LEARNING FROM FAILURES: VERIFIED EDIT DISTILLATION FOR VARIABLE-LENGTH DIFFUSION LANGUAGE MODELS

Anonymous authors
Paper under double-blind review

ABSTRACT

Variable-length diffusion language models insert and delete tokens as they generate, so their outputs are not tied to a fixed set of positions. This matters when the model is to learn from its own failed output. A corrected version of a failure usually differs from it in length, so the two do not share positions, and the standard masked SFT objective, which supervises tokens at fixed positions, can only treat the correction as a whole new response; it cannot express which tokens of the failure to keep and which to change. We propose Verified Edit Distillation (VEDI), which obtains such corrections as execution-verified teacher repairs of the model's own failures. VEDI aligns each failure with its repair, accounting for insertions and deletions, and jointly trains one backbone to predict where to edit and to reconstruct only the edited region given the preserved context. Inference uses the original sampler without a teacher or an additional repair stage. On two 7B models, FlexMDM and DreamOn, VEDI improves over Base on all five benchmarks, with gains of up to 12.7 HumanEval+ points on DreamOn. Trained on the same failures and repairs, full-rewrite SFT recovers only part of this gain: VEDI exceeds it by 6.3 and 5.0 HumanEval+ points on FlexMDM and DreamOn and matches or exceeds it on MBPP+. Controlled comparisons show that the gain requires a real student failure, a repair written for that failure rather than a reference solution, and edit-location supervision that reaches the shared backbone.

1 INTRODUCTION

Masked diffusion language models generate by repeatedly reconstructing masked tokens, allowing positions to be resolved in a non-causal order (Austin et al., 2021a; Sahoo et al., 2024). Recent variable-length variants, FlexMDM and DreamOn, also insert and remove tokens during generation (Kim et al., 2026b; Wu et al., 2026). This removes a known failure mode of fixed-length decoding, whose accuracy collapses when the length budget is mis-set (Kim et al., 2026a), and lets the model decide its own length, which matches or improves the best fixed-length setting on reasoning and code (Li et al., 2026; Kim et al., 2026b). Post-training for these models, however, still scores or reconstructs a complete response, as in supervised fine-tuning and reinforcement learning. The new ability to express an edit to a completed program as keeping, remasking, deleting, and inserting tokens has not yet been used as a source of supervision.

What a failure can teach depends on which correct program it is compared with. Masked training usually corrupts a reference answer and reconstructs it. A reference, however, is written without seeing what the student produced. When it differs from the student's own way of solving the problem, the student is not refining what it already does but learning a different solution from scratch. On our data, a reference requires 1.8 to 2.5 times as many edits as a repair written for the failure, and aligning the failure to a reference instead of a repair costs more than 13 HumanEval+ points on both backbones (Section 5.1). A repair r produced from the student's own failure s stays close to it, but it usually has a different length (97% of the verified repairs in our data change length), so the positions of s and r do not correspond and a standard masked objective has no fixed set of positions at which to supervise. Comparing the two position by position inflates the apparent edit 4.3 times at the median; only an alignment that accounts for insertions and deletions recovers which tokens to keep and which to change (Appendix C.1). One choice then remains. Even with the right target, training on the repair

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

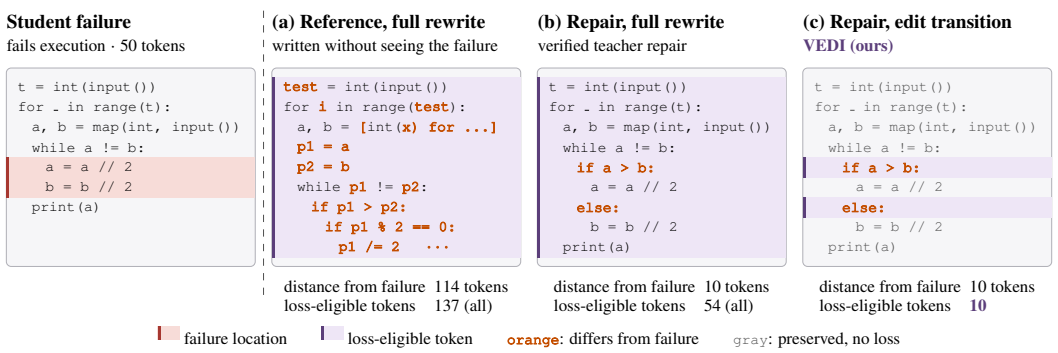


Figure 1: **Three ways to learn from the same failure.** (a) Aligning the failure to the reference rewrites 114 of 137 tokens. (b) The verified repair changes two lines, but full-rewrite training makes every token eligible for loss. (c) VEDI makes only the changed tokens eligible. Orange: differs from the failure; shaded: eligible for loss; gray: preserved.

as a complete output does not distinguish tokens that should be preserved from those that should be changed. For an editable model, the useful supervision is the transition $\Delta(s, r)$ from the failed program to a correct one. Figure 1 illustrates the three targets on one failure.

We introduce **Verified Edit Distillation (VEDI)**, which learns this transition from model-generated failures. We sample completed rollouts from a fixed student checkpoint p_{θ_0} and execute them. A rollout that passes the execution validator $\mathcal{V}$ becomes an identity target that preserves the program; a failed rollout enters the repair procedure. For each failure, a black-box teacher proposes K candidate repairs given the problem, the failure, and its execution feedback, and only candidates that pass the same validator form the verified repair set $\mathcal{R}^+$. Aligning the selected repair with the failure yields an edit transition $\Delta = (a, \ell, v)$: a KEEP/REMASK/DELETE label a for every token, an insertion count ℓ for every gap, and the contents v to fill in. Two losses share this transition. An EditSelector predicts where to edit (a and whether $\ell > 0$), and a local reconstruction loss rebuilds only the edited region with the backbone's own denoising objective, conditioned on the preserved context. Both update a single backbone θ. The teacher supplies only text, so VEDI needs neither teacher logits, a shared tokenizer, nor a shared generation order. At inference the selector is discarded and the backbone runs with its original sampler.

Our experiments separate three choices that prior work couples: whether the failure is the student's own, whether the target is a reference or a repair written for that failure, and whether the target is learned as a whole response or as an aligned edit transition. Each choice helps, and the last helps most. On DreamOn, learning the full repair with the failure in context raises HumanEval+ by 7.7 points, and learning the same failure and repair as an edit transition raises it by 12.7. Controlled comparisons then show where the gain comes from: it shrinks when the failure is replaced by a synthetically corrupted correct program, when the reconstruction region is moved to a random position, and when the selector's gradient is blocked from the backbone. The failure must therefore be real, the reconstruction region must be the true edit site, and the signal about where to edit must shape the shared backbone.

Our main contributions are:

- We formulate post-training for variable-length diffusion language models as learning execution-verified edit transitions from model-generated failures, and give a deterministic procedure that converts black-box text repairs into token-action, insertion-count, and reconstruction targets that exactly reconstruct the selected repair.
- On two 7B backbones, this supervision improves over Base on every benchmark and yields the largest average gain among the compared training methods, using the same failures and repairs.
- Controlled comparisons isolate why: the failure must be the student's own, the target must be written for that failure, and edit-location supervision must reach the backbone.

2 Background and Related Work

2.1 Variable-Length Diffusion Language Models

Masked diffusion language models generate by iteratively predicting masked tokens, so positions need not be resolved left to right (Austin et al., 2021a; Sahoo et al., 2024). LLaDA and Dream scale this paradigm to general-purpose models (Dream by adapting a pretrained autoregressive model), both aligned with supervised fine-tuning (SFT) (Nie et al., 2025; Ye et al., 2025). These models generate on a fixed-length canvas (Nie et al., 2025; Wu et al., 2026). FlexMDM learns, for each gap, the expected number of tokens still to be inserted there and uses it as an insertion rate, interleaving insertion with unmasking (Kim et al., 2026b); DreamOn adds two length-control states, [expand] and [delete], that grow or shrink the canvas (Wu et al., 2026). Neither natively revises tokens it has already decoded. Their length-changing operations nevertheless make it possible to express a revision of a completed program as keeping, remasking, deleting, and inserting tokens, which is the supervision we construct in Section 3. To our knowledge, no post-training method yet supervises such revisions directly.

2.2 Post-Training and Self-Correction for Diffusion Language Models

Post-training for diffusion models has mainly adapted objectives that score a complete response: LLaDA 1.5 reduces the variance of preference optimization, and d1 and DiffuCoder develop diffusion-specific reinforcement learning (Zhu et al., 2025; Zhao et al., 2025; Gong et al., 2025). Closer to our work, ProSeCo trains a masked diffusion model to correct its own predicted tokens (Schiff et al., 2026), and ME-DLM adds an edit stage with replacement, deletion, and insertion targets on top of a fixed-length model (Ren et al., 2026). Both derive targets from a ground-truth reference and spend inference steps on correction: ProSeCo interleaves corrector passes with unmasking, and ME-DLM reserves part of a fixed step budget for editing. Our Reference Edit control aligns a student failure with the reference in the same way and tests this choice of target (Section 5.1); VEDI instead targets a repair written for the failure and keeps the original sampler.

