TMT: CO-EVOLVING MODELS AND HARNESSES WITH LOCAL TEST FEEDBACK

Anonymous authors
Paper under double-blind review

ABSTRACT

The performance of a coding agent depends on both the model and the harness. A complete task rollout may succeed despite ineffective actions, or fail despite correctly implemented subtasks. We propose Task–Method–Test (TMT), a framework that uses test feedback to improve both components. TMT splits each original task into subtasks and, following test-driven development (TDD), generates corresponding tests for them. The tests are admitted and fixed before implementation. Each subtask is then addressed by one or more methods. Each method comprises one or more model responses and their associated tool calls. During rollouts on training tasks, TMT records these methods and their test outcomes. Records whose tests pass are used for model fine-tuning, including records from rollouts that fail whole-task evaluation. Records whose tests fail guide harness adaptation through aggregation, diagnosis, and targeted repair, even when their rollouts pass whole-task evaluation. We evaluate the first TMT update cycle, in which the model and harness are each updated once, using a Model $\times$ Harness matrix on SWE-bench to measure the individual and combined effects of these updates. We then assess the updated system on eight zero-shot coding benchmarks. The updated model–harness pair outperforms the vanilla system and the systems with only a model update or only a harness update in whole-task pass rate, while using fewer rollout tokens. The updated harness generalizes well to frozen external models.

1 INTRODUCTION

Coding agents solve software tasks by reading code, making edits, and using execution feedback. Their performance depends on both the model and the harness that provides context, tools, and control over execution. SWE-Gym and SWE-smith show how executable tasks can supply data for model training (Pan et al., 2025; Yang et al., 2025). SWE-agent shows that the interface between the model and its environment also affects task success (Yang et al., 2024). Improving a coding agent therefore calls for attention to both components.

Recent work has begun to improve these two components jointly. HarnessForge updates the harness and policy together, while HELIX links harness changes to verified rollouts for later model learning (Chen et al., 2026; Fan & Huang, 2026). Self-Harness shows that an agent can improve its own harness without a stronger external agent (Zhang et al., 2026a). Together, these directions motivate co-evolving the model and the harness, so that improvements to one component can inform improvements to the other.

These joint updates depend on data that can guide both model training and harness changes. However, task data is often organized as complete rollouts, which can mix correct code changes, failed attempts, and later repairs. For example, an agent may correctly repair input validation but fail to handle a separate parsing case. Treating the whole rollout as one training example is too coarse: discarding a failed rollout loses its successful work, while training on every action from a successful rollout can retain failed or redundant actions. The final task result also does not reveal whether a failure came from the model or the harness. These limitations call for a fine-grained record that links the work performed on each subtask to its test result.

To address this data limitation, we introduce **Task–Method–Test (TMT)**, which links subtasks, methods, and tests. In the first cycle, an external teacher assists with decomposing each task into

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.

