outcome	**o4-mini**	**GPT-5.2**	**Sonnet**	**o4-mini**	**GPT-5.2**	**Sonnet**
Repair Fail | Code Pass	16.4%	16.9%	8.2%	27.9%	27.1%	41.0%
Repair Fix | Code Fail	59.1%	54.4%	71.2%	42.4%	36.8%	43.9%

	LM Errors (Qwen-7B)			LM Errors (gpt-oss-20b)		
Outcome	**o4-mini**	**GPT-5.2**	**Sonnet**	**o4-mini**	**GPT-5.2**	**Sonnet**
Repair Fail | Code Pass	42.6%	39.0%	41.0%	41.0%	28.8%	39.3%
Repair Fix | Code Fail	33.3%	30.9%	31.8%	40.9%	32.4%	30.3%

The performance results are included in Table 4. We find that test feedback improves repair pass rate as the traces help in locating some of the causes and types of bugs, while diff-patch based repair often yields lower performance due to the complexity of the format and of repairing code with more restricted output. For Diff-Repair, we implement a custom diff applier which handles fuzzy context matching to best handle the model outputs. In our main experiments we opted to use the testcases-based approach, motivated by its improved performance as well as its alignment with most real-world code repair settings.

Debugging is a distinct skill from code generation. Solving a task from scratch and repairing an existing solution succeed on different instances. In Table 5, we can observe that, for instance, Claude-Sonnet on BugBench-Human has a sizable fraction of cases where code generation passes but repair fails (0.410), and many cases where repair succeeds but code generation fails (0.439). Similar patterns hold across models and splits, indicating that repair requires capabilities, such as localizing faults, that are not captured by end-to-end code generation.

B Training Details

B.1 Optimization with GRPO

We optimize the generator-fixer self-play loop with Group Relative Policy Optimization (GRPO). For each task x, we first sample $G{=}4$ candidate bugs $b_i \sim \pi_G(\cdot \mid x)$. For each bug (x, b_i), we then sample $K{=}4$ independent repair attempts $y_i^{(1:K)} \sim \pi_F(\cdot \mid x, b_i, \mathrm{o}(x, b_i))$ and compute the solve rate $\rho(x, b_i) = \frac{1}{K}\sum_{k=1}^{K} v(x, y_i^{(k)})$.

We write the generator reward for proposing b on task x as $R^{\mathrm{G}}(x, b)$ and the fixer reward for producing repair y on (x, b) as $R^{\mathrm{F}}(x, b, y)$. In standard self-play, R^{F} is the unit-test pass indicator and R^{G} is solve-rate–shaped (Section 4.2); ASP adds auxiliary terms to R^{G} and occasionally replaces b with b^{ref} when updating the fixer (Section 4.5).

Generator update. We update the generator with GRPO. For each task x, we sample G candidate bugs $\{b_i\}_{i=1}^{G}$ and compute group-normalized advantages:

$$\hat{A}_i^G = \frac{R^{\mathrm{G}}(x, b_i) - \mu^G(x)}{\sigma^G(x) + \epsilon}, \quad \mu^G(x), \sigma^G(x) \text{ computed over } i \in \{1, \ldots, G\}.$$

We then optimize a GRPO objective using a clipped policy-gradient update on $\pi_\theta(\cdot \mid x)$.

Fixer update. We update the fixer with GRPO. Using the same sampled bugs, we assign each repair attempt the reward $R^F(x, b_i, y_i^{(k)})$. For each bug b_i, we compute a per-bug baseline across the K repairs and form group-normalized advantages:

$$\hat{A}_{i,k}^F = \frac{R^{\mathrm{F}}(x, b_i, y_i^{(k)}) - \mu^F(x, b_i)}{\sigma^F(x, b_i) + \epsilon}, \quad \mu^F(x, b_i), \sigma^F(x, b_i) \text{ computed over } k \in \{1, \ldots, K\}.$$

We then optimize a GRPO objective using a clipped policy-gradient update on $\pi_\theta(\cdot \mid x, b, \mathrm{o}(x, b))$.