2.3 Distilling Repairs from Model Failures

Knowledge distillation trains a student on teacher outputs, either complete sequences (Kim & Rush, 2016) or teacher token distributions over student-generated sequences (Agarwal et al., 2024). For code, execution provides a verifier: synthetic programs can be filtered by tests before training (Haluptzok et al., 2023; Rozière et al., 2023), and execution outcomes can train reward models or verifiers that select among candidates (Le et al., 2022; Ni et al., 2023). A second line conditions learning on the model's own failed program. CYCLE, LETI, and RLEF feed the failure and its execution feedback back to the model and train toward the reference solution, a reward, or the model's own next attempt (Ding et al., 2024; Wang et al., 2024; Gehring et al., 2025); SCoRe does so from a generic retry prompt without feedback (Kumar et al., 2025). These targets are written without seeing the failure, which our Reference ProSeCo and Reference Edit controls test (Section 5.1). OpenCodeInterpreter instead distills GPT-4 revisions of GPT-3.5 drafts given execution output (Zheng et al., 2024), but the drafts are not the student's own. In all of these, the corrected program is learned as a complete rewrite, which corresponds to our Repair SFT + Draft baseline (Section 5.2). LintSeq trains on edits, but synthesizes them from correct programs rather than deriving them from failures (Piterbarg et al., 2025). VEDI distills execution-verified teacher repairs of the student's own failures and aligns each pair into an edit transition, so supervision falls only on what must change.

3 Verified Edit Distillation

Verified Edit Distillation (VEDI) converts completed student rollouts into edit-structured post-training supervision in three stages. We first collect and execute rollouts from a fixed student checkpoint (Section 3.1). For failed rollouts, we then obtain execution-validated teacher repairs, align each failure–repair pair, and select one repair target (Section 3.2). Finally, we train the shared diffusion backbone with edit-location and local-reconstruction objectives (Section 3.3).

3.1 Model-Generated Failure Collection

Figure 2 summarizes how we collect failures and construct verified repairs. For problem x, the student samples a completed response $s \sim p_{\theta_0}(\cdot \mid x)$. Each rollout exposes the model's own errors, partial solutions, and choice of response length. We execute the raw decoded response directly, without code extraction or syntax filtering. The execution result routes each rollout into one of two supervision paths. **A passing rollout** receives an identity target: every token is labeled KEEP, every gap gets an insertion count of zero, and no tokens need to be reconstructed. **A failed rollout** is sent to the repair procedure in Section 3.2.

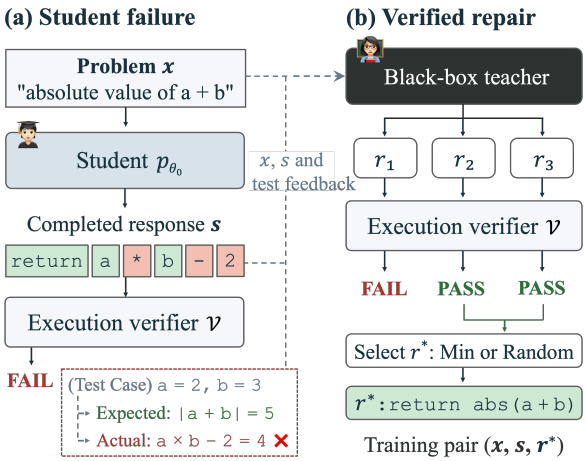


Figure 2: **Failure-conditioned repair construction.**

3.2 Execution-Validated Repair Selection and Alignment

Let $f(x, s)$ denote the failure feedback for response s (e.g., error messages and failing test cases). A black-box teacher conditioned on (x, s, f) produces K candidate programs $r_1, \ldots, r_K$. Each candidate is executed with the same pipeline used for student rollouts, and only passing candidates are kept. Let $\mathcal{V}(x, r) \in \{0, 1\}$ indicate whether program r passes all tests for x. The verified repair set is

$$\mathcal{R}^+(x, s) = \{\, r_k : 1 \le k \le K,\ \mathcal{V}(x, r_k) = 1 \,\}.$$

If $\mathcal{R}^+(x, s) = \varnothing$, the rollout provides no repair target and is excluded from training.

For every $r \in \mathcal{R}^+(x, s)$, we align s to r and derive an edit transition $\Delta(s, r)$ satisfying $E(s, \Delta(s, r)) = r$. Our main experiments align tokens with an exact longest common subsequence (LCS); an AST-based alignment is reported in Appendix B.2. The transition decomposes as $\Delta(s, r) = (a, \ell, v)$: $a_i \in \{\text{KEEP}, \text{REMASK}, \text{DELETE}\}$ labels each existing token $i = 1, \ldots, |s|$; $\ell_j \in \mathbb{N}$ gives the number of tokens inserted at each gap $j = 0, \ldots, |s|$; and v supplies the contents of remasked and inserted positions.

VEDI-Min selects the verified repair whose transition has the lowest edit cost, with a fixed total order $\prec$ resolving ties:

$$r^* = \min_{\prec} \operatorname*{arg\,min}_{r \in \mathcal{R}^+(x,s)} C\big(\Delta(s, r)\big), \qquad \Delta^* = \Delta(s, r^*) = (a^*, \ell^*, v^*), \tag{1}$$

where the edit cost $C(\Delta) = \sum_{i=1}^{|s|} \mathbf{1}[a_i \neq \text{KEEP}] + \sum_{j=0}^{|s|} \ell_j$ counts remasked, deleted, and inserted tokens. This is a deterministic projection only over the finite, teacher-discovered verified set; it neither claims global minimality nor assumes that the closest repair is best for generalization. VEDI-Random, which samples uniformly from the verified set, tests whether this selection rule explains the empirical gain. By construction, the selected transition exactly recovers the selected repair, $E(s, \Delta^*) = r^*$, so the derived targets reproduce an execution-valid program.

3.3 Edit-Supervised Student Update

Figure 3 summarizes how the derived targets supervise the shared diffusion backbone and how the trained model returns to its original generation interface at inference time.

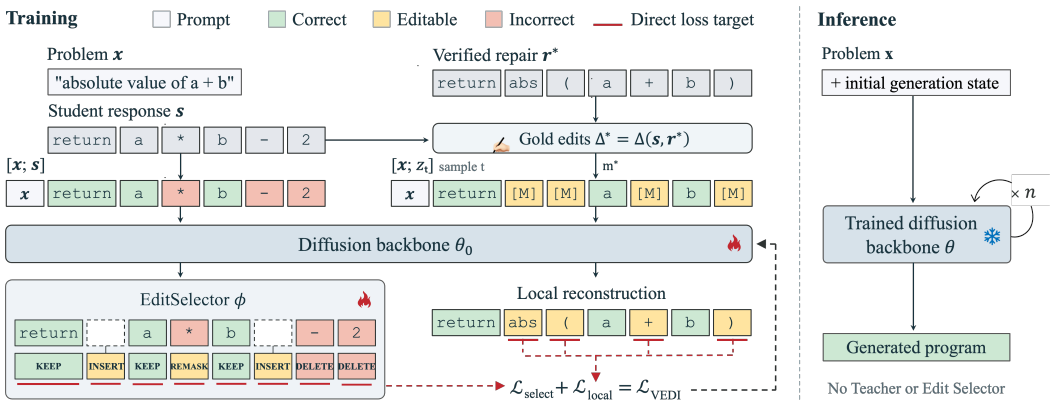


Figure 3: **VEDI training and inference.** Edit-location and local-reconstruction losses update one diffusion backbone; inference uses its original sampler.

A factorized edit-transition objective. Let $m^* = (a^*, \mathbf{1}[\ell^* > 0])$ collect the selector targets: the KEEP/REMASK/DELETE label of every token in s and the insert/no-insert decision of every gap. Given s, the selected transition $\Delta^* = (a^*, \ell^*, v^*)$ is determined by m^* together with the insertion counts ℓ^* and the contents v^*. Equivalently, given s, the pair (m^*, r^*) specifies the selected transition and admits the chain-rule factorization

$$p_{\theta,\phi}(m^*, r^* \mid x, s) = \underbrace{p_{\theta,\phi}(m^* \mid x, s)}_{\text{where to edit}} \underbrace{p_\theta(r^* \mid x, s, m^*)}_{\text{how to fill}}, \tag{2}$$

which motivates one edit-selection term and one conditional reconstruction term. The second factor uses only preserved context: once m^* is fixed, the KEEP tokens of s are given, and remasked or deleted tokens are not shown to the reconstruction pass.

Edit selection. We instantiate the first factor with an *EditSelector*, a light head with parameters ϕ on the final backbone hidden states. It predicts a three-way action for each token and a binary insertion decision for each gap. Let $\mathcal{S}$ denote the $|s|$ token decisions and $|s|+1$ gap decisions of s, and split it into edited decisions $\mathcal{S}^+$ (REMASK, DELETE, or insert) and preserved decisions $\mathcal{S}^- = \mathcal{S} \setminus \mathcal{S}^+$. The selector loss is

$$\mathcal{L}_{\text{select}}(\theta, \phi) = \sum_{i=1}^{|s|} w_i \big[-\log p_{\theta,\phi}(a_i^* \mid x, s)\big] + \sum_{j=0}^{|s|} w_j \big[-\log p_{\theta,\phi}(\mathbf{1}[\ell_j^* > 0] \mid x, s)\big], \tag{3}$$

where each decision is weighted by $1/|\mathcal{S}^+|$ if it is edited (REMASK, DELETE, or insert) and by $1/|\mathcal{S}^-|$ otherwise. This group normalization keeps the abundant KEEP labels from drowning out the edit signal. Because the selector reads the backbone hidden states, this loss updates both ϕ and θ, so edit-location supervision shapes the failed-state representation used for reconstruction as well.

Local reconstruction. Both backbones train the second factor with their own denoising objective, corrupting only the remasked and inserted positions of r^*:

$$\mathcal{L}_{\text{local}}(\theta) = \mathbb{E}_{t\sim\mathcal{U}[0,1]}\, \mathbb{E}_{z_t\sim q_t(\cdot|r^*,m^*)} \big[\mathcal{L}_t^{\text{bb}}(\theta;\, x, z_t, r^*)\big]. \tag{4}$$

During training, the reconstruction region is derived from the target m^*, rather than from the EditSelector's predictions. Here q_t leaves every KEEP token unchanged, and $\mathcal{L}_t^{\text{bb}}$ is the backbone's original loss evaluated only at corrupted positions. FlexMDM realizes local reconstruction with its native insertion and unmasking losses, whereas DreamOn applies masked-token reconstruction on a repair-length canvas. Appendix A.1 and Figure 7 give the backbone-specific objectives and length-handling details.

