

Practical Design Decisions Can Matter More Than Training Algorithm Choice: A Study of LLM-Based Rust Bug Repair

Stanford CS224N Custom Project

Ethan Morgan

Department of Computer Science
Stanford University
ethanmo@stanford.edu

Abstract

While large language models demonstrate promising capabilities in automated program repair, practitioners face a set of underexplored design decisions—such as output format, sequence budget, and preference pair construction—that may heavily influence downstream performance. We investigate these factors in a single-turn setting, requiring the model to produce a fix in a single inference call without multi-turn compiler feedback. We fine-tune Qwen2.5-Coder-7B on 3,726 real-world Rust bug fixes from the SWE-smith-rs dataset, systematically comparing three output formats (full-file rewrite, unified diff, and search/replace blocks) and three training paradigms (SFT, DPO, and GRPO) across 747 held-out instances evaluated via ‘cargo test’.

Our findings indicate that foundational design configurations can outweigh the choice of training algorithm. Output format introduces substantial performance variances: unified diff achieves only a 1.1% test pass rate, whereas full-file generation and search/replace reach 47.5% and 35.8% on non-formatting instances, respectively. Furthermore, format selection interacts with evaluation methodologies, artifactually swinging pass rates by up to 25 percentage points. In contrast, the variance between the best and worst evaluated training algorithms is only 7 percentage points. Simple adjustments, such as increasing sequence length, yield a 3.5× improvement in compilation rates with no model changes. Notably, neither DPO nor GRPO surpasses standard SFT in our setting. DPO with trivially constructed preference pairs degrades performance by 5.1 percentage points, and all five evaluated GRPO reward structures fail to improve upon the baseline. We hypothesize that preference optimization does not transfer cleanly from reasoning tasks to code repair due to sparse reward signals at low baseline solve rates and the complexities of constructing robust preference pairs. These results suggest that optimizing foundational design choices and exploring multi-turn agentic loops with compiler feedback remain the most promising directions for advancing automated program repair.

Key Information

- Mentor: Arpandeeep Khatua (akhatua@stanford.edu)
- External Collaborators: N/A
- Sharing project: N/A

1 Introduction

Large language models (LLMs) have demonstrated an increasing capacity to resolve real-world software bugs. Recent systems like SWE-agent and Agentless have achieved strong results on benchmarks such as SWE-bench, though they predominantly target Python, rely on multi-turn agentic architectures, and utilize frontier-scale models. Concurrently, preference optimization techniques, namely Direct Preference Optimization (DPO) and Group Relative Policy Optimization (GRPO), have proven highly effective for mathematical reasoning and instruction-following tasks. A natural hypothesis is that these techniques should excel in code repair, where compilers and test suites provide deterministic, automatic reward signals without the need for human labeling. We initially set out to validate this hypothesis by applying DPO and GRPO to Rust bug repair, leveraging cargo test as an objective reward mechanism.

We scope our investigation to a single-turn setting: the model processes a buggy file and generates a fix in a single shot, isolated from compiler feedback loops. While this presents a more challenging environment than iterative agentic workflows, it effectively isolates the model’s learned capabilities from the benefits derived via interaction. Contrary to our initial expectations, empirical results did not support the superiority of preference optimization in this context; neither DPO nor GRPO yielded improvements over plain supervised fine-tuning (SFT). However, investigating the root causes of this underperformance revealed a more fundamental insight: the practical design decisions established prior to training—such as output format representation, sequence budget allocation, and preference pair construction—exert a substantially larger impact on final performance than the training algorithm itself.

We support this thesis with three primary contributions:

1. **The dominance of output representations:** We demonstrate that output format dictates task feasibility and interacts unexpectedly with evaluation methodologies. Unified diff formats achieve a minimal 1.1% test pass rate due to the strict positional constraints of patch application. Full-file generation is simpler and highly effective when files fit within the context window (achieving 47.5% on non-formatting instances), but it inadvertently passes exact-match whitespace tests, inflating its overall metrics by 25 percentage points. Search/replace blocks scale to arbitrary file sizes but introduce structural failure modes. Crucially, the ~ 35 percentage point performance gap between output formats dwarfs the variance observed across training methods.
2. **Limitations of preference optimization in code repair:** We find that preference optimization does not transfer seamlessly from reasoning tasks to bug repair. DPO utilizing a trivial “do nothing” rejection baseline inadvertently teaches the model to unconditionally modify code, resulting in a 5.1 percentage point degradation. Similarly, GRPO applied at a low base solve rate ($\sim 13\%$) produces insufficient gradient signal; after one epoch, over 88% of outputs remained byte-identical to the SFT baseline across all tested reward functions.
3. **Sequence length as a primary lever:** We identify sequence budget as the single most impactful factor for full-file generation. Expanding the maximum token limit from 2k to 8k yields a $3.5\times$ improvement in compilation rates, entirely independent of model architecture or training updates.