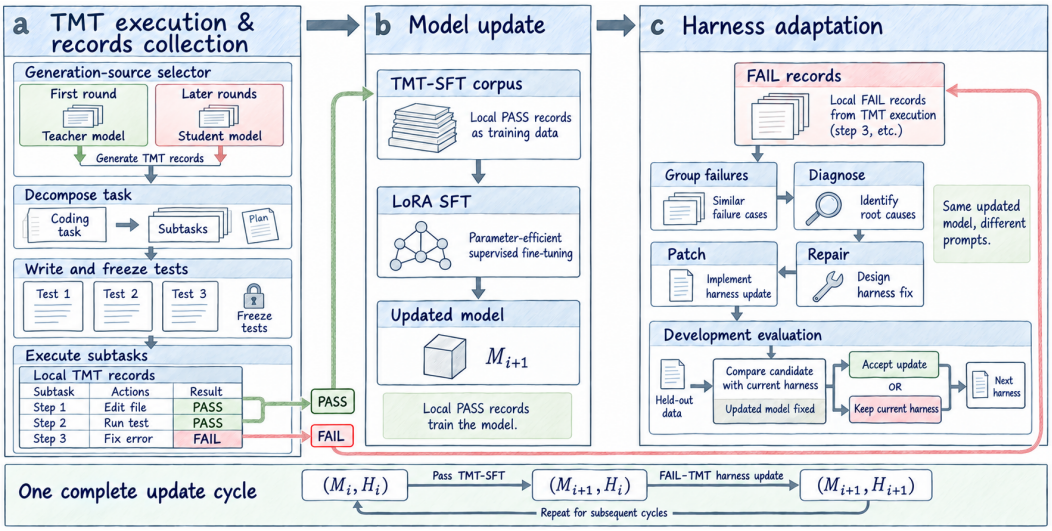

Figure 1: **TMT model–harness co-evolution.** (a) Decompose tasks, admit and freeze tests, and collect local records. (b) Select actions from PASS records for LoRA SFT. (c) The same updated model groups failures, diagnoses defects, and proposes harness patches. The development stage compares harnesses with the model fixed. It uses development data, not the final test set. The loop shows the general update cycle; experiments cover its first cycle.

subtasks and generating corresponding tests before implementation, following test-driven development (TDD); the tests are admitted and fixed before the coding agent begins implementation. In subsequent cycles, the updated model performs task decomposition and test generation itself. The agent addresses each subtask through one or more methods. Each method comprises one or more model responses and their associated tool calls. Running an admitted test creates a record linking the subtask and its methods to the tested code state and test result. PASS records support model fine-tuning, while FAIL records guide harness diagnosis and repair. These local records remain useful even when their results differ from the whole-task outcome.

Figure 1 illustrates the TMT update cycle: collecting TMT records, updating the model, and then adapting the harness. In this paper, we study only the first cycle. DeepSeek-V4-Flash-0731 serves as the teacher for cold-start TMT data collection on SWE-smith tasks, and Qwen3.6-27B is the student model. After fine-tuning on the collected records, the student uses its own harness to aggregate failures, diagnose defects, and propose repairs; the teacher is not involved in these steps. We use a Model $\times$ Harness matrix on SWE-bench to evaluate the individual and joint updates, and evaluate the updated model–harness pair on eight zero-shot coding benchmarks. We also compare TMT-SFT with SFT trained on complete task trajectories (Raw-SFT) and test whether the updated harness generalizes well to frozen external models.

Our contributions are threefold:

- **TMT co-evolution.** We design a training framework that uses TMT records to improve both model and harness. These records provide finer-grained feedback than complete task rollouts.
- **Efficient model training.** TMT selects locally verified actions for training and excludes failed actions from training targets. It exceeds Raw-SFT's SWE-bench pass rate and matches its overall zero-shot pass rate with less training data and lower mean rollout token use.
- **Harness improvement and generalization.** Our 27B model uses its own harness to group failures, find defects, and propose fixes, without an external meta-agent. The updated harness also improves task success with frozen external models.

Organization of this paper. Section 2 reviews model–harness co-evolution, training data, and harness self-improvement. Section 3 presents TMT record construction and the model and harness updates. Section 4.1 describes the experimental setup. Section 4.2 examines joint and individual updates, Section 4.3 compares TMT-SFT with SFT trained on complete task trajectories, and Section 4.4 evaluates harness generalization across frozen external models. Section 5 concludes the paper, and Section 6 states its limitations. The appendix provides the first-cycle update procedure, test contracts, prompts, admission examples, and detailed experimental settings.

2 Related Work

Related work relevant to TMT falls into three strands: model–harness co-evolution, test- and execution-based supervision for coding agents, and harness self-improvement. We review these strands in turn and then position TMT as linking local test outcomes to both model training and harness repair.