Training objective and inference. The training objective is

$$\mathcal{L}_{\text{VEDI}}(\theta, \phi) = \mathcal{L}_{\text{select}}(\theta, \phi) + \mathcal{L}_{\text{local}}(\theta), \tag{5}$$

a normalized training surrogate of Equation 2 with unit weights and no tunable coefficient. Accordingly, passing rollouts contribute only the preserved-decision term of $\mathcal{L}_{\text{select}}$. At inference, ϕ is discarded and the trained backbone θ runs with its original sampler.

4 Experimental Setup

4.1 Models and Data

We post-train the released 7B FlexMDM and DreamOn code models (Kim et al., 2026b; Wu et al., 2026) on failures sampled from TACO training problems (Li et al., 2023). The teacher, Qwen2.5-Coder-32B (Hui et al., 2024), proposes $K = 4$ candidate repairs per failure; its prompt is in Appendix E. Within a backbone, every arm starts from the same released checkpoint and shares the failure collection, verified-repair pool, training budget, and validation protocol. Appendix A gives collection statistics and optimization settings.

4.2 Research Questions and Comparisons

Each comparison changes one source of supervision while holding the backbone and training protocol fixed; Base is the released checkpoint. All SFT arms use each backbone's native masked-diffusion objective over the full response, with the prompt left unmasked; VEDI applies the same objective only to the aligned edit region and adds the selector loss. Section 5 addresses the first two questions with the main comparisons, and Section 6 addresses the last two with matched controls and diagnostics.

Do failure-specific repairs provide better supervision? Reference SFT trains on the reference answer, Reference ProSeCo adapts Progressive Self-Correction (Schiff et al., 2026): a second forward pass corrects a generated draft toward the same reference target. Reference Edit aligns a student failure with the reference answer.

How should verified repairs be used for post-training? Repair SFT predicts the full repair, Repair SFT + Draft also receives the failed program, and VEDI learns the aligned edit transition. VEDI-Min selects the minimum-edit verified repair, and VEDI-Random samples one uniformly.

Why does edit supervision help? Synthetic Corruption replaces the student failure with a corrupted correct program; Random-Region Reconstruction keeps the failure but moves the reconstruction region to a size-matched random position.

What does edit-structured supervision buy? We decompose each arm's pass@1 change into gained and lost tasks and measure scaling with supervised repair tokens.

4.3 Evaluation

We report pass@1 on in-domain TACO (Li et al., 2023) and on out-of-domain HumanEval(+) (Chen et al., 2021; Liu et al., 2023) and MBPP(+) (Austin et al., 2021b; Liu et al., 2023), as three-seed means for trained methods. Because the two backbones use different samplers and generation budgets, we compare methods within, not across, backbones.

5 Results

5.1 Do Failure-Specific Repairs Provide Better Supervision?

Reference answers come for free, whereas verified repairs require sampling a student failure, repairing it, and validating the result. Figure 4 asks whether this extra work provides supervision that reference answers cannot.

Under the same full-response SFT objective, Repair SFT exceeds Reference SFT on HumanEval+ on both backbones (by 4.52 and 2.80 points), although MBPP+ is mixed. Both draft-conditioned reference controls also fall short. Reference ProSeCo stays near Base on DreamOn and degrades FlexMDM by 18.14 HumanEval+ points, and Reference Edit remains more than 13 HumanEval+ points below VEDI-Min on both backbones.

Method	Student draft	Aligned edit target	Marker
Reference targets			
Reference SFT	✗	✗	●/○
Reference ProSeCo	✓	✗	●/○
Reference Edit	✓	✓	●/○
Verified-repair targets			
Repair SFT	✗	✗	★/☆
VEDI-Min	✓	✓	★/☆

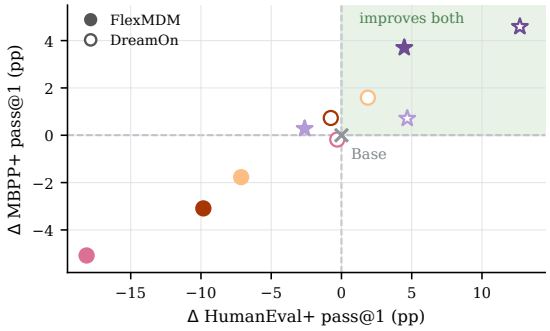


Figure 4: **Reference targets versus verified repairs.** Left: the information available to each arm. *Student draft*: a training forward pass is conditioned on a model-produced intermediate (a collected failure or a draft from an earlier pass). *Aligned edit target*: the draft–target pair is converted into token/gap edit labels and edit-region reconstruction targets. Right: change in pass@1 relative to Base; the shaded quadrant improves both HumanEval+ and MBPP+. Circles denote reference-target controls and stars verified-repair arms; filled markers show FlexMDM and open markers DreamOn.

Their shared shortfall is consistent with target distance: both pair a student draft with a reference endpoint written without seeing the failure, rather than a repair written for it. For Reference Edit's pairs, a reference answer requires 1.79 (DreamOn) and 2.52 (FlexMDM) times as many edits as the verified repair at the median under the same alignment (Appendix C.2). On FlexMDM, correcting toward such a distant target also dominates ProSeCo's loss and leaves the insertion head undertrained. A failure-specific repair therefore provides a more local target, concentrating supervision on the changes needed to turn the observed failure into a passing program. Together, these results support using a repair matched to the failure rather than a reference solution. Section 6.1 separately tests the value of grounding supervision in the student's actual failure.

5.2 How Should Verified Repairs Be Used for Post-Training?

We read Table 1 as a progression in how much of the student's failure the supervision uses. Among its arms, only the two VEDI variants improve over Base on all five suites on both backbones.

Table 1: **Code generation on two variable-length diffusion backbones.** Pass@1 (%) on five evaluation suites. Parentheses give changes from Base in percentage points.

Method	TACO	HumanEval	HumanEval+	MBPP	MBPP+
FlexMDM (7B)					
Base	1.93	50.00	46.57	64.00	54.35
Synthetic Corruption	2.63 (+0.70)	44.82 (-5.18)	41.35 (-5.22)	63.45 (-0.55)	53.90 (-0.45)
Repair SFT	2.94 (+1.01)	47.76 (-2.24)	43.95 (-2.62)	64.40 (+0.40)	54.63 (+0.28)
Repair SFT + Draft	3.41 (+1.48)	48.21 (-1.79)	44.70 (-1.87)	65.10 (+1.10)	55.51 (+1.16)
VEDI-Random	**3.70** (+1.77)	53.76 (+3.76)	50.67 (+4.10)	**68.28** (+4.28)	**58.69** (+4.34)
VEDI-Min	3.54 (+1.61)	**54.42** (+4.42)	**51.04** (+4.47)	67.50 (+3.50)	58.05 (+3.70)
DreamOn (7B)					
Base	3.11	52.44	48.17	70.37	58.99
Synthetic Corruption	3.15 (+0.04)	51.22 (-1.22)	47.92 (-0.25)	69.75 (-0.62)	59.88 (+0.89)
Repair SFT	3.94 (+0.83)	55.49 (+3.05)	52.85 (+4.68)	71.52 (+1.15)	59.70 (+0.71)
Repair SFT + Draft	3.90 (+0.79)	59.55 (+7.11)	55.84 (+7.67)	**75.57** (+5.20)	63.40 (+4.41)
VEDI-Random	**4.73** (+1.62)	62.80 (+10.36)	59.50 (+11.33)	72.57 (+2.20)	62.70 (+3.71)
VEDI-Min	4.65 (+1.54)	**64.84** (+12.40)	**60.87** (+12.70)	72.75 (+2.38)	**63.58** (+4.59)

Exposing the failure helps. Conditioning on the failed draft improves full-rewrite SFT on nine of ten backbone–benchmark means, most clearly on DreamOn (+2.99 HumanEval+). Supplying the same failure as an edit transition helps far more: VEDI-Min exceeds Repair SFT on HumanEval+ by 7.09 points on FlexMDM and 8.02 on DreamOn.

Specifying what to change helps further. With the failure available in both arms, VEDI-Min still exceeds Repair SFT + Draft on HumanEval+ by 6.34 and 5.03 points. The advantage is clearest on the HumanEval family: on DreamOn, Repair SFT + Draft is higher on MBPP and indistinguishable on MBPP+. Minimum-edit and random repair selection exchange rank across benchmarks, and the rest of the paper uses minimum edit as the default. These results suggest that the gain comes from concentrating supervision on the part of the failure that must change, rather than on the whole repair; Section 6.1 tests this with matched controls.

6 Analysis and Limitations

6.1 Why Does Edit Supervision Help?

We test three parts of this reading: whether the failure must be the student's own, whether reconstruction must occur at the aligned edit site, and whether the selector must update the shared backbone.

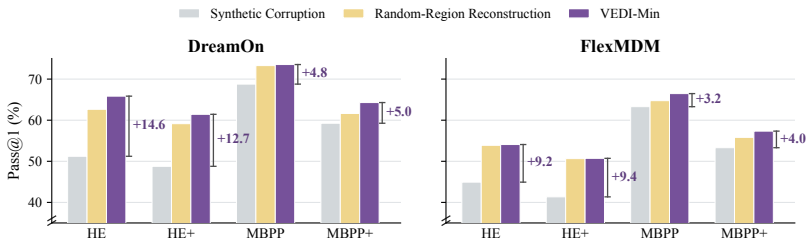


Figure 5: **Performance of failure-state and reconstruction-region controls.** Annotations report the improvement of VEDI-Min over Synthetic Corruption in percentage points.