We conduct all experiments on SWE-smith-rs, evaluating models on 747 held-out instances from 6 repositories, curated from a broader set of 3,726 real-world Rust bug fixes. To the best of our knowledge, this work represents the first large-scale fine-tuning study dedicated to logical bug repair in Rust, providing actionable insights for practitioners designing LLM-based repair systems.

2 Related Work

2.1 Automated Program Repair

SWE-bench [1] established real-world bug repair as a benchmark, curating thousands of Python issues from GitHub. SWE-smith [2] generates training data through controlled code mutations—we use the Rust subset. SWE-agent [3] takes a multi-turn agentic approach; Agentless [4] showed simpler pipelines can be competitive. RepairLLaMA [5] studies how different *input* representations

affect repair. Our work is complementary: we look at *output* representations in a fine-tuning setting, applied to Rust instead of Python.

2.2 Output Representations for Code Editing

Aider [6] pioneered the search/replace block format and compared formats by prompting frontier models on small exercises. But nobody has done a systematic comparison of output formats in a *fine-tuning* setting on real bugs. This matters because fine-tuned models have to learn to produce valid output purely from training signal—they don’t get the benefit of in-context examples like prompted models do.

2.3 Preference Optimization for Code

DPO [7] is a stable alternative to RLHF; we use compiler and test results to build preferences. GRPO [8] was introduced for math reasoning in DeepSeek-Math, replacing the critic with group-based advantage estimation. The key detail: DeepSeek-Math used GRPO on tasks where the model could already solve $\sim 50\%$ of problems, giving rich within-group variance. Our code repair task sits at $\sim 13\%$ solve rate—a very different regime. Other related work includes CodeRL [9], RLTF [10], and DPO-F+ [11] for Python repair. Our finding that both DPO and GRPO degrade performance suggests code repair needs more careful preference design than what works for reasoning.

2.4 Rust-Specific LLM Work

Rust is underexplored compared to Python. Eniser et al. [12] find the best LLM only handles 47% of Rust translation tasks. CRUST-Bench [13] reports o1 solves just 15 out of 100 C-to-Rust cases. We’re not aware of prior fine-tuning work for Rust bug repair at this scale.

3 Approach

3.1 Output Formats

In single-turn code repair, the model sees a buggy file and produces a fix in one shot. How you ask it to represent that fix turns out to matter a lot. We compare three formats.

Full-file generation. The model outputs the entire corrected file. This is robust—getting line 50 wrong doesn’t break line 200—but wildly token-inefficient. The median file in our eval set is $\sim 5,300$ tokens, and the typical fix changes 2–3 lines. The model has to faithfully reproduce hundreds of unchanged lines just to express a small edit.

Unified diff. The model outputs a git-style patch applied via `git apply`. Token-efficient (~ 100 – 200 tokens) but brittle: it needs exact line numbers, matching context lines, and correct hunk arithmetic (`@@ -127,6 +127,7 @@`). One wrong line number and the whole patch gets rejected.

Search/replace blocks. Inspired by Aider [6], the model outputs edit blocks:

```
<<<<<< SEARCH
[exact lines to find]
=====
[replacement lines]
>>>>>> REPLACE
```

No line numbers, no structured metadata—just string matching. If the SEARCH block doesn’t match exactly, the edit doesn’t apply, but nothing silently breaks. Completions run 100–300 tokens regardless of file size.

3.2 Training Methods

We use Qwen2.5-Coder-7B [14], a 7B code-specialized model, with three training approaches.