Agent systems can improve through changes to their workflows, harnesses, and model weights. AFlow searches over executable workflows, while the Darwin Gödel Machine edits agent code and evaluates the resulting agents (Zhang et al., 2025a; 2026b). Moving toward joint updates, Harness-Forge updates the harness and policy together, while HELIX preserves harness changes and verified sibling rollouts for later model learning (Chen et al., 2026; Fan & Huang, 2026). These studies raise a data question: what level of detail is needed to guide both model training and harness repair within a task rollout?

Work on training data addresses part of this question. SWE-Gym provides executable software engineering tasks, while SWE-smith generates training tasks at scale (Pan et al., 2025; Yang et al., 2025). Test-driven development places tests before implementation (Beck, 2002), and SWE-Flow uses unit tests and runtime dependencies to construct verifiable incremental tasks (Zhang et al., 2025b). Process supervision shows the value of feedback on intermediate steps in mathematical reasoning (Lightman et al., 2024). However, executable tasks and local checks alone do not determine which parts of a coding rollout should become model training targets or whether a failure points to a harness problem.

The same execution feedback can also expose harness weaknesses. Self-Harness identifies model-specific failure patterns in execution traces, proposes harness edits, and validates them with regression tests (Zhang et al., 2026a). Recursive Harness Self-Improvement refines a prompt-level agent loop using feedback on its revision history (Lee et al., 2026). These studies show how agent experience can guide harness changes. They do not also use that feedback to construct a fine-grained corpus for model SFT.

TMT connects these two uses through records created by admitted subtask tests. Each record links a subtask and its methods to a tested code state and result. PASS records support the selection of model training targets, while FAIL records guide harness diagnosis and repair.

3 Method

Section 3.1 presents the full update cycle. Section 3.2 explains how we collect TMT records. Section 3.3 uses passing records to train the model. The updated model then uses FAIL records to improve the harness, as described in Section 3.4.

3.1 TMT Co-evolution

At update cycle i, model M_i acts through harness H_i. The harness provides context and tools, controls execution, and stores task state and test feedback. We group its responsibilities into seven components (Table 1). Let $\mathcal{D}_{\text{train}}$, $\mathcal{D}_{\text{dev}}$, and $\mathcal{D}_{\text{test}}$ denote the training, development, and test sets, respectively. TMT records collected from $\mathcal{D}_{\text{train}}$ support both a model update and a harness update:

$$(M_i, H_i) \xrightarrow{\text{PASS records}} (M_{i+1}, H_i) \xrightarrow{\text{FAIL records}} (M_{i+1}, H_{i+1}). \tag{1}$$

We first update the model and then hold it fixed while adapting the harness. This separates changes to the model weights from changes to the harness. FAIL records include TMT FAIL records and whole-task failures from fallback rollouts.

We use DeepSeek-V4-Flash-0731 as the teacher for cold-start data collection. It splits tasks into subtasks, builds tests, and runs methods. The tests check the code for each subtask and return PASS or FAIL. We first tried Qwen3.5-9B for task decomposition, TDD test generation, and harness updates. It failed to complete these steps, and the tests it generated failed admission, forcing tasks onto the non-TDD route. These failures also blocked its use in later cycles without a teacher. We therefore choose Qwen3.6-27B as the student. After fine-tuning, the student uses its own harness to group failures, diagnose defects, and propose repairs. The teacher does not take part in these steps.

Table 1: Seven components of the coding harness.

Symbol	Function	Symbol	Function
I	Task and session gateway	P	Planning and routing
C	Context construction	T	Tools and execution
S	State, workspace, and memory	V	Verification and test feedback
R	Retry, repair, rollback, and stopping		

3.2 Collecting TMT Records

From subtasks and methods to TMT records. Figure 1(a) shows task decomposition, test preparation, and record collection. Each original coding task in $\mathcal{D}_{\text{train}}$ is divided into subtasks with explicit goals, dependencies, and checks. Following test-driven development (TDD), model constructs and admits tests for each subtask before implementation. Each admitted test run creates a TMT record. The record stores the methods used since the subtask's last test run. It also stores the tested code state and the PASS or FAIL result:

$$z = (s, \mathbf{m}, \tau, X, y), \qquad y = \text{Test}(\tau, X) \in \{\textsc{Pass}, \textsc{Fail}\}. \tag{2}$$