The failure must be the student's own. Synthetic Corruption keeps VEDI's selector, edit labels, and edit-region reconstruction but derives them from a corrupted correct program instead of a real student failure. It falls below Base on HumanEval+ on both backbones and trails VEDI-Min by 9.37 points on FlexMDM and 12.65 on DreamOn. Random-Region Reconstruction, which keeps the real failure, exceeds Synthetic Corruption by 9.30 and 10.37 HumanEval+ points on FlexMDM and DreamOn, respectively. The edit format alone is therefore not what helps: the same machinery fails when the failure is artificial.

A random reconstruction region does not match VEDI. Random-Region Reconstruction keeps the real failure and its edit labels but moves the reconstruction region to a size-matched random position. This is not a fully controlled test, because the random regions still cover 69% and 64% of the true edit tokens. Even so, reconstructing a random region does not exceed VEDI on any backbone or benchmark.

Selector supervision shapes the shared backbone. At inference, the selector is gone, so its benefit must pass through the shared backbone. Detached VEDI uses the same targets but does not let the selector update the backbone. The same failure–repair pairs still train it through edit-region reconstruction. If that loss alone taught where failures occur, detaching should cost little. Instead, HumanEval+ drops by 5.22 points on FlexMDM and 2.49 on DreamOn, and edit localization also degrades. The selector loss therefore adds an edit-location signal that reconstruction alone lacks, improving generation.

Table 2: **Blocking selector gradients at the backbone.** Non-Keep AUROC falls from 0.820 to 0.706 on DreamOn and from 0.917 to 0.841 on FlexMDM.

	Standard generation		Edit localization	
Setting	HE+	MBPP+	Token edit F1	Non-Keep AUPRC
DreamOn				
Intact	59.30	63.14	0.578	0.621
Detached VEDI	56.81 (-2.49)	62.24 (-0.90)	0.464 (-0.114)	0.418 (-0.204)
FlexMDM				
Intact	49.89	57.98	0.736	0.783
Detached VEDI	44.66 (-5.22)	53.44 (-4.54)	0.599 (-0.138)	0.661 (-0.122)

6.2 What Does Edit-Structured Supervision Buy?

We next ask what VEDI trades away for its gains and how performance scales with supervised repair content.

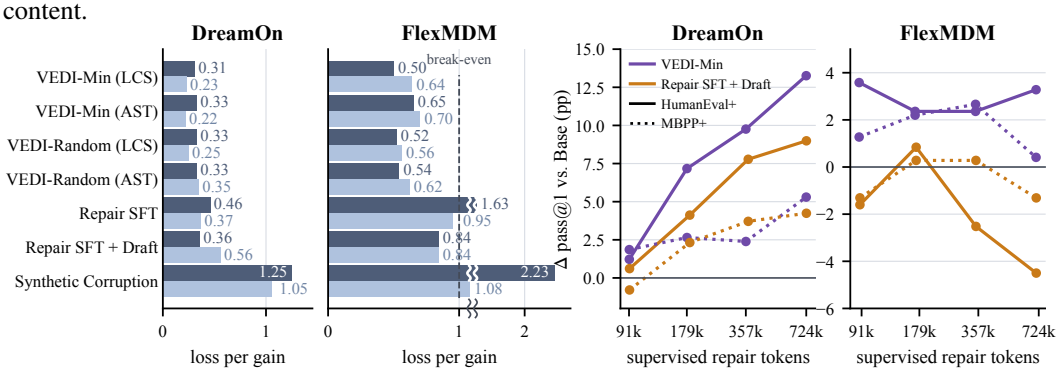


Figure 6: **Task retention and scaling with supervised repair tokens.** Left: L_w/G_w on DreamOn and FlexMDM; dark and light bars are task- and token-weighted, and the dashed line marks break-even. Right: pass@1 change from Base as supervised repair tokens increase. Purple is VEDI-Min and orange is Repair SFT + Draft; solid and dotted lines are HumanEval+ and MBPP+.

Task-level gains and losses. An average gain can hide regressions, while counting regressions alone favors arms that barely change the model. We therefore report lost mass per unit of gained mass, L_w/G_w, where $G_w = \sum_i w_i[p_i - p_i^0]_+$, $L_w = \sum_i w_i[p_i^0 - p_i]_+$, p_i^0 and p_i are the Base and post-training pass rates on task i, and w_i is 1 (task-weighted) or the task's rewritten-token count (token-weighted). Since $G_1 - L_1$ is the net pass-rate change, the ratio fixes how this change splits into gains and losses. On both backbones and under both weightings, every VEDI variant has a lower ratio than Repair SFT and Repair SFT + Draft (Figure 6, left), with the widest gap on FlexMDM (0.50 to 0.65 vs. 0.84 to 1.63, task-weighted). VEDI thus gives up less existing capability per unit of new capability.

Scaling with supervised repair tokens. At every measured budget of supervised repair tokens, VEDI stays above Base on both HumanEval+ and MBPP+ for both backbones (Figure 6, right). On DreamOn, its HumanEval+ gain grows faster than that of Repair SFT + Draft. On FlexMDM, VEDI stays positive while Repair SFT + Draft ends below Base on both suites. The advantage is thus robustness across budgets, not monotonic scaling on every benchmark.

6.3 Limitations

To our knowledge, FlexMDM and DreamOn are the only publicly released variable-length diffusion code models, so our evaluation covers these two 7B backbones and a single teacher; broader evidence across model scales, teachers, and diffusion architectures is needed to establish generality. We also study only code generation. Domains without executable feedback would need another way to validate repairs, as well as an alignment between failures and repairs suited to them. Finally, we use a fixed offline collection of failures and repairs and do not test whether refreshing it as the student changes brings further gains.

7 Conclusion

We introduced Verified Edit Distillation (VEDI), which turns execution-verified repairs of a model's own failures into supervision over what to preserve, replace, delete, and insert. Across two variable-length diffusion backbones, VEDI improves over Base on every benchmark and yields the largest average gain among the compared training methods. Controlled comparisons indicate that these gains depend on repairs matched to the student's own failures: reference targets written without seeing the failure and synthetic corruptions fall short of VEDI. Together, these results suggest that verified, failure-specific edit transitions provide a more effective post-training signal than learning only from successful endpoints. Extending this approach beyond code and to a broader range of models and teachers is an important direction for future work.

Reproducibility statement

We document model and dataset revisions, rollout and repair-generation settings, validation policies, edit-alignment conventions, and training hyperparameters in the main text and appendices. Versioned provenance artifacts record corpus, quarantine, rollout, verdict, and repair hashes. Code, configurations, and de-identified provenance records will be released with the final version.

AI use statement

Generative AI tools were used to assist with literature search, research discussion, manuscript drafting and editing, and software implementation. All AI-assisted mathematical arguments, literature claims, code, experiments, and manuscript text were reviewed by the authors; software changes were tested, and empirical claims are derived from versioned experiment artifacts. The authors take responsibility for the final content, claims, and artifacts.

References

Rishabh Agarwal, Nino Vieillard, Yongchao Zhou, Piotr Stanczyk, Sabela Ramos, Matthieu Geist, and Olivier Bachem. On-policy distillation of language models: Learning from self-generated mistakes. In *International Conference on Learning Representations*, 2024. URL https://arxiv.org/abs/2306.13649.

Jacob Austin, Daniel D. Johnson, Jonathan Ho, Daniel Tarlow, and Rianne van den Berg. Structured denoising diffusion models in discrete state-spaces. In *Advances in Neural Information Processing Systems*, volume 34, pp. 17981–17993, 2021a.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021b.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Yangruibo Ding, Marcus J. Min, Gail Kaiser, and Baishakhi Ray. CYCLE: Learning to self-refine the code generation. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA1), 2024. doi: 10.1145/3643787.

Jonas Gehring, Kunhao Zheng, Jade Copet, Vegard Mella, Taco Cohen, and Gabriel Synnaeve. RLEF: Grounding code LLMs in execution feedback with reinforcement learning. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 19034–19055, 2025. URL https://proceedings.mlr.press/v267/gehring25a.html.

Shansan Gong, Ruixiang Zhang, Huangjie Zheng, Jiatao Gu, Navdeep Jaitly, Lingpeng Kong, and Yizhe Zhang. DiffuCoder: Understanding and improving masked diffusion models for code generation. *arXiv preprint arXiv:2506.20639*, 2025. URL https://arxiv.org/abs/2506.20639.

Patrick Haluptzok, Matthew Bowers, and Adam Tauman Kalai. Language models can teach themselves to program better. In *International Conference on Learning Representations*, 2023.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, Yunlong Feng, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. Qwen2.5-Coder technical report. *arXiv preprint arXiv:2409.12186*, 2024. URL https://arxiv.org/abs/2409.12186.

BumJun Kim, Dongjae Jeon, Dueun Kim, Wonje Jeung, and Albert No. Rainbow padding: Mitigating early termination in instruction-tuned diffusion LLMs. In *International Conference on Learning Representations*, 2026a.

Jaeyeon Kim, Cheuk-Kit Lee, Carles Domingo-Enrich, Yilun Du, Sham Kakade, Timothy Ngotiaoco, Sitan Chen, and Michael Albergo. Any-order flexible length masked diffusion. In *International Conference on Learning Representations*, 2026b. URL https://arxiv.org/abs/2509.01025.

Yoon Kim and Alexander M. Rush. Sequence-level knowledge distillation. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, 2016.