SFT. Standard cross-entropy on (prompt, completion) pairs:

$$\mathcal{L}_{\text{SFT}} = -\mathbb{E}_{(x,y)} [\log p_{\theta}(y \mid x)] \quad (1)$$

DPO [7]. Starting from the SFT checkpoint, with the correct fix as chosen (y_w) and the unchanged buggy code as rejected (y_l):

$$\mathcal{L}_{\text{DPO}} = -\mathbb{E} \left[\log \sigma \left(\beta \left(\log \frac{p_\theta(y_w | x)}{p_{\text{ref}}(y_w | x)} - \log \frac{p_\theta(y_l | x)}{p_{\text{ref}}(y_l | x)} \right) \right) \right] \quad (2)$$

Using the unchanged file as the rejection is a significant shortcut. We didn’t sample wrong fixes from the model—we just used “do nothing” as the negative. This turns out to matter, and we dig into why in Section 5.4.

GRPO [8]. Starting from SFT, we sample $G=4$ candidate fixes per prompt and score them with `cargo test`:

$$R(x, o) = \begin{cases} 1.0 & \text{tests pass} \\ 0.5 & \text{compiles, tests fail} \\ 0.0 & \text{otherwise} \end{cases} \quad (3)$$

Rewards get normalized within each group: $A_i = (r_i - \mu_G) / \sigma_G$. We tried five different reward designs including compile-only, test-only, and a two-phase curriculum (Section 4.6).

3.3 Implementation

We wrote the search/replace parser and applicator (`sr_utils.py`) from scratch. Training uses HuggingFace `trl` (v0.29.0). The evaluation harness—multi-GPU parallel inference with git worktree isolation so we can run `cargo test` concurrently on all 747 instances—is also our custom code.

4 Experiments

4.1 Data

Dataset. SWE-smith-rs has 5,311 instances from 39 Rust repos. We filter to 3,726 single-file instances and split by repository: 2,979 train (33 repos) / 747 eval (6 held-out repos). Every bug is semantic—both the buggy and correct versions compile.

Input. Each prompt has a task instruction (specifying output format), the file path, a natural-language bug description, and the full buggy source file. Prompt templates are in Appendix A.

The clippy problem. 328 of our 747 eval instances (44%) come from rust-clippy, and they’re unusual. 179 of them use `cargo fmt` as a test gate—they’re checking whitespace formatting, not logic. Another 148 use UI snapshot tests that compare exact compiler diagnostic output character-by-character. Neither actually tests whether the code fix is correct. They get near-zero pass rates across every model we tried, for reasons that have nothing to do with repair capability. We report on all 747 instances in the main tables but flag the non-clippy numbers (599 instances) where the distinction matters.

Held-out repositories: rust-clippy (328), rust-itertools (194), rust-csv (82), dust (66), indicatif (48), request (29).

4.2 Evaluation Method

Three metrics:

- **Apply rate** (SR and diff only): did the model’s output successfully apply to the source file?
- **Compilation rate:** does the modified code compile (`cargo check`)?
- **Test pass rate** (primary): do all FAIL_TO_PASS tests pass (`cargo test`)?

We use greedy decoding (temperature 0), generate on 8 GPUs with data-parallel inference, and verify with `cargo` in parallel using git worktrees—all 747 instances in 30–60 minutes.

4.3 Experimental Details

Model. Qwen2.5-Coder-7B (7B parameters, vocabulary size 152,064). Full fine-tuning in bf16. The large vocabulary is relevant: the logit tensor eats $\sim 9.4\text{GB/GPU}$, which causes OOM at sequence lengths beyond 8k on A100 40GB.

Hardware. Full-file experiments: $8\times$ NVIDIA A100 SXM4 40GB (max 8,192 tokens). Search/replace experiments: $8\times$ NVIDIA H200 SXM5 143GB (max 32,768 tokens). This hardware difference is a limitation we discuss in Section 5.5—we couldn’t run everything on the same GPUs.

Training configurations are summarized in Table 1.

Table 1: Training configurations. All models use Qwen2.5-Coder-7B as the base.

Model	Init	Epochs	Seq Len	Eff. Batch	LR	Time
SFT (full-file)	Base	1	8,192	32	$2e-5$	16 min
DPO (full-file)	SFT	1	8,192	16	$5e-7$	32 min
SFT SR	Base	3	32,768	32	$2e-5$	~ 2 hrs
DPO SR	SFT SR	1	16,384	16	$5e-7$	30 min
GRPO variants	SFT SR	1	16,384	$G=4$	$1e-6$	70 min
SFT (diff)	Base	3	8,192	32	$2e-5$	92 min

4.4 Output Format Comparison

Each format breaks in a different place. To see this clearly, we track a “failure funnel”—where instances drop out at each pipeline stage (Figure 1).