Here, s is the subtask, $\mathbf{m}$ is the sequence of methods since the previous test execution, τ is the admitted and frozen test, and X is the tested code state. The result y remains tied to that test and code state. A test checks the resulting code. It does not enforce every method in $\mathbf{m}$ to be necessary or correct. A retry creates a new record and does not change an earlier result. Multiple records may therefore refer to the same subtask. We denote all TMT records collected during cycle i by $\mathcal{D}_i^{\text{TMT}}$.

Test admission and freezing. For reference-assisted admission, a target test must reach its declared assertion and fail on the initial code C_0. It must then pass on the private reference state C_{ref}. Preservation tests must pass on both states. Import or setup errors do not count as a target-test failure. Changing an admitted test or its execution command requires a new admission check.

Reference code is used only to build and admit tests. During method rollouts, the model sees public task information and sanitized feedback. Private tests and reference patches remain hidden. Rejected contracts send tasks to the non-TDD route. Fallback rollouts are excluded from TMT-SFT, but their whole-task failures support harness updates. Contract rejection itself does not create a TMT FAIL record. Appendix B gives the full admission checks and access rules.

Retries and whole-task outcomes. Figure 2 shows how a subtask's test result can differ from the whole-task outcome. The example uses an integer parser with two requirements: reject empty input and parse negative numbers. In Run A, the input check passes but the negative-number check fails. The PASS record can still supply training targets. In Run B, a failed input-check attempt is repaired and both subtasks pass. The earlier FAIL record remains available for harness diagnosis. The same logic applies to other tasks with separately tested subtasks and retries. The tests for each subtask determine its PASS or FAIL results. Whole-task outcomes are stored separately.

3.3 Model Update from PASS Records

Selecting training targets. Let $\mathcal{D}_i^{+} = \{z \in \mathcal{D}_i^{\text{TMT}} : y(z) = \textsc{Pass}\}$ be the PASS records collected during update cycle i. From these records, we select code edits and relevant investigation actions as training targets, retaining each selected action with its original public context. This selection forms the "TMT-SFT corpus" in Figure 1(b). An action is supervised only once, even if multiple tests

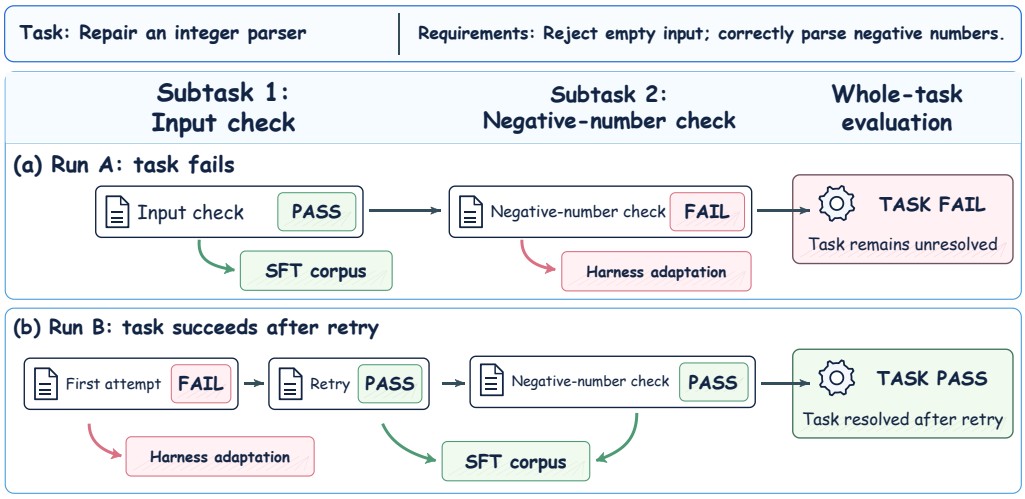

Figure 2: **TMT records for model and harness updates.** (a) Run A fails overall, but its passing input-check record can supply SFT targets. (b) Run B succeeds after a retry, but its earlier failed record remains useful for harness adaptation. Each tested attempt may contain several methods. SFT uses selected actions from passing records. Each record keeps the PASS or FAIL result from its test.