B.2 SIMILARITY-GUIDED GENERATOR SHAPING DETAILS

Reference mixing anchors the fixer but does not encourage the generator to produce reference-like bugs. We therefore add an auxiliary reward to the generator based on similarity to bugs in $\mathcal{D}_{\text{ref}}$.

Edit embedding and kNN similarity. For each generated bug b, we compute a unified diff between the reference solution and b, embed the diff with a code embedding model, and compute its average cosine similarity to the k nearest neighbors in a pool of reference edit embeddings. We map the resulting score to $[0, 1]$ and denote it by $\text{sim}_{01}(b)$.

Similarity reward weight λ. For valid, on-policy generated bugs, we augment the base generator reward with a similarity term weighted by λ:

$$r^{\text{G}}(x, b) = r^{\text{G}}_{\text{base}}(x, b) + \lambda\, \tilde{\delta}(b), \tag{1}$$

where $\tilde{\delta}(b)$ is a centered (and optionally clipped) version of $\text{sim}_{01}(b)$. Larger λ increases pressure to match the reference edit distribution; smaller λ prioritizes the unit-test-based curriculum.

EMA centering with rate β. Raw similarity scores can have a nonzero mean that changes over training, which can cause the auxiliary term to act like a shifting reward offset. To make the shaping term reflect *relative* similarity (above/below the recent average), we maintain an exponential moving average (EMA) baseline B_t:

$$\delta_t(b) = \text{sim}_{01}(b) - B_t, \tag{2}$$

$$B_t \leftarrow \beta B_{t-1} + (1 - \beta)\, \text{sim}_{01}(b), \tag{3}$$

and set $\tilde{\delta}(b)$ to a clipped (optional) version of $\delta_t(b)$. Here $\beta \in (0, 1)$ controls the timescale of the baseline: higher β yields a slower, smoother baseline; lower β adapts more quickly.

Interaction between p_{mix}, λ, and β. Reference mixing (p_{mix}) anchors the *fixer* directly by training on true reference bugs, even if the generator distribution drifts. Similarity shaping (λ) anchors the *generator* by making reference-like edits more rewarding. The EMA rate (β) stabilizes this shaping signal by centering similarity relative to the recent training distribution. In our experiments, we tune p_{mix} and λ to trade off on-policy curriculum vs. reference alignment, and use a high β to keep the baseline stable.

B.3 ANCHORED SELF-PLAY (ASP) & SELF-PLAY HYPERPARAMETERS

We train a Qwen2.5-Coder-7B-Instruct policy with GRPO under the following settings.

- **Optimization and regularization.** Learning rate 1×10^{-6}. PPO-style clipping with ratio 0.28.
- **Batching.** Training batch size 64. Validation batch size 256. PPO minibatch size 32. Dynamic batch sizing enabled. Maximum PPO token budget of $30{,}000$ tokens per GPU.
- **Sequence lengths.** Maximum prompt length 8192 tokens. Maximum response length 2048 tokens.
- **Rollouts.** Asynchronous rollouts. Sampling temperature 0.6 for training and validation. Top-p 0.95 for validation. Samples per prompt: $n = 4$ for training and $n = 1$ for validation.
- **Systems settings.** Gradient checkpointing enabled.
- **Training schedule and logging.** 10 epochs total.
- **Parallelism and hardware.** One node with 8 GPUs. Tensor parallel size 1 and sequence parallel size 1.

Self-play loop hyperparameters. For each task, we sample $G = 4$ candidate bugs. For each bug, we sample $K = 4$ independent repair attempts and compute the solve rate ρ. We use band-shaped generator rewards with $\rho_\ell = 0.25$, $\rho_h = 0.75$, invalid-bug reward -1.0, and extreme-case penalty $\alpha = 0.2$. We include failing test output in the fixer context and normalize advantages separately for the generator and fixer roles.