Table 2: Failure funnel by output format (all SFT). Each row shows where 747 instances are lost.

Model	N	Apply Fail	Compile Error	Test Failure	Pass
SFT (diff, 8k)	747	716 (96%)	4 (1%)	19 (3%)	8 (1%)
SFT (full-file, 8k)	747	—	519 (70%)	160 (21%)	68 (9%)
SFT SR (32k)	747	195 (26%)	39 (5%)	361 (48%)	152 (20%)

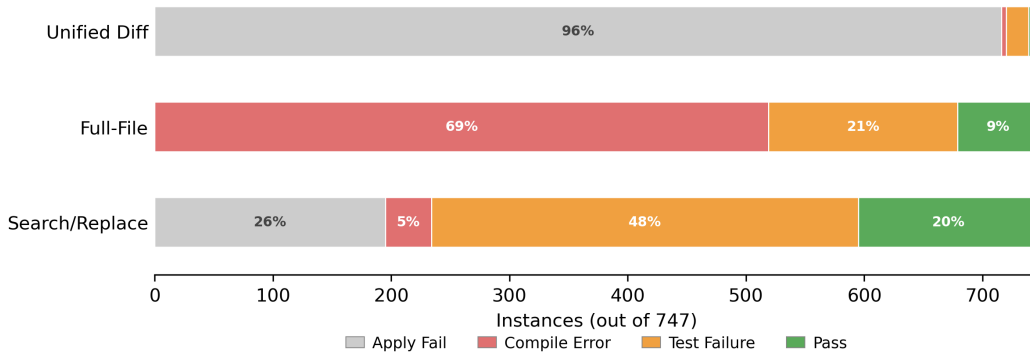


Figure 1: Failure funnel by output format. Unified diff loses 96% of instances at the application stage. Full-file loses most at compilation. Search/replace pushes the bottleneck to test failure—the model is attempting real repairs instead of fighting output formatting.

Unified diff dies at application (96%). The model hits 96% token accuracy during training but produces patches that `git apply` rejects almost every time. The format needs exact line numbers, precise context lines, and correct hunk arithmetic—and the model just can’t reliably count to line 500 in a file it’s seen once.

Full-file dies at compilation (70%). Regenerating an entire valid Rust file under a token budget is hard. The model has to reproduce type signatures, import paths, and trait bounds perfectly for hundreds of lines it isn’t trying to change.

Search/replace pushes the bottleneck to test failure (44%). SR models produce structurally valid edits that are semantically wrong—the format isn’t the bottleneck anymore, program understanding is. The model is attempting real repairs instead of fighting with output formatting.

Table 3: Format comparison summary. Full-file and SR are evaluated on different hardware (see Section 5.5).

Format	Seq Len	Apply/Compile	Test Pass	Non-Clippy	Clippy
Unified diff	8k (A100)	4.1% apply	1.1%	—	—
Full-file	8k (A100)	69.6% compile	37.8%	47.5%	25.3%
Search/replace	32k (H200)	73.9% apply	20.3%	35.8%	0.3%

The overall numbers (37.8% vs 20.3%) are misleading. Breaking it down by subset tells a clearer story:

Full-file beats SR on non-clippy instances (47.5% vs 35.8%). When the file fits in the token budget, full-file’s simplicity is an advantage—just output the correct code. SR adds failure points: the model has to identify the right lines, match them exactly as strings, and then produce the fix. More things to get wrong.

Full-file’s clippy advantage is an evaluation artifact. Full-file gets 25.3% on clippy while SR gets 0.3%. The reason: 179 clippy instances use `cargo fmt -check` as their test gate. Full-file reproduces the entire file including exact whitespace, so formatting checks pass for free. SR blocks can’t preserve exact formatting by construction—any edit disrupts whitespace. This isn’t about repair capability; it’s about an accidental interaction between output format and evaluation method.

The real format story is about tradeoffs, not a single winner. Full-file is simpler and wins when files fit in context, but it hits a hard ceiling at the sequence budget (files above 8k characters approach 0% pass rate). SR scales to any file size—completions are 100–300 tokens regardless—but adds structural failure points. Diff is catastrophically bad at 1.1%. The gap between diff and the viable formats (~ 35 pp) is still much larger than any training method difference.

4.5 Training Method Comparison

Here we hold format constant (full-file) and hardware constant ($8\times$ A100, 8k sequence) to isolate the training method effect.

Table 4: Training method comparison (full-file, $8\times$ A100, $N=747$).