support it, with failed attempts and unselected actions excluded. Test source, reference solutions, and standalone private reasoning fields are excluded.

Recovery examples link a failed attempt to a verified repair. The original context contains the observed test-failure feedback. A later retained edit passes the same test version in the same environment. Code states connect the failure to the repair, so an unsupported failure claim alone is not enough. For a chain of failed edit, failure feedback, and verified repair, the earlier exchange may remain in context, but only the selected repair is a positive target. This preserves the context in which the repair was made without training the failed action. Full selection rules are in Appendix B.2.

Masked, task-normalized training. We use LoRA (Hu et al., 2022) to reduce trainable parameters and GPU memory use during fine-tuning. The "LoRA SFT" step in Figure 1(b) updates M_i using selected assistant-action tokens. It produces M_{i+1}, shown in the "Updated model" block. Let $\mathcal{Q}_i$ be the tasks with at least one selected target. For task q, let $\mathcal{A}_q$ be the set of selected target-token instances (c, a, j), where c is the context, a is the assistant action, and j indexes a token in that action. We minimize

$$\mathcal{L}^{(i)}_{\mathrm{TMT}}(\theta) = -\frac{1}{|\mathcal{Q}_i|} \sum_{q \in \mathcal{Q}_i} \frac{1}{|\mathcal{A}_q|} \sum_{(c,a,j) \in \mathcal{A}_q} \log p_\theta(a_j \mid c, a_{<j}). \tag{3}$$

Here, $p_\theta(a_j \mid c, a_{<j})$ is the probability assigned by the fine-tuned model to token a_j, conditioned on context c and the preceding tokens $a_{<j}$ of action a. Each task contributes equally, regardless of how many selected tokens it contains. Prompts, tool outputs, failed actions, unselected actions, and padding are not training targets.

3.4 Harness Update from FAIL Records

Harness updates use two types of FAIL records: TMT records whose admitted tests fail, and fallback rollouts that fail whole-task evaluation. Let $\mathcal{F}_i$ contain the fallback failures. We combine both sources as

$$\mathcal{D}_i^- = \{z \in \mathcal{D}_i^{\mathrm{TMT}} : y(z) = \textsc{Fail}\} \cup \mathcal{F}_i. \tag{4}$$

The set $\mathcal{D}_i^+$ supplies model training targets, while $\mathcal{D}_i^-$ supplies evidence for harness updates. Each TMT FAIL record retains its subtask s, methods $\mathbf{m}$, frozen test τ, tested code state X, and result y.

Fallback failures carry whole-task results and execution feedback. They have no admitted subtask test. With model M_{i+1} held fixed, the coding model uses its harness to aggregate the failures, diagnose harness defects, and propose repairs. Each role uses a separate prompt and context; none uses an external meta-agent. Inputs preserve record identifiers, test results, and execution feedback. The following stages turn this evidence into failure groups, diagnoses, and a candidate harness for development evaluation.

Aggregation. In the "Group failures" block of Figure 1(c), the model first groups failures within partitions. It then merges related groups across the batch. Each supplied FAIL record must be covered once. Groups describe shared symptoms and cite the records they contain. Aggregation organizes repeated failures but does not by itself establish their cause.