Anchoring hyperparameters. For ANCHORED SELF-PLAY (ASP), we use a reference dataset from all BUGSOURCEBENCH splits: Human, Human-edited LM, LM Errors (Qwen-7B), LM Errors (gpt-oss-20b). We mix in reference examples for fixer training with a mix ratio $p_{\text{mix}} = 0.5$. We enable embedding-similarity shaping for the generator with weight $\lambda = 0.20$ and use embedding scores from `voyage-code-3` on diffs between the buggy program and reference program. We

compute similarity using diff-based edit embeddings and a k-nearest-neighbor score with $k = 5$. We use margin scoring with temperature 5.0 and an exponential-moving-average baseline with decay $\beta = 0.99$. In each fixer training batch, 20% of the samples are from the reference bugs, randomly sampled from the reference dataset.

B.4 Fixer-only Hyperparameters

We train a Qwen2.5-Coder-7B-Instruct *fixer* with GRPO while keeping the *generator* frozen and served externally. To approximately control for compute relative to Self-Play and ASP, we use $n=16$ actor rollouts per training prompt (validation uses $n=1$).

Data and prompting hyperparameters. To align supervision between Fixer-only and ASP, we train on a mixture of BigCodeBench and the training splits of the reference bug-source datasets (i.e., the same reference/target data available to ASP). During repair, the fixer input includes the failed unit-test output when available. At validation time, we evaluate both repair (given a buggy program) and standard code generation.

- **Optimization and regularization.** Learning rate 1×10^{-6}. PPO-style clipping with ratio 0.28.
- **Batching.** Training batch size 64. Validation batch size 256. PPO minibatch size 32. Dynamic batch sizing enabled. Maximum PPO token budget of $24{,}000$ tokens per GPU.
- **Sequence lengths.** Maximum prompt length 8192 tokens. Maximum response length 2048 tokens.
- **Rollouts.** Asynchronous rollouts. Sampling temperature 0.6 for training and validation. Top-p 0.95 for both the frozen generator and validation sampling. Samples per prompt: $n = 16$ for training and $n = 1$ for validation.
- **Frozen generator configuration.** Generator model: Qwen2.5-Coder-7B-Instruct. Generation temperature 0.6 and top-p 0.95.
- **Training schedule and logging.** 10 epochs total.
- **Parallelism and hardware.** One node with 8 GPUs. Tensor parallel size 1 and sequence parallel size 1.

Data and prompting hyperparameters. We train on a mixture of BigCodeBench and train splits of the reference bug-source datasets to control for data supervision between Fixer-only and ASP. During repair, we include failing unit-test output in the fixer input. At validation time, we also evaluate standard code generation in addition to repair.

Compute. All runs use a single node with 8 H100 GPUs and take approximately 48 hours for 120 training steps.

C Prompts

We use two roles: a *bug generator* that injects subtle faults into a correct reference solution, and a *bug fixer* that repairs the buggy program using unit-test feedback when available. In the prompts below, `<PROBLEM>` and `<CODE>` denote placeholders filled at runtime.

Bug generator prompt

```
You are a *bug generator* for Python solutions to competitive
    programming problems.

You will be given:

1. A problem description.
2. A *correct* reference implementation in Python.

Your task:

- Introduce **one or a few subtle bugs** into the code.
- The resulting code **must still be syntactically valid Python**.
```

```
- It should change the behavior so that **at least one unit test fails**.
- Do **not** drastically rewrite the code; keep the overall structure similar.
- Do **not** change the function signature, imports, or I/O format.
- Output **only** the full buggy Python code inside a single ```python``` block.

Problem:
<PROBLEM>

Correct reference implementation:
<CORRECT_CODE>

Now generate the buggy version of this code. Return the entire function with the buggy code inside a ```python``` block:
```

Bug fixer prompt

```
You are an expert Python debugging assistant.

You will be given:

1. A problem description.
2. A buggy Python implementation that may fail some hidden unit tests.
3. (Optional) Failed unit test output from running the buggy implementation.

Your task:

- Carefully read the code and identify the bug(s).
- Produce a fixed version of the code that makes all unit tests pass.
- Preserve the original function signature, imports, and I/O format.
- Keep the solution reasonably close to the given implementation.
- Output **only** the full corrected Python code inside a single ```python``` block.

Problem:
<PROBLEM>

Buggy implementation:
<BUGGY_CODE>

Failed unit test output (if available):
```text
<FAILED_TEST_OUTPUT>
Now fix the bugs in this code. Return the entire function with the fixed code inside a python block:
```

D Additional Experiments

D.1 Qualitative Examples of Generated Bugs

Figure 8 presents an example of a bug generated by the model before and after ASP training. The base model generates a very simple and artificial typo, and after training creates more logical errors,
```

with missing de-duplication, improper error handling, and failing to follow instructions in creating a new directory.