Model	Compile Rate	Test Pass Rate
Zero-shot	64.1%	13.5%
SFT	69.6%	37.8%
DPO	67.9%	32.7%

SFT gives a solid +24.3pp boost over zero-shot. DPO doesn’t improve on that—it actually drops 5.1pp. The gap between the best and worst training method (7pp) is about $5\times$ smaller than the format gap (36pp).

4.6 GRPO: Five Reward Designs, All Degrade

We tested five different GRPO setups, all starting from SFT SR and training for 1 epoch with group size 4. We wanted to understand whether the issue was the reward function or something more fundamental.

Every variant makes things worse. The compile-only version gets the highest training reward (0.55) but that’s misleading—it just reflects the SFT model’s existing compile rate, not anything the model learned. After one epoch, 88–89% of outputs are byte-identical to SFT no matter which reward we

Table 5: GRPO variants. All start from SFT SR (32k) and train for 1 epoch with $G=4$.

Reward Design	Avg Reward	Compile%	Test Pass%	Δ from SFT
SFT baseline	—	68.6%	20.3%	—
Flat (SR=0.15, comp=0.5, test=1.0)	0.12	59.5%	13.7%	−6.6pp
Compile-only (comp=1.0, else=0.0)	0.55	54.9%	3.5%	−16.8pp
Test-only (test=1.0, else=0.0)	0.12	42.7%	0.0%	−20.3pp
Curriculum P1→P2 (comp→test)	0.55→0.12	52.0%	2.6%	−17.7pp
From DPO (flat reward)	0.12	58.0%	13.0%	−7.3pp

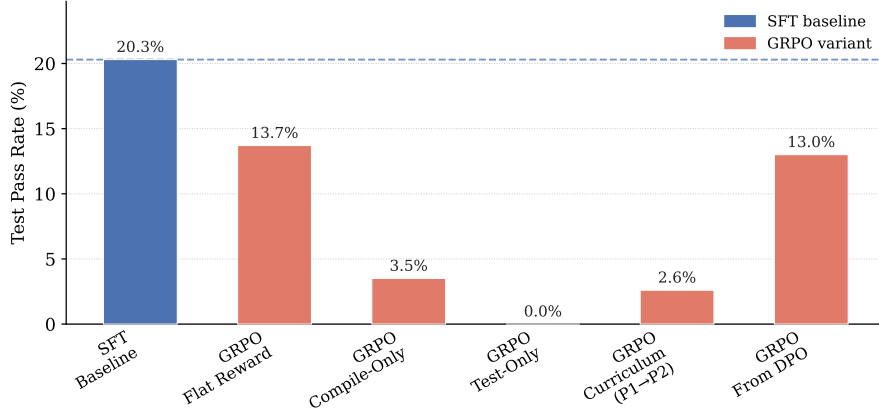


Figure 2: Test pass rate across GRPO variants. The dashed line marks the SFT baseline (20.3%). Every GRPO variant degrades performance regardless of reward design.

use. The test-only variant is the most destructive: with 0 reward for 85% of samples, the sparse signal actually damages the policy. We dig into the mechanics of these failures in Section 5.4.

4.7 Sequence Length as the Dominant Factor

Table 6: Effect of generation budget (zero-shot, full-file format).

max_tokens	Compile Rate	Test Pass Rate
2,048	18.1%	6.3%
8,192	64.1%	13.5%

Just giving the model more room to write—2k to 8k tokens, same model, no training—yields $3.5\times$ better compilation and $2.1\times$ better test pass rate. Since 60% of our eval files are longer than 8,000 characters, the 2k setting is really measuring how well the model handles truncation, not how well it fixes bugs. SR sidesteps this entirely: completions are 100–300 tokens no matter how big the file is.

4.8 Per-Repository Results

The model gets above 50% on dust and request—smaller codebases with self-contained bugs. Clippy’s near-zero is the evaluation artifact from Section 4.1, not a model failure.