Aviral Kumar, Vincent Zhuang, Rishabh Agarwal, Yi Su, John D. Co-Reyes, Avi Singh, Kate Baumli, Shariq Iqbal, Colton Bishop, Rebecca Roelofs, Lei M. Zhang, Kay McKinney, Disha Shrivastava, Cosmin Paduraru, George Tucker, Doina Precup, Feryal Behbahani, and Aleksandra Faust. Training language models to self-correct via reinforcement learning. In *International Conference on Learning Representations*, 2025. URL https://arxiv.org/abs/2409.12917.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven C. H. Hoi. CodeRL: Mastering code generation through pretrained models and deep reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 35, 2022.

Jinsong Li, Xiaoyi Dong, Yuhang Zang, Yuhang Cao, Jiaqi Wang, and Dahua Lin. Beyond fixed: Training-free variable-length denoising for diffusion large language models. In *International Conference on Learning Representations*, 2026.

Rongao Li, Jie Fu, Bo-Wen Zhang, Tao Huang, Zhihong Sun, Chen Lyu, Guang Liu, Zhi Jin, and Ge Li. TACO: Topics in algorithmic COde generation dataset. *arXiv preprint arXiv:2312.14852*, 2023. URL https://arxiv.org/abs/2312.14852.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by ChatGPT really correct? rigorous evaluation of large language models for code generation. *arXiv preprint arXiv:2305.01210*, 2023.

Ansong Ni, Srini Iyer, Dragomir Radev, Veselin Stoyanov, Wen-Tau Yih, Sida Wang, and Xi Victoria Lin. LEVER: Learning to verify language-to-code generation with execution. In *Proceedings of the 40th International Conference on Machine Learning*, pp. 26106–26128, 2023.

Shen Nie, Fengqi Zhu, Zebin You, Xiaolu Zhang, Jingyang Ou, Jun Hu, Jun Zhou, Yankai Lin, Ji-Rong Wen, and Chongxuan Li. Large language diffusion models. In *Advances in Neural Information Processing Systems*, 2025. URL https://arxiv.org/abs/2502.09992.

Ulyana Piterbarg, Lerrel Pinto, and Rob Fergus. Training language models on synthetic edit sequences improves code synthesis. In *International Conference on Learning Representations*, 2025. URL https://arxiv.org/abs/2410.02749.

Houxing Ren, Mingjie Zhan, Zimu Lu, Ke Wang, Yunqiao Yang, Haotian Hou, Junting Pan, and Hongsheng Li. Edit-based refinement for parallel masked diffusion language models. In *International Conference on Machine Learning*, 2026. URL https://arxiv.org/abs/2605.09603.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. Code Llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023. URL https://arxiv.org/abs/2308.12950.

Subham Sekhar Sahoo, Marianne Arriola, Yair Schiff, Aaron Gokaslan, Edgar Marroquin, Justin T. Chiu, Alexander Rush, and Volodymyr Kuleshov. Simple and effective masked diffusion language models. In *Advances in Neural Information Processing Systems*, 2024.

Yair Schiff, Omer Belhasin, Roy Uziel, Guanghan Wang, Marianne Arriola, Gilad Turok, Ran Zilberstein, Michael Elad, and Volodymyr Kuleshov. Learn from your mistakes: Self-correcting masked diffusion models. *arXiv preprint arXiv:2602.11590*, 2026. URL https://arxiv.org/abs/2602.11590.

Xingyao Wang, Hao Peng, Reyhaneh Jabbarvand, and Heng Ji. LETI: Learning to generate from textual interactions. In *Findings of the Association for Computational Linguistics: NAACL 2024*, pp. 223–239, 2024. doi: 10.18653/v1/2024.findings-naacl.16.

Zirui Wu, Lin Zheng, Zhihui Xie, Jiacheng Ye, Jiahui Gao, Shansan Gong, Yansong Feng, Zhenguo Li, Wei Bi, Guorui Zhou, and Lingpeng Kong. DreamOn: Diffusion language models for code infilling beyond fixed-size canvas. *arXiv preprint arXiv:2602.01326*, 2026. URL https://arxiv.org/abs/2602.01326.

Jiacheng Ye, Zhihui Xie, Lin Zheng, Jiahui Gao, Zirui Wu, Xin Jiang, Zhenguo Li, and Lingpeng Kong. Dream 7b: Diffusion large language models. *arXiv preprint arXiv:2508.15487*, 2025. URL https://arxiv.org/abs/2508.15487.

Siyan Zhao, Devaansh Gupta, Qinqing Zheng, and Aditya Grover. d1: Scaling reasoning in diffusion large language models via reinforcement learning. *arXiv preprint arXiv:2504.12216*, 2025. URL https://arxiv.org/abs/2504.12216.

Tianyu Zheng, Ge Zhang, Tianhao Shen, Xueling Liu, Bill Yuchen Lin, Jie Fu, Wenhu Chen, and Xiang Yue. OpenCodeInterpreter: Integrating code generation with execution and refinement. In *Findings of the Association for Computational Linguistics: ACL 2024*, pp. 12834–12859, 2024. doi: 10.18653/v1/2024.findings-acl.762.

Fengqi Zhu, Rongzhen Wang, Shen Nie, Xiaolu Zhang, Chunwei Wu, Jun Hu, Jun Zhou, Jianfei Chen, Yankai Lin, Ji-Rong Wen, and Chongxuan Li. LLaDA 1.5: Variance-reduced preference optimization for large language diffusion models. *arXiv preprint arXiv:2505.19223*, 2025. URL https://arxiv.org/abs/2505.19223.

A Implementation Details

Within each backbone, all arms start from the same released 7B checkpoint and differ only in the intervention specified in Section 4.2.

A.1 Backbone-Specific Edit-Region Reconstruction

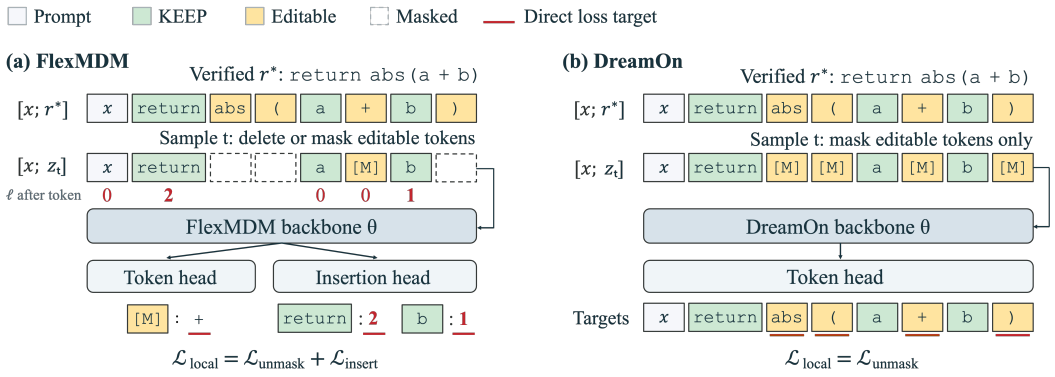


Figure 7: **Backbone-specific edit-region reconstruction.** FlexMDM reconstructs editable tokens and gaps with its token and insertion heads; DreamOn applies its token-reconstruction objective to masked edit positions on a repair-length canvas.

The local reconstruction loss of Equation 4 is instantiated with each backbone's own denoising objective (Figure 7). In both backbones, reconstruction operates on the selected repair r^*: prompt and KEEP positions are given as context and receive no loss, and only remasked and inserted positions of r^* are corrupted. Tokens labeled DELETE do not appear in r^*, so they are supervised only through the EditSelector.

FlexMDM. q_t deletes or masks editable repair tokens, and $\mathcal{L}_t^{\text{bb}} = \mathcal{L}_{\text{unmask}} + \mathcal{L}_{\text{insert}}$. The token head reconstructs masked positions, and the insertion head predicts how many deleted tokens belong at each gap of z_t.

DreamOn. q_t masks each editable position of a canvas of length $|r^*|$ with probability $t \sim \mathcal{U}(0, 1)$, and $\mathcal{L}_t^{\text{bb}} = \mathcal{L}_{\text{unmask}}$ is DreamOn's $1/t$-weighted masked cross-entropy. Because the canvas already has the repair's length, this surrogate does not supervise DreamOn's `[expand]` and `[delete]` states; at inference, the original sampler still uses them.

A.2 Failure and Repair Collection

We use the 11,255 TACO problems whose prompt fits a 768-token prompt-and-generation budget, and sample two programs per problem from each released student. FlexMDM draws both samples with the decoding settings in Table 5. DreamOn decoding at temperature 0.1 is deterministic, so we draw one program at temperature 0.1 and one at 0.3, otherwise with the settings in Table 5. For DreamOn we drop empty programs and, within a problem, programs identical to an earlier sample. The teacher, Qwen2.5-Coder-32B-Instruct, receives the prompt in Appendix E and samples $K = 4$ candidate repairs per failure with temperature 0.6, top-p 0.95, and at most 768 new tokens. A candidate is kept only if it passes all tests, and an alignment is kept only if applying the derived targets exactly reconstructs the repair. Table 3 reports the resulting counts.

Table 3: **Collected failures and verified transitions.** Coverage is over all failures in both splits. DreamOn counts are after removing empty and duplicate programs. Length changes are measured in student tokens over all verified transitions.

	FlexMDM	DreamOn
Training programs (pass / fail)	2,470 / 18,374	2,469 / 16,279
Validation programs (pass / fail)	208 / 1,458	206 / 1,295
Failures with ≥ 1 verified repair	6,938 (34.98%)	7,127 (40.55%)
Training / validation transitions	6,432 / 506	6,600 / 527
Training rows (repairs + identity)	8,902	9,069
Transitions that change length	96.7%	97.5%

A.3 Training and Evaluation

Table 4: **Optimization settings shared across arms and backbones.** The EditSelector learning rate applies only to VEDI arms.