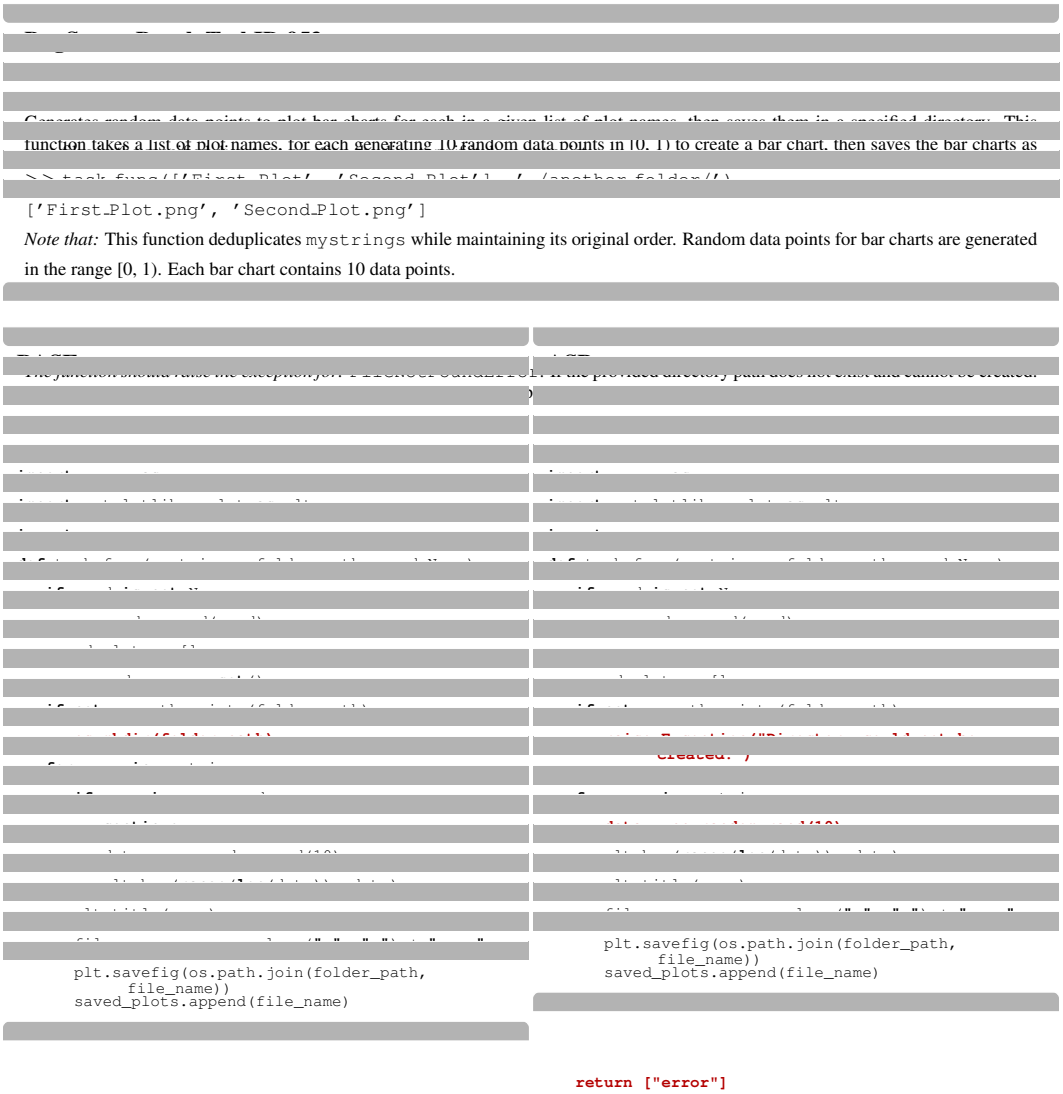


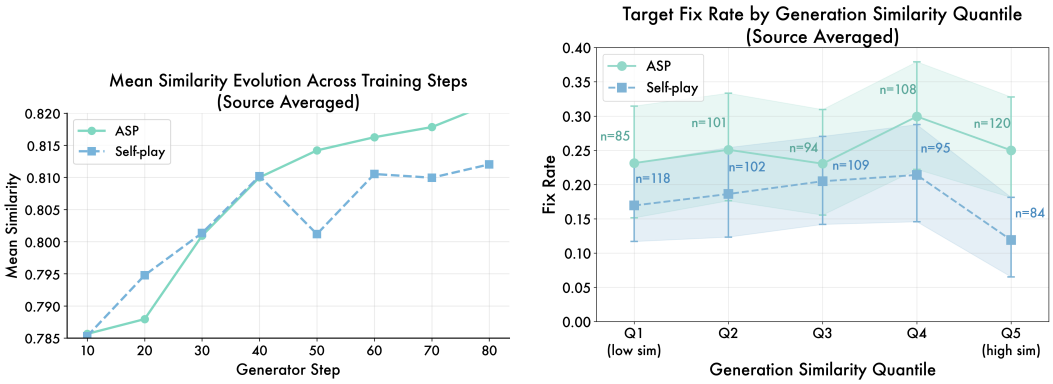
Figure 8: BUGSOURCEBENCH example showing bugs generated by the model before and after anchored self-play training. The base model generates a very simple and artificial typo, and after training creates more logical errors, with missing de-duplication, improper error handling, and failing to follow instructions in creating a new directory.

D.2 SIMILARITY OF GENERATED BUGS

Mean similarity evolution. For each generator checkpoint and method (Self-Play vs. ASP), we sample a large set of synthetic bugs on the tasks from test set. We embed each generated bug (diff-embedding) and compute its similarity to a chosen target bug pool (e.g., BugSourceBench source split) as the mean top-(k) cosine similarity to the pool embeddings (excluding same-task matches when available). We then report the mean similarity across generated bugs as a function of training step, producing a "similarity evolution" curve that tracks how the generator's output distribution moves relative to each target pool over training.

Target fix rate by generation-similarity quantile. To test whether downstream gains are explained solely by generating more target-like bugs, we construct global similarity quantile bins using the generator's similarities to a target pool (shared bin boundaries across methods). For each bin, we evaluate fixer performance on the *test* BUGSOURCEBENCH *instances* whose task IDs correspond

to that bin, reporting the target test solve rate (with confidence intervals) per quantile. This yields a "matched-similarity" diagnostic: it compares ASP vs. Self-Play at comparable levels of generator–target similarity, allowing us to assess whether ASP improves target robustness beyond simply shifting the generator toward higher-similarity bugs.



(a) **Similarity-guided shaping increases target-likeness of generated bugs.**

(b) **Performance on Test Taskd by Generated Similarity.**

Figure 9: We sample $n = 3$ bugs from the generator for the tasks in the held-out BUGSOURCEBENCH test and plot the kNN embedding score of the generations to the target bugs (higher is more target-like). Similarity-guided shaping in ASP yields a consistent increase over training compared to standard self-play, indicating generator outputs move toward the target bug distribution under the shaping signal (left). Next, we bucket tasks into similarity quantiles based on their generated bugs, sample $k = 3$ fix attempts per test task, and report fix rate as a function of the task's corresponding similarity bucket with 95% confidence intervals (right).

D.3 ADDITIONAL BASELINES

We present results for several additional baseline experiments.