5 Analysis

5.1 Bug Characteristics and Learnability

Not all bugs are equally learnable. Mutation type is a strong predictor: operator mutations (op_swap, flip_operators) hit 18–21% SR SFT pass rates, while structural mutations (ctrl_shuffle, op_break_chains) stay at 0% across every model (Table 8). The model can learn to reverse

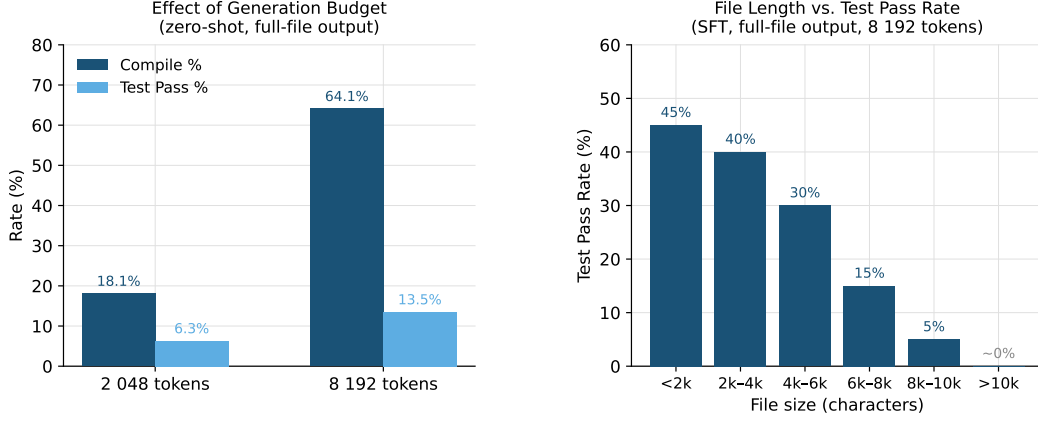


Figure 3: Left: extending `max_tokens` from 2k to 8k yields $3.5\times$ compile improvement with no model changes. Right: test pass rate drops sharply with file length, approaching 0% above 10k characters.

Table 7: SFT SR 32k per-repository results ($N=747$ total).

Repository	N	SR Apply%	Compile%	Test Pass%
request	29	75.9%	72.4%	55.2%
dust	66	86.4%	83.3%	54.5%
rust-csv	82	45.1%	42.7%	37.8%
indicatif	48	56.2%	54.2%	33.3%
itertools	194	76.3%	60.8%	26.3%
rust-clippy	328	79.6%	78.7%	0.6%

localized, formulaic changes—swap an operator back, flip a condition—but it can’t reconstruct shuffled control flow, which requires reasoning beyond the immediate edit site.

Table 8: Test pass rate by mutation type.

Mutation Type	N	Zero-shot	Full-file SFT	SR SFT
op_swap	39	0.0%	0.0%	20.5%
flip_operators	111	4.5%	9.9%	19.8%
op_change	124	7.3%	17.7%	17.7%
ctrl_invert_if	156	5.1%	7.1%	14.1%
ctrl_shuffle	12	0.0%	0.0%	0.0%
op_break_chains	10	0.0%	0.0%	0.0%

Change size tells a similar story: 1-line fixes hit 20.2% SR pass rate; anything over 50 lines hits 0%.

5.2 Error Profile Evolution

The compiler errors shift in an interesting way after SFT (Table 9). Zero-shot errors are mostly syntactic: E0765 (unterminated literals, 20 instances), E0282 (missing type annotations, 14). After SFT, E0765 drops to 9 but E0599 (method not found) jumps from 2 to 27 and E0308 (type mismatch) goes from 3 to 15. What’s happening is that SFT makes the model more confident—it stops producing tentative, half-finished output and starts making ambitious fixes that sometimes reference APIs that don’t exist. It’s hallucinating project-specific methods.

DPO has almost the exact same error profile as SFT. It shifts the distribution without actually improving the model’s understanding of the code.

Table 9: Compiler error evolution (full-file format, top 5 error codes).

Error	Description	Zero-shot	SFT	DPO
E0765	Unterminated literal	20	9	11
E0282	Type annotations needed	14	5	7
E0599	Method not found	2	27	28
E0308	Mismatched types	3	15	15
E0277	Trait bound not satisfied	5	15	15

5.3 Why Unified Diff Fails

The unified diff format needs exact line numbers, precise context lines, and correct hunk arithmetic (@@ -127,6 +127,7 @@). These structured metadata tokens get interleaved with the actual code, and the model can’t handle it—it can’t reliably count to line 500 in a file it’s seen once. `git apply` is all-or-nothing: one wrong line number rejects the entire patch. We hit 96% token accuracy during training but only 4.1% of patches actually apply. The takeaway: formats that mix structured positional metadata with natural content hit a precision floor that current LLMs can’t clear.