Setting	Value
Optimizer	AdamW
(β_1, β_2)	$(0.9, 0.95)$
Weight decay	0.01
Learning-rate schedule	10% linear warmup, then constant
Backbone learning rate	1×10^{-5}
EditSelector learning rate	1×10^{-4}
Micro-batch size $\times$ gradient accumulation	1×16
Parameter updates	1,114
Maximum sequence length	1,024
Numerical precision	bfloat16
Checkpoint interval	200 updates
Training seeds	42, 231, 10234

Table 4 lists the optimization settings, which are shared across arms and backbones. Every arm is trained for 1,114 updates of 16 examples each, about two epochs over the verified repairs and identity targets of its backbone (2.03 epochs on FlexMDM and 1.97 on DreamOn after dropping the 113 and 27 rows that exceed 1,024 tokens). We save a checkpoint every 200 updates and at the final update. The checkpoint is selected on two frozen subsets of the TACO validation split: the 64 problems with the smallest task identifiers, and the first 32 problems that the backbone's BASE solves at least once. We choose the checkpoint with the most passing programs, breaking ties by fewer problems that BASE solves but the candidate does not. Table 5 lists decoding and evaluation settings.

Table 5: **Generation and evaluation configuration.** EvalPlus reuses the corresponding HumanEval or MBPP generations. DreamOn's response length is further capped so that it never exceeds FlexMDM's response slots under the same sequence budget.

Decoding	FlexMDM	DreamOn
Temperature	0.1	0.1, top-p 0.9
Denoising steps	512	One token per step until no mask remains
Length control	Insertion head, power $\bar{a} = 2.9$	`[expand]` / `[delete]`
Initial mask length	–	128
Maximum response length	Sequence budget	256 (128 + 128 expand budget)

Evaluation	TACO	HumanEval	MBPP
Evaluated problems	803	Full benchmark	Full benchmark
Samples per problem (FlexMDM / DreamOn)	4 / 4	8 / 4	4 / 4
Sequence budget (prompt + response)	768	768	1,100
EvalPlus test suite	–	HumanEval+	MBPP+

A.4 Baselines and Controls

Reference SFT and Repair SFT. The backbone's masked-diffusion objective is applied to the full reference or repair, with the prompt left unmasked. The reference is the first reference solution, in dataset order, that passes the problem's tests.

Repair SFT + Draft. As Repair SFT, but the failed program is inserted into the prompt with the draft-conditioning template in Appendix E.

Reference ProSeCo. Following Schiff et al. (2026), each update adds a second forward pass with tied weights. The first pass's argmax predictions, with `[MASK]` excluded, replace every response token of the noised input, and the second pass is trained toward the reference. The correction loss is added to the native loss with weight 1, and the hand-off between the passes is detached. The corrector's input is thus the model's own prediction at noise level t, not a completed student draft. On FlexMDM, whose forward process deletes positions, the correction covers only retained positions; deleted positions are supervised by the insertion head alone.

Reference Edit. The student failure is aligned with the reference instead of the repair, and the VEDI objective is applied unchanged.

Synthetic Corruption. Each verified repair is corrupted by FlexMDM's forward process q_t, which deletes some tokens and masks others, and the corrupted program replaces the student failure. The noise level t is set per example by bisection so that the corruption's normalized edit ratio matches that of the real failure (median absolute gap 0.004 on FlexMDM and 0.006 on DreamOn). Edit targets are read off the corruption, so they contain KEEP, REMASK, and INSERT but never DELETE; the selector and local reconstruction are unchanged.

Random-Region Reconstruction. The selector targets are unchanged, but the reconstruction region is replaced by random spans of the repair with the same number and lengths as the true edit spans. The random regions still cover 69% (DreamOn) and 64% (FlexMDM) of the true edit tokens.

Detached VEDI. A stop-gradient is placed between the failed-state backbone representation and the EditSelector; the selector and reconstruction paths are otherwise unchanged.

B Additional Results

B.1 Minimum-Edit vs. Random Repair Selection

VEDI-Min and VEDI-Random exchange rank across backbones and benchmarks in Table 1. VEDI-Min is higher on six of the ten backbone–benchmark pairs, including the HumanEval family for both backbones, whereas VEDI-Random is higher on the remaining four, including FlexMDM TACO and the FlexMDM MBPP family. Minimum-edit selection alone therefore does not explain VEDI's gains; we use it as a deterministic default rather than as a universally better selection rule.

The two rules nonetheless produce different training targets. Across the 6,432 repaired training failures, a uniform draw selects a repair whose text differs from the minimum-edit repair on 31.2–32.0% of examples across the three seeds (31.7% on average). Relative to VEDI-Min, the random targets require 3.35–3.58% more reconstruction tokens. The comparison thus changes about one third of the repair texts while only slightly increasing the total amount reconstructed, so its mixed ranking is not explained by Random receiving a much larger supervision budget.

B.2 AST vs. LCS Alignment

How the alignments differ. Our AST alignment is an AST–token hybrid rather than a tree-edit-distance objective. When both programs parse as Python, it partitions each into statements, compound headers, and the filler spans between them. It first matches spans with formatting-insensitive AST keys and then performs a token diff only within the matched or replaced statement blocks. This structural seeding prevents common tokens such as indentation, parentheses, or `return` from being matched across unrelated statements. If either program does not parse, it falls back to a whole-sequence token diff.

Exact LCS instead ignores syntax and computes a deterministic longest common subsequence over student-tokenizer IDs with Hunt–Szymanski alignment. Tokens on the subsequence become Keep anchors; unmatched intervals become Remask, Delete, or insertion targets according to their relative lengths. LCS therefore maximizes the number of preserved tokens and also handles syntactically invalid drafts, whereas the AST hybrid favors statement-local correspondence and can preserve fewer tokens than the global optimum.

Table 6: **AST-hybrid versus exact-LCS alignment.** Three-seed mean pass@1 (%). Bold marks the better alignment under the same backbone and selection rule.

Alignment	Selection	TACO	HE	HE+	MBPP	MBPP+
DreamOn (7B)						
AST	Min	3.90	60.92	58.18	**73.10**	61.20
LCS	Min	**4.65**	**64.84**	**60.87**	72.75	**63.58**
AST	Random	4.57	61.94	58.64	**72.75**	61.02
LCS	Random	**4.73**	**62.80**	**59.50**	72.57	**62.70**
FlexMDM (7B)						
AST	Min	**3.77**	53.05	49.89	66.75	57.98
LCS	Min	3.54	**54.42**	**51.04**	**67.50**	**58.05**
AST	Random	3.51	53.30	49.77	67.37	58.16
LCS	Random	**3.70**	**53.76**	**50.67**	**68.28**	**58.69**

LCS outperforms the AST hybrid in 17 of the 20 backbone, selection-rule, and benchmark combinations in Table 6. The three exceptions are DreamOn MBPP under both rules and FlexMDM TACO under minimum-edit selection, each by less than 0.4 points. We therefore use LCS in all main experiments. The choice of alignment does not change the conclusions that VEDI improves over Base or that the two selection rules exchange rank across benchmarks.

B.3 Sharpening versus Reach

A higher pass@1 can come from solving new problems or from concentrating probability on programs the base model could already sample. We separate the two on FlexMDM with pass@k up to $k = 8$

on HumanEval and $k = 4$ on MBPP, using the unbiased estimator of Chen et al. (2021) (Table 7). We omit DreamOn, whose samples at temperature 0.1 are nearly identical, so its pass@k equals pass@1.

Table 7: **Pass@k on FlexMDM HumanEval and MBPP.** Three-seed mean (%). Trained arms draw 8 samples per HumanEval problem and 4 per MBPP problem

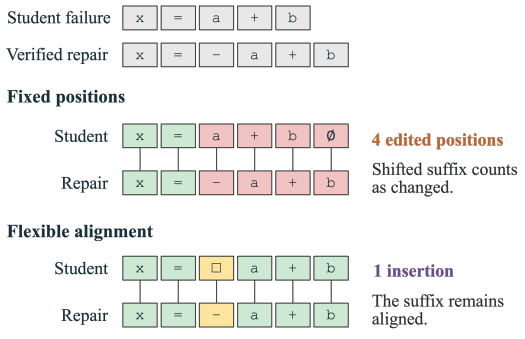
	HumanEval				MBPP		
Arm	k=1	k=2	k=4	k=8	k=1	k=2	k=4
Base	50.00	65.21	76.93	85.27	64.00	76.12	83.12
Synthetic Corruption	44.82	59.23	71.06	79.27	63.45	74.44	81.13
Repair SFT	47.76	62.86	74.73	83.27	64.40	76.28	82.98
Repair SFT + Draft	48.21	63.65	75.44	83.31	65.10	77.00	**83.69**
VEDI-Min (AST)	53.05	67.82	78.31	85.04	66.75	77.63	83.33
VEDI-Min (LCS)	**54.42**	**69.19**	**79.42**	**85.90**	**67.50**	**77.85**	83.25

VEDI-Min raises HumanEval pass@1 by 4.42 points, and its lead narrows as k grows ($+3.98$, $+2.49$, and $+0.63$ at $k = 2, 4, 8$), leaving pass@8 close to Base. Full-rewrite training and Synthetic Corruption instead stay below Base at $k = 8$ (Repair SFT -2.00, Repair SFT + Draft -1.96, Synthetic Corruption -6.00): they no longer solve, within eight samples, some problems the base model solves. On MBPP, VEDI-Min's lead narrows the same way ($+3.50$, $+1.73$, and $+0.13$ at $k = 1, 2, 4$). Synthetic Corruption again ends below Base (-1.99 at $k = 4$), whereas the two full-rewrite arms end within 0.6 points of it (-0.14 and $+0.57$). VEDI therefore concentrates probability on solutions the base model can already reach without losing them, whereas synthetic failures lose some on both benchmarks and full-rewrite training loses some on HumanEval. Because the base model already reaches most HumanEval problems within eight samples, this benchmark cannot test whether VEDI adds new capability.