Diagnosis. In the "Diagnose" block of Figure 1(c), the model checks each group against the harness code and its expected behavior. It distinguishes harness defects from coding errors, test defects, infrastructure failures, and unclear cases. A harness diagnosis must identify a harness responsibility that was not met. For example, a subtask failing its tests does not by itself show that the harness verified it incorrectly. A claim that feedback was missing must be checked against what the verifier returned and what the agent received. Unsupported diagnoses do not justify a patch.

Repair and selection. The "Repair" block in Figure 1(c) uses supported diagnoses to propose edits within the allowed harness components. The "Patch" block applies these edits to form H_{cand}. Each proposal identifies the affected code, intended change, and regression checks. It cannot change the frozen test results, data splits, or reference-access rules. If no supported edit is found, the current harness is retained.

The "Development evaluation" block in Figure 1(c) compares H_{cand} with H_i on $\mathcal{D}_{\text{dev}}$, with M_{i+1} fixed. Task manifests, tool permissions, and inference budgets are shared. The configured rule requires at least one additional solved task and no increase in the infrastructure failure rate. Regression checks and the development decision are recorded with the patch. If the candidate fails the selection rule, H_i is retained. The fixed test set $\mathcal{D}_{\text{test}}$ is then used to evaluate the retained model–harness pair. Appendix A.1 reports the first-cycle checks and development selection. Retained edits change harness code, not model weights. The updated harness generalizes to frozen external models.

4 Experiments

4.1 Experimental Setup

We evaluate one model–harness update cycle through three questions. **RQ1** examines how joint and individual updates affect task success and inference efficiency on SWE-bench and eight zero-shot coding benchmarks. **RQ2** compares TMT-SFT with trajectory-level SFT in task success and training-data efficiency. **RQ3** examines whether the updated harness generalizes across multiple frozen external models on SWE-bench.

Training and development data. We use 1,000 SWE-smith repository tasks (Yang et al., 2025) as $\mathcal{D}_{\text{train}}$ for cold-start data collection. TMT rollouts decompose these repository-level tasks into behavioral subtasks and evaluate their methods using admitted, frozen local tests. Harness selection uses $\mathcal{D}_{\text{dev}}$, a separate set of 300 SWE-smith tasks. The training and development sets do not overlap: $\mathcal{D}_{\text{dev}} \cap \mathcal{D}_{\text{train}} = \varnothing$. Appendix E summarizes the data sources and split sizes.

Systems. The initial system combines the Qwen3.6-27B checkpoint M_0 with a harness H_0 built on mini-swe-agent, a lightweight implementation from the SWE-agent project (Yang et al., 2024). DeepSeek-V4-Flash-0731 serves as the frozen teacher during cold-start data collection. LoRA fine-tuning on the selected PASS records produces M_1. The same frozen student model then uses FAIL evidence to aggregate failures, diagnose defects, and propose harness repairs. Candidate harnesses are compared with H_0 on $\mathcal{D}_{\text{dev}}$, holding M_1 fixed, to select H_1. The teacher does not participate in harness adaptation.

For the SFT comparison, Raw-SFT retains complete trajectories accepted by the benchmark verifier, whereas TMT-SFT retains locally verified actions from TDD-based rollouts. Both corpora use the

same source tasks and cold-start teacher, and both updates start from M_0. Corpus sizes and training settings are reported in Appendix E.2.

Evaluation benchmarks. SWE-bench (Jimenez et al., 2024) tests repository repairs using executable tests. The zero-shot suite contains 516 tasks across DSCodeBench (data science; Ouyang et al., 2026), EffiBench-X and Mercury (code correctness and efficiency; Qing et al., 2025; Du et al., 2024), MBPP (Python functions; Austin et al., 2021), xCodeEval (multilingual code; Khan et al., 2024), BIRD (text-to-SQL; Li et al., 2023), BigCodeBench (library composition; Zhuo et al., 2025), and ODEX (open-domain Python; Wang et al., 2023). None supplies training data or harness-selection feedback.