5.4 Why Preference Optimization Did Not Improve

DPO and GRPO failed for different reasons, and both are instructive about when these methods transfer from reasoning to code.

DPO: the “do nothing” rejection teaches the wrong thing. We used the unchanged buggy code as the rejected response—the simplest possible preference pair. But this teaches the model “always change something” rather than “change the right thing.” The model learns to be over-aggressive, generating edit blocks that look plausible but modify the wrong code. What DPO actually needs here is model-sampled rejections: generate candidate fixes, test them, use the ones that fail as negatives. That’s expensive—you need inference plus verification for every rejection—but it’s the only way to give the model an informative signal about what kinds of changes are wrong.

GRPO: not enough signal at low solve rates. With $\sim 20\%$ test pass rate and $G=4$, most groups get identical zero rewards, which means near-zero gradient. This is a task structure problem: DeepSeek-Math [8] used GRPO on tasks where the model already solved $\sim 50\%+$ of problems, creating real within-group variance. At 13%, roughly 1 in 8 groups has any positive test reward. The other 7 groups contribute nothing. After one epoch, 88–89% of outputs are byte-identical to SFT regardless of which reward function we use.

Curriculum training confirms reward hacking. We tried a two-phase approach: Phase 1 with compile-only reward (to get the model compiling more reliably), then Phase 2 with test reward. Phase 1 got the highest training reward of any variant (0.55) but the worst test pass rate (3.5%). The model figured out how to maximize compilation without actually fixing bugs—classic reward hacking. Phase 2 couldn’t recover from this in one epoch.

We’re not saying these methods can’t work for code repair. But they need: (1) model-sampled rejections for DPO instead of trivial ones, (2) larger group sizes (8–16) and more epochs for GRPO, and (3) a high enough base solve rate to produce within-group variance. These are specific, testable predictions—not hand-waving.

5.5 Format-Evaluation Interaction and Limitations

The clippy results reveal something worth highlighting: output format doesn’t just affect whether the model can express its fix—it interacts with the evaluation methodology itself. Full-file gets 25.3% on clippy largely because reproducing the entire file preserves exact whitespace that `cargo fmt -check` requires. SR gets 0.3% because edit blocks can’t preserve formatting by construction. Neither result reflects repair capability—they reflect an accidental interaction between format and evaluation. This is a 25pp swing from something nobody would think to control for, and it suggests that format comparisons in code repair need to account for what the tests actually measure.

We should also be upfront about hardware: full-file ran on A100s (40GB, 8k tokens) and SR ran on H200s (143GB, 32k tokens). We had to switch because Qwen2.5-Coder-7B’s 152k vocabulary creates

a 9.4GB/GPU logit tensor that makes long sequences impossible on A100. We tried full-file SFT at 32k on H200 but ran into evaluation harness issues that prevented reliable test pass measurement. The full-file vs diff comparison (both A100, 8k) is clean, and the training method comparisons within each format use identical hardware. The cross-format comparison between full-file and SR comes with both caveats—different hardware and different format-evaluation interactions.

5.6 Qualitative Examples

Example 1: Partial vs. complete fix. A CSV filter has two operator errors (`||` should be `&&`, `<=` should be `==`). Zero-shot catches one; SFT gets both:

```
// Buggy:  if record.country == "us" || record.region <= "MA"
// ZS:      if record.country == "us" && record.region <= "MA"
// SFT:     if record.country == "us" && record.region == "MA"
```

Example 2: Near-miss semantic error. A heap utility has $2*n - 1$ instead of $2*n + 1$. SFT compiles but changes the wrong tuple element—it’s doing surface pattern matching, not understanding the program.

Example 3: Truncation vs. targeted edit. A 30,652-character file needs one operator changed (`-` to `/`). Full-file SFT truncates at token 2,048. SR emits a 506-character block—60× compression, which is exactly why format choice matters so much for large files.

6 Conclusion

We set out to improve Rust bug repair with preference optimization and found that it didn’t help—but the investigation taught us something we think is more useful. Practical design decisions—output format, sequence budget, how you construct preference pairs—can have a bigger impact on performance than which training algorithm you use.

The concrete results: SFT on real bug fixes works well (+24.3pp over zero-shot). Output format creates a ~35pp feasibility gap between diff and the viable formats, and interacts with evaluation methodology in ways that swing results by 25pp. Full-file is simpler and wins when files fit in context; SR scales to any file size. And preference optimization actually hurts in our setting, because of trivial rejections (DPO) and reward sparsity at low solve rates (GRPO).