Reinforcement Learning for Code Generation, We show that code generation capabilities are distinct from code repair. In order to present an ablation baseline on the impact of training for code generation capabilities, we train the base model on on the task of code generation given the task specs with binary rewards based on testcase execution; we follow aligned settings and training data splits to our standard repair experiments.

Supervised Fine-Tuning for Fixer Training, We present performance for training the fixer with SFT. We train for the task of fixing buggy code from the train instances of BUGSOURCEBENCH across all 4 sources and use a masked cross-entropy loss on the corresponding correct reference solutions.

Table 6: **Repair rates on BUGSOURCEBENCH for additional baselines and code generation pass rate on corresponding problems.** For the BUGSOURCEBENCH splits, we report repair pass rates (in %), and for the codegen interface we report code pass rate.

Method	Codegen	Human	Human-Edited LM	LM (Qwen-7B)	LM (gpt-oss-20b)
Base	21.7	**58.3**	**30.8**	7.1	7.0
Codegen	**31.8**	55.8	**30.8**	2.9	6.3
Fixer SFT	–	46.5	18.1	**10.2**	**11.0**

Results in Table 6 show that while RL post-training on code generation improves code generation capabilities, performance on repair tasks stay stable or even degrade. The SFT-trained Fixer-only model de

E ADDITIONAL RESULTS

We provide a table for the main results in the bar plot of Figure 3.

Table 7: Fix rate (%) by bug source for ANCHORED SELF-PLAY (ASP) versus baselines.

Method	Avg.	Human	Human-Edited LM	LM (Qwen-7B)	LM (gpt-oss-20b)
Base	25.8	58.3	30.8	7.1	7.0
Fixer-only	27.9	58.3	30.7	10.2	12.6
Self-play	29.1	63.7	31.5	8.7	12.6
Anchored self-play (Ours)	**36.1**	**65.0**	**37.0**	**21.3**	**21.3**

REFERENCES

Miltiadis Allamanis, Earl Barr, Marc Brockschmidt, and Oleksandr Polozov. Self-supervised bug detection and repair. In *Advances in Neural Information Processing Systems*, 2021.

Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pp. 41–48. ACM, 2009.

Lili Chen, Mihir Prabhudesai, Katerina Fragkiadaki, Hao Liu, and Deepak Pathak. Self-questioning language models. *arXiv preprint arXiv:2508.03682*, 2025.

Zimin Chen, Michele Tufano, Jevgenija Pantiuchina, Cody Watson, Gabriele Bavota Jiang, and Denys Poshyvanyk. Sequencer: Sequence-to-sequence learning for end-to-end program repair. In *Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019.

Zixiang Chen, Yihe Deng, Huizhuo Yuan, Kaixuan Ji, and Quanquan Gu. Self-play fine-tuning converts weak language models to strong language models. *arXiv preprint arXiv:2401.01335*, 2024.

Jiale Cheng, Xiao Liu, Cunxiang Wang, Xiaotao Gu, Yida Lu, Dan Zhang, Yuxiao Dong, Jie Tang, Hongning Wang, and Minlie Huang. Spar: Self-play with tree-search refinement to improve instruction-following in large language models. *arXiv preprint arXiv:2412.11605*, 2024.

Cédric Colas, Tristan Karch, Olivier Sigaud, and Pierre-Yves Oudeyer. Autotelic agents with intrinsically motivated goal-conditioned reinforcement learning: a short survey. 2022. URL https://arxiv.org/abs/2012.09830.

Kevin Zheyuan Cui, Mert Demirer, Sonia Jaffe, Leon Musolff, Sida Peng, and Tobias Salz. The productivity effects of generative ai: Evidence from a field experiment with github copilot. 2024.

Kefan Dong and Tengyu Ma. Stp: Self-play llm theorem provers with iterative conjecturing and proving. 2025. URL https://arxiv.org/abs/2502.00212.

Shihan Dou, Haoxiang Jia, Shenxi Wu, Huiyuan Zheng, Muling Wu, Yunbo Tao, Ming Zhang, Mingxu Chai, Jessica Fan, Zhiheng Xi, Rui Zheng, Yueming Wu, Ming Wen, Tao Gui, Qi Zhang, Xipeng Qiu, and Xuanjing Huang. What's wrong with your code generated by large language models? an extensive study, 2025. URL https://arxiv.org/abs/2407.06153.