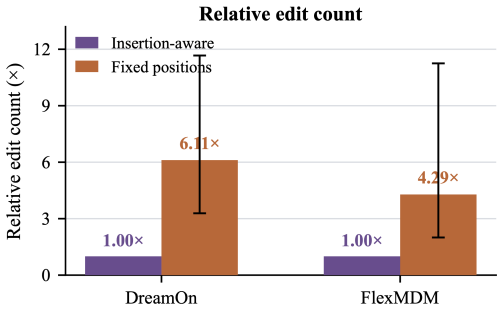
C Additional Analysis

C.1 Insertion-Aware vs. Fixed-Position Alignment

Figure 8 illustrates why an insertion-aware alignment is necessary. A local insertion shifts every later token under a fixed positional comparison, even when the suffix should be preserved (Figure 8a). Across 6,938 verified FlexMDM transitions, fixed positions count 4.29 times as many edits at the median (IQR 2.00 to 11.25). The effect is larger across 6,600 DreamOn transitions: 6.11 times at the median (IQR 3.29 to 11.67; Figure 8b).



(a) Aligning one repair



(b) Edit count across verified transitions

Figure 8: **Flexible geometry keeps local repairs local.** (a) A fixed positional comparison counts the shifted suffix as rewritten, whereas a flexible representation expresses the same transformation as a single insertion and keeps the suffix aligned. (b) Edit count under fixed positions relative to insertion-aware alignment. Bars report medians and error bars span the 25th–75th percentiles over 6,600 DreamOn and 6,938 FlexMDM verified transitions.

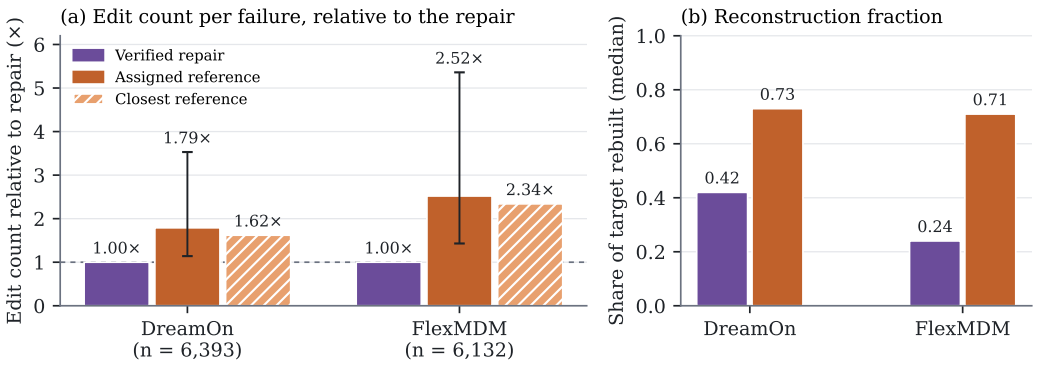


Figure 9: **Reference answers require larger edits than verified repairs.** Exact token LCS, paired by failure, passing rollouts excluded. (a) Median edit cost C relative to the verified repair; error bars span the 25th to 75th percentiles. *Assigned reference* is the solution the Reference Edit arm trained on; *closest reference* is the cheapest of the problem's three solutions. (b) Median share of the target that must be reconstructed.

C.2 Edit Size of Repair and Reference Targets

Reference Edit and VEDI-Min train on the same student failures but align them to different targets. We measure how large an edit each target requires with the exact token LCS used by the main arms and the edit cost C of Equation 1, paired by failure. Passing rollouts are excluded, since their repair equals the draft.

A reference answer needs a larger edit than the verified repair on 81.8% of 6,393 paired failures on DreamOn and 88.3% of 6,132 on FlexMDM (Figure 9a). The median ratio is 1.79 on DreamOn and 2.52 on FlexMDM. The result does not depend on which reference the arm trained on: using the closest of each problem's three reference solutions lowers the median only to 1.62 and 2.34. In the unit the objective trains on, a repair asks the model to rebuild 0.42 (DreamOn) and 0.24 (FlexMDM) of the target, compared with 0.73 and 0.71 for a reference (Figure 9b).

The spread matters as much as the median. The lower quartile of the ratio is 1.14 on DreamOn and 1.43 on FlexMDM, so on about a quarter of failures the two targets require nearly the same edit. The size of the ratio also depends on the alignment: an AST-based alignment finds more shared structure and gives larger ratios, while the ordering is the same under both.

D Qualitative Examples

Training transitions. Figure 10 shows student failures from the TACO training split with their verified teacher repair and the dataset reference that Reference Edit aligns to instead. For each backbone, one case is a one-token slip and one is a logic fix. On DreamOn, the first draft writes `in(n)` for `in range(n)`; the repair adds `range`, whereas the reference rebuilds the program around a set. The second draft halves both nodes on every step, so they never meet at different depths; the repair adds the missing `if a > b` branch at one site, whereas the reference reimplements the loop with parity cases. On FlexMDM, the first draft tests `__main__` instead of `__name__`, so `main()` never runs; the repair changes one identifier in a 24-line draft. The second draft iterates over rows as if they were columns and never resets `started`; the repair fixes both sites, whereas the reference is a different program built on `groupby`. Although these cases are selected for legibility rather than representativeness, they reflect the population-level pattern in Appendix C.2: the repair keeps the draft's structure, whereas the reference rewrites most of it.

Generations after training. Figure 11 shows HumanEval programs that Base, Repair SFT + Draft, and VEDI-Min generate from the prompt alone. For each backbone we show one problem that both trained models solve and one that only VEDI-Min solves, chosen for a VEDI-Min answer that stays close to the Base program. In the first case, Base checks whether the array is already sorted and ignores rotation; VEDI-Min keeps its guard and loop and replaces only the test inside

the loop, whereas REPAIR SFT + DRAFT writes a new body around a rotated slice. In the second, BASE seeds the running maximum with `numbers[0]`, which fails on an empty list; VEDI-MIN changes that one initializer to `float('-inf')`, whereas REPAIR SFT + DRAFT reproduces the BASE program token for token, bug included.

On FlexMDM (seed 10234, 8 samples), samples are picked by a fixed rule: a failing BASE sample, then the passing VEDI-MIN sample closest to it, then the REPAIR SFT + DRAFT sample closest to it. In the first case, the BASE sample reverses the no-letter string twice and appends lowercase letters in both branches; VEDI-MIN keeps its structure and fixes both lines, whereas the passing REPAIR SFT + DRAFT sample is a different program built on a `for...else`. Over 8 samples, however, REPAIR SFT + DRAFT passes 3, the same as BASE, and VEDI-MIN passes 8. In the second, the BASE sample stops counting digits after the first one; VEDI-MIN keeps the checks in order and replaces the counting loop with a `sum`, whereas REPAIR SFT + DRAFT reorders the checks and adds `name.isalpha()`, which wrongly rejects names containing digits. VEDI-MIN and BASE both pass 5 of 8 samples and REPAIR SFT + DRAFT passes none, so this gain is over REPAIR SFT + DRAFT only.

DREAMON

Report whether each name has been seen before `taco/train/04/002565` runtime_error · 0/12 tests

Student failure

fails · 67 tokens

```
n = int(input())
s = [input() for _ in(n)]
ans = []
d = {}
for i in range(n):
    if s[i] in d:
        ans.append("YES")
    else:
        ans.append("NO")
        d[s[i]] = 1
print("\n".join(ans))
```

Verified repair

passes · 2 tokens edited

```
n = int(input())
s = [input() for _ in range(n)]
ans = []
d = {}
for i in range(n):
    if s[i] in d:
        ans.append("YES")
    else:
        ans.append("NO")
        d[s[i]] = 1
print("\n".join(ans))
```

Reference

passes · 65 edited

```
def main():
    N = int(input())
    memo = set()
    for _ in range(N):
        s = input()
        print('YES' if s in memo else
        'NO')
        memo.add(s)
main()
```

Lowest common ancestor in an implicit binary tree `taco/train/04/002356` wrong_answer · 0/1 tests

Student failure

fails · 50 tokens

```
t = int(input())
for _ in range(t):
    a, b = map(int, input().split())
    while a != b:
        a = a // 2
        b = b // 2
    print(a)
```

Verified repair

passes · 10 tokens edited

```
t = int(input())
for _ in range(t):
    a, b = map(int, input().split())
    while a != b:
        if a > b:
            a = a // 2
        else:
            b = b // 2
    print(a)
```

Reference

passes · 114 edited

```
test = int(input())
for i in range(test):
    a , b = [int(x) for x in
    input().split()]
    p1 = a
    p2 = b
    while p1 != p2:
        if p1 > p2:
            if p1 % 2 == 0:
                p1 /= 2
            else:
                p1 = ( p1  - 1) / 2
        elif p2 > p1:
            if p2 % 2 == 0:
                p2 /= 2
            else:
                p2 = ( p2  - 1) / 2
    print(p1)
```

FLEXMDM

Count distinct x and y coordinates `taco/train/07/000810` runtime_error · 0/1 tests