This study has real limitations: one model size (7B), one language (Rust), and the format comparison is confounded by different hardware and format-evaluation interactions. GRPO was limited to 1 epoch with group size 4—more compute might change the picture.

The natural next steps are model-sampled DPO rejections, GRPO with larger groups at higher base solve rates, multi-turn generation with compiler feedback loops, and a clean cross-format comparison on the same hardware.

References

- [1] 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? *International Conference on Learning Representations*, 2024.
- [2] John Yang, Carlos E Jimenez, Alexander Wettig, Ofir Press, and Karthik Narasimhan. SWE-smith: Scaling data for LLM-based software engineering. *arXiv preprint arXiv:2504.21798*, 2025.
- [3] John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Liber, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. *arXiv preprint arXiv:2405.15793*, 2024.
- [4] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying LLM-based software engineering agents. *arXiv preprint arXiv:2407.01489*, 2024.

- [5] André Silva, Sen Fang, and Martin Monperrus. RepairLLaMA: Efficient representations and fine-tuned adapters for program repair. *arXiv preprint arXiv:2312.15698*, 2025.
- [6] Paul Gauthier. Aider: AI pair programming in your terminal. <https://aider.chat>, 2023.
- [7] Rafael Rafailov, Archit Sharma, Eric Mitchell, Stefano Ermon, Christopher D Manning, and Chelsea Finn. Direct preference optimization: Your language model is secretly a reward model. *Advances in Neural Information Processing Systems*, 36, 2023.
- [8] Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Mingchuan Zhang, Y.K. Li, Y. Wu, and Daya Guo. DeepSeekMath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.
- [9] Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Hoi. CodeRL: Mastering code generation through pretrained models and deep reinforcement learning. *Advances in Neural Information Processing Systems*, 35, 2022.
- [10] Jiate Liu, Yiqin Xia, Yongbin Wang, Yiming Zhang, Jianhua Huang, et al. RLTF: Reinforcement learning from unit test feedback. *arXiv preprint arXiv:2307.04349*, 2023.
- [11] Sen Fang, André Silva, and Martin Monperrus. DPO-F+: Preference optimization for python repair with execution feedback. *arXiv preprint*, 2025.
- [12] Hasan Ferit Eniser, Hanliang Zhang, Cristina Brito, et al. Towards translating real-world code with LLMs: A study of translating to Rust. *arXiv preprint arXiv:2405.11514*, 2024.
- [13] Anirudh Khatri, Sriram Pavithran, Vaibhav Gurbani, et al. CRUST-Bench: A comprehensive benchmark for C-to-Rust translation. *arXiv preprint arXiv:2504.15254*, 2025.
- [14] Binyuan Hui, Jian Yang, Zeyu Cui, Jiayi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, et al. Qwen2.5-Coder: A large language model family for code. *arXiv preprint arXiv:2409.12186*, 2024.

A Prompt Templates

Full-file prompt:

The following Rust file has a bug. Fix the bug and return the complete corrected file.

File: {file_path}
Bug description: {problem_statement}

Buggy code:
```rust  
{buggy\_content}  
```

Fixed code:
```rust

### Search/replace prompt:

The following Rust file has a bug. Output your fix using this exact format:

```
<<<<<< SEARCH
<copy the exact lines you want to change>
=====
<the replacement lines>
>>>>>> REPLACE
```

Rules:  
- The SEARCH block must match the file exactly  
- Use multiple blocks for multiple changes

File: {file\_path}  
Bug description: {problem\_statement}

Buggy code:  
```rust  
{buggy_content}
```

Fix:

### Unified diff prompt:

The following Rust file has a bug. Output a unified diff patch to fix the bug.

File: {file\_path}  
Bug description: {problem\_statement}

Buggy code:  
```rust  
{buggy_content}
```

Diff:  
```diff

B Noise and Reproducibility

Two evaluations of the same SFT SR checkpoint produced 152 vs 99 test passes (7pp gap) due to batched generation non-determinism (349/747 outputs differed despite temperature=0) and cargo test flakiness (42 instances with identical outputs but different verdicts). Our main findings—SFT effectiveness (+24pp), output format impact (~35pp gap), and truncation as the primary bottleneck—all have signal-to-noise ratios above 10 $\times$ and are robust. Smaller effects like the DPO-SFT gap (5pp) should be read directionally.