Stephanie Forrest, ThanhVu Nguyen, Westley Weimer, and Claire Le Goues. A genetic programming approach to automated software repair. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation (GECCO)*, 2009.

Jingxuan He, Luca Beurer-Kellner, and Martin Vechev. On distribution shift in learning-based bug detectors. In *International conference on machine learning*, pp. 8559–8580. PMLR, 2022.

Chengsong Huang, Wenhao Yu, Xiaoyang Wang, Hongming Zhang, Zongxia Li, Ruosen Li, Jiaxin Huang, Haitao Mi, and Dong Yu. R-zero: Self-evolving reasoning llm from zero data. *arXiv preprint arXiv:2508.05004*, 2025.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*, 2024.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*, 2023.

René Just, Darioush Jalali, and Michael D Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 international symposium on software testing and analysis*, pp. 437–440, 2014.

Jakub Grudzien Kuba, Mengting Gu, Qi Ma, Yuandong Tian, Vijai Mohan, and Jason Chen. Language self-play for data-free training. *arXiv preprint arXiv:2509.07414*, 2025.

Claire Le Goues, Neal Holtschulte, Edward K Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. The manybugs and introclass benchmarks for automated repair of c programs. *IEEE Transactions on Software Engineering*, 41(12):1236–1256, 2015.

Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. Quixbugs: A multi-lingual program repair benchmark set based on the quixey challenge. In *Proceedings Companion of the 2017 ACM SIGPLAN international conference on systems, programming, languages, and applications: software for humanity*, pp. 55–56, 2017.

Zi Lin, Sheng Shen, Jingbo Shang, Jason Weston, and Yixin Nie. Learning to solve and verify: A self-play framework for code and test generation. *arXiv preprint arXiv:2502.14948*, 2025.

Bo Liu, Chuanyang Jin, Seungone Kim, Weizhe Yuan, Wenting Zhao, Ilia Kulikov, Xian Li, Sainbayar Sukhbaatar, Jack Lanchantin, and Jason Weston. Spice: Self-play in corpus environments improves reasoning. *arXiv preprint arXiv:2510.24684*, 2025.

Fan Long and Martin Rinard. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2016.

Fernanda Madeiral, Simon Urli, Marc Maia, and Martin Monperrus. Bears: An extensible java bug benchmark for automatic program repair studies. In *Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2019.

Gustavo A Oliva, Gopi Krishnan Rajbahadur, Aaditya Bhatia, Haoxiang Zhang, Yihao Chen, Zhilong Chen, Arthur Leung, Dayi Lin, Boyuan Chen, and Ahmed E Hassan. Spice: An automated swe-bench labeling pipeline for issue clarity, test coverage, and effort estimation. *arXiv preprint arXiv:2507.09108*, 2025.

Jack Parker-Holder, Minqi Jiang, Michael Dennis, Mikayel Samvelyan, Jakob Foerster, Edward Grefenstette, and Tim Rocktäschel. Evolving curricula with regret-based environment design. 2023. URL https://arxiv.org/abs/2203.01302.

Minh VT Pham, Huy N Phan, Hoang N Phan, Cuong Le Chi, Tien N Nguyen, and Nghi DQ Bui. Swe-synth: Synthesizing verifiable bug-fix data to enable large language models in resolving real-world bugs. *arXiv preprint arXiv:2504.14757*, 2025.

Gabriel Poesia, David Broman, Nick Haber, and Noah D. Goodman. Learning formal mathematics from intrinsic motivation. 2024. URL https://arxiv.org/abs/2407.00695.

Julien Pourcel, Cédric Colas, Gaia Molinaro, Pierre-Yves Oudeyer, and Laetitia Teodorescu. Aces: Generating a diversity of challenging programming puzzles with autotelic generative models. *Advances in Neural Information Processing Systems*, 37:67627–67662, 2024.