Student failure

fails · 133 tokens

```
import sys
def main():
    input = sys.stdin.read().split()
    idx = 0
    T = int(input[idx])
    idx += 1
    for _ in range(T):
        N = int(input[idx])
        idx += 1
        xs = set()
        ys = set()
        for _ in range(N):
            x = int(input[idx])
            idx += 1
            y = int(input[idx])
            idx += 1
            xs.add(x)
            ys.add(y)
        total = len(xs) + len(ys)
        print(total)
if __main__ == "__main__":
    main()
```

Verified repair

passes · 2 tokens edited

```
import sys
def main():
    input = sys.stdin.read().split()
    idx = 0
    T = int(input[idx])
    idx += 1
    for _ in range(T):
        N = int(input[idx])
        idx += 1
        xs = set()
        ys = set()
        for _ in range(N):
            x = int(input[idx])
            idx += 1
            y = int(input[idx])
            idx += 1
            xs.add(x)
            ys.add(y)
        total = len(xs) + len(ys)
        print(total)
if __name__ == "__main__":
    main()
```

Reference

passes · 118 edited

```
n = int(input())
for i in range(n):
    k = int(input())
    l1 = []
    l2 = []
    for i in range(k):
        (a, b) = map(int,
        input().split())
        l1.append(a)
        l2.append(b)
    print(len(set(l1)) + len(set(l2)))
```

Count the disconnected segments of a wall `taco/train/04/000961` wrong_answer · 0/12 tests

Student failure

fails · 65 tokens

```
R, C = map(int, input().split())
columns = [input().strip() for _ in
range(R)]
count = 0
started = False
for col in columns:
    if 'B' in col:
        if not started:
            count += 1
            started = True
print(count)
```

Verified repair

passes · 13 tokens edited

```
R, C = map(int, input().split())
columns = [input().strip() for _ in
range(R)]
count = 0
started = False
for col in zip(*columns):
    if 'B' in col:
        if not started:
            count += 1
            started = True
    else:
        started = False
print(count)
```

Reference

passes · 126 edited

```
mod = 1000000007
ii = lambda : int(input())
si = lambda : input()
dgl = lambda : list(map(int, input()))
f = lambda : map(int, input().split())
il = lambda : list(map(int,
input().split()))
ls = lambda : list(input())
from itertools import groupby
(n, m) = f()
cnt = 0
x = ''
for i in range(n):
    x = si()
c = 0
for (i, j) in groupby(x):
    c += i == 'B'
print(c)
```

Figure 10: **Training transitions: one student failure, two candidate targets.** Red marks the draft lines the teacher changed, green the teacher's version of them, and gray the dataset reference. Edit counts are exact token-LCS costs (changed draft tokens plus inserted tokens).

DREAMON

Can one rotation sort the array? `HumanEval/109` both trained samples pass

Base

0/4 pass · sample 0 fails

```
def move_one_ball(arr):
    if not arr:
        return True
    for i in range(1, len(arr)):
        if arr[i] < arr[i - 1]:
            return False
    return True
```

Repair SFT + Draft

4/4 pass · sample 0 passes · sim. to Base 0.40

```
    def move_one_ball(arr):
        if not arr:
            return True
        n = len(arr)
        for i in range(n):
            rotated = arr[-i:] +
            arr[:-i]
            if all((x <= y for (x, y)
            in zip(rotated,
            rotated[1:]))):
                return True
        return False
```

VEDI-Min

4/4 pass · sample 0 passes · sim. to Base 0.80

```
def move_one_ball(arr):
    if not arr:
        return True
    for i in range(len(arr)):
        if arr[i:] + arr[:i] ==
        sorted(arr):
            return True
    return False
```

Rolling maximum `HumanEval/9` only the VEDI-Min sample passes

Base

0/4 pass · sample 0 fails

```
def rolling_max(numbers: List[int]) ->
List[int]:
    result = []
    current_max = numbers[0]
    for num in numbers:
        current_max = max(current_max,
        num)
        result.append(current_max)
    return result
```

Repair SFT + Draft

0/4 pass · sample 0 fails · sim. to Base 1.00

```
def rolling_max(numbers: List[int]) ->
List[int]:
    result = []
    current_max = numbers[0]
    for num in numbers:
        current_max = max(current_max,
        num)
        result.append(current_max)
    return result
```

VEDI-Min

4/4 pass · sample 0 passes · sim. to Base 0.89

```
def rolling_max(numbers: List[int]) ->
List[int]:
    result = []
    current_max = float('-inf')
    for num in numbers:
        current_max = max(current_max,
        num)
        result.append(current_max)
    return result
```

FLEXMDM

Swap letter case, or reverse a string with no letters `HumanEval/161` both trained samples pass

Base

3/8 pass · sample 5 fails

```
def solve(s):
    if not any((char.isalpha() for
    char in s)):
        return s[::-1][::-1]
    result = []
    for char in s:
        if char.islower():
            result.append(char.upper())
            result.append(char.lower())
        elif char.isupper():
            result.append(char.lower())
        else:
            result.append(char)
    return ''.join(result)
```

Repair SFT + Draft

3/8 pass · sample 6 passes · sim. to Base 0.34

```
def solve(s):
    for char in s:
        if char.isalpha():
            return
            ''.join((char.upper() if
            char.islower() else
            char.lower() for char in
            s))
    else:
        return s[::-1]
```

VEDI-Min

8/8 pass · sample 4 passes · sim. to Base 0.85

```
def solve(s):
    if not any((char.isalpha() for
    char in s)):
        return s[::-1]
    result = []
    for char in s:
        if char.isalpha():
            if char.islower():
                result.append(char.up
                per())
            else:
                result.append(char.lo
                wer())
        else:
            result.append(char)
    return ''.join(result)
```

Validate a file name `HumanEval/141` only the VEDI-Min sample passes

Base

5/8 pass · sample 3 fails

```
def file_name_check(file_name):
    if file_name.count('.') != 1:
        return 'No'
    (name, extension) =
    file_name.split('.')
    if not name or not
    name[0].isalpha():
        return 'No'
    digit_count = 0
    for char in name:
        if char.isdigit():
            digit_count += 1
            break
    if digit_count > 3:
        return 'No'
    if extension not in ['txt', 'exe',
    'dll']:
        return 'No'
    return 'Yes'
```

Repair SFT + Draft

0/8 pass · sample 8 fails · sim. to Base 0.69

```
def file_name_check(file_name):
    if sum((char.isdigit() for char in
    file_name)) > 3:
        return 'No'
    if file_name.count('.') != 1:
        return 'No'
    (name, extension) =
    file_name.split('.')
    if not name or not extension:
        return 'No'
    if not name.isalpha():
        return 'No'
    if extension in ['txt', 'exe',
    'dll']:
        return 'Yes'
    return 'No'
```

VEDI-Min

5/8 pass · sample 7 passes · sim. to Base 0.89

```
def file_name_check(file_name):
    if file_name.count('.') != 1:
        return 'No'
    (name, extension) =
    file_name.split('.')
    if not name:
        return 'No'
    if not name[0].isalpha():
        return 'No'
    digit_count = sum((1 for char in
    name if char.isdigit()))
    if digit_count > 3:
        return 'No'
    if extension not in ['txt', 'exe',
    'exe', 'dll']:
        return 'No'
    return 'Yes'
```

Figure 11: **HumanEval generations after training**, at the checkpoints of Table 1. Green marks what VEDI-Min changed relative to Base, red the Base lines it replaced, and amber what Repair SFT + Draft wrote differently from Base. Headers give the pass count over all samples, the sample shown, and its similarity to the Base sample shown (token-level matching ratio; 1 means identical).

E PROMPT TEMPLATES

The following templates are taken from the code used for data collection and training. Braced fields are filled per example, and bracketed blocks are included only when the corresponding evidence is available. The repair teacher sees execution feedback; no execution feedback enters a student training input.

E.1 REPAIR-TEACHER PROMPTS

System message

```
You are an expert Python engineer. You will be shown
a problem statement, a candidate solution,
and evidence of how it fails.

Your goal, in priority order:
1. Produce a COMPLETE program that correctly solves the stated
    problem.
   This comes first. A short or partial answer is a failure
   even if it is close to the candidate.
2. Keep the parts of the candidate that are correct and useful.
3. Among programs that are fully correct,
   prefer the one that changes less.

If the candidate is incomplete, truncated, or structurally unusable,
replace large spans or rewrite it entirely -- that is expected
and correct.
Do not preserve a broken structure, a broken approach, or misleading
variable names just because the candidate had them.

Read the whole problem statement and handle the input and
output format it specifies, including printing the result.

Do not add comments explaining the fix. Reply with the complete
program in a single ```python code block, and nothing else.
```

User message

````
# Problem
{problem_statement}

# Candidate solution
```python
{student_draft}
```

# Failure evidence
Result: {outcome}. Passed {n_tests_passed} of {n_tests_total} tests.
[The entry point must remain a function named `{fn_name}`
 that returns its answer; do not read stdin.]

[Traceback:
```
{traceback}
```]

[Failing case {index}:
  input:
````

```
{stdin}
  expected output:
{expected}
  actual output:
{actual}]

# Task
Return the complete corrected program in one ```python block. Make
    the
smallest change that fixes the failure.
```

The problem statement is middle-truncated at 6,000 characters, each failing case field at 1,500 characters, and the traceback at 1,200 characters. The template supports up to three distinct failing cases.

E.2 Draft-Conditioning Prompt

The canonical student prompt ends in an open ```` ```python ```` fence. That fence is removed and the following block is appended. Unlike the repair teacher prompt, this training prompt contains no execution feedback.

Draft-conditioning prompt

````
{canonical_student_prompt_without_trailing_fence}

# Candidate solution
```python
{student_draft}
```

# Task
Revise the candidate solution into a complete correct program
that solves the problem above. If the candidate is already correct,
reproduce it.
Write only the program.

```python
````
````