Marco Tulio Ribeiro and Scott Lundberg. Adatest: Adaptive testing for model improvements. In *Advances in Neural Information Processing Systems*, 2022.

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *nature*, 550(7676):354–359, 2017.

Atharv Sonwane, Isadora White, Hyunji Lee, Matheus Pereira, Lucas Caccia, Minseon Kim, Zhengyan Shi, Chinmay Singh, Alessandro Sordoni, Marc-Alexandre Côté, and Xingdi Yuan. Bugpilot: Complex bug generation for efficient learning of swe skills, 2025. URL https://arxiv.org/abs/2510.19898.

Laetitia Teodorescu, Cédric Colas, Matthew Bowers, Thomas Carta, and Pierre-Yves Oudeyer. Codeplay: Autotelic learning through collaborative self-play in programming environments. In *IMOL 2023-Intrinsically Motivated Open-ended Learning workshop at NeurIPS 2023*, 2023.

Yinjie Wang, Ling Yang, Ye Tian, Ke Shen, and Mengdi Wang. Cure: Co-evolving coders and unit testers via reinforcement learning. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025.

Yuxiang Wei, Zhiqing Sun, Emily McMilin, Jonas Gehring, David Zhang, Gabriel Synnaeve, Daniel Fried, Lingming Zhang, and Sida Wang. Toward training superintelligent software agents through self-play swe-rl. *arXiv preprint arXiv:2512.18552*, 2025.

Ratnadira Widyasari, Sheng Qin Sim, Camellia Lok, Haodi Qi, Jack Phan, Qijin Tay, Constance Tan, Fiona Wee, Jodie Ethelda Tan, Yuheng Yieh, et al. Bugsinpy: a database of existing bugs in python programs to enable controlled testing and debugging studies. In *Proceedings of the 28th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, pp. 1556–1560, 2020.

Alex Wilf, Pranjal Aggarwal, Bryan Parno, Daniel Fried, Louis-Philippe Morency, Paul Pu Liang, and Sean Welleck. Propose, solve, verify: Self-play through formal verification. *arXiv preprint arXiv:2512.18160*, 2025.

Frank F Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN international symposium on machine programming*, pp. 1–10, 2022.

John Yang, Kilian Leret, Carlos E Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. Swe-smith: Scaling data for software engineering agents. *arXiv preprint arXiv:2504.21798*, 2025a.

Weiqing Yang, Hanbin Wang, Zhenghao Liu, Xinze Li, Yukun Yan, Shuo Wang, Yu Gu, Minghe Yu, Zhiyuan Liu, and Ge Yu. Coast: Enhancing the code debugging ability of llms through communicative agent based data synthesis, 2025b. URL https://arxiv.org/abs/2408.05006.

Michihiro Yasunaga and Percy Liang. Break-it-fix-it: Unsupervised learning for program repair. In *International Conference on Machine Learning (ICML)*, 2021.

He Ye, Matias Martinez, Xiapu Luo, Tao Zhang, and Martin Monperrus. Selfapr: Self-supervised program repair with test execution diagnostics. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450394758. doi: 10.1145/3551349.3556926. URL https://doi.org/10.1145/3551349.3556926.

Wenhao Yu, Zhenwen Liang, Chengsong Huang, Kishan Panaganti, Tianqing Fang, Haitao Mi, and Dong Yu. Guided self-evolving llms with minimal human supervision. *arXiv preprint arXiv:2512.02472*, 2025.

Andrew Zhao, Yiran Wu, Yang Yue, Tong Wu, Quentin Xu, Matthieu Lin, Shenzhi Wang, Qingyun Wu, Zilong Zheng, and Gao Huang. Absolute zero: Reinforced self-play reasoning with zero data. *arXiv preprint arXiv:2505.03335*, 2025.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, et al. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *arXiv preprint arXiv:2406.15877*, 2024.

Armin Zirak and Hadi Hemmati. Improving automated program repair with domain adaptation. *ACM Trans. Softw. Eng. Methodol.*, 33(3), March 2024. ISSN 1049-331X. doi: 10.1145/3631972. URL https://doi.org/10.1145/3631972.