Evaluation settings. We use a Model $\times$ Harness matrix on the same 170 held-out SWE-bench tasks to assess the individual and joint updates. The eight-benchmark results are reported in a separate benchmark-level table. Harness generalization and the in-domain SFT ablation also use the held-out SWE-bench tasks. The development and held-out test sets are disjoint: $\mathcal{D}_{\text{dev}} \cap \mathcal{D}_{\text{test}} = \varnothing$.

The primary metric, $S(M, H; \mathcal{D})$, is the percentage of tasks in evaluation set $\mathcal{D}$ resolved by model–harness pair (M, H) according to the benchmark verifier. For the eight-benchmark suite, overall pass rate is the fraction of all 516 tasks solved; unjudged outcomes count as failures. Overall token use is the task-count-weighted average of the per-benchmark token means. Pass rates are rounded to one decimal place. Mean tokens are reported in thousands, rounded to integers for SWE-bench and one decimal place for the zero-shot suite. We report the two evaluations separately: SWE-bench tests improvements on repository-repair tasks of the same type as the SWE-smith training tasks, whereas the eight zero-shot benchmarks test generalization to other coding task families outside the training distribution.

All evaluations follow the standard non-TDD inference route, with the benchmark verifier scoring the final output. The agent cannot access reference solutions or hidden verifier feedback. Within each comparison, task manifests, decoding settings, tool permissions, and inference budgets are shared. Local test pass rates measure supervision yield and are not substituted for benchmark task success. Appendix E gives further details on the data and evaluation settings.

4.2 RQ1: How Do Joint and Individual Updates Affect Task Success and Inference Efficiency?

Results on SWE-bench. Figure 3 compares all four model–harness pairs on the same held-out tasks. The initial system achieves 64.1% pass rate, compared with 65.9% for the model-only update, 65.3% for the harness-only update, and 67.6% for the joint update. The joint system also uses the fewest rollout tokens: 1152k per task, compared with 1362k, 1230k, and 1316k, respectively. It improves pass rate over the initial system by 3.5 percentage points while reducing mean token use by 15.4%.

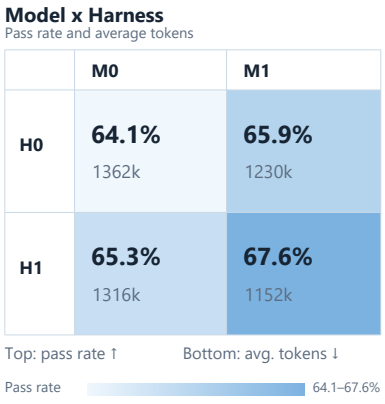

Figure 3: **SWE-bench matrix.** Pass rate and mean tokens (k).

Results on zero-shot benchmarks. Table 2 reports results on the eight zero-shot benchmarks. The initial system achieves 53.1% overall pass rate; the model-only and harness-only systems achieve 54.5% and 54.3%, respectively. The jointly updated system achieves the highest overall pass rate, 55.0%. It also has the lowest overall mean token use: 117.9k tokens per task, compared with 215.6k for the initial system, 150.2k for the model-only system, and 182.1k for the harness-only system.

Analysis. The joint update achieves the highest pass rate and lowest mean token use on SWE-bench and overall across the zero-shot suite. Each update improves SWE-bench performance with either version of the other component, suggesting complementary contributions from model learning

Table 2: **First-cycle cross-evaluation on eight zero-shot benchmarks.** Success rates are percentages; tokens are mean rollout consumption per task in thousands. Bold and underlining mark the best and second-best distinct values within each metric and row, including ties. The Overall row reports success across 516 tasks and task-count-weighted mean token use.

		(M_0, H_0)		(M_1, H_0)		(M_0, H_1)		(M_1, H_1)	
Benchmark	n	Pass	Tok./task	Pass	Tok./task	Pass	Tok./task	Pass	Tok./task
DSCodeBench	55	47.3	189.1	**49.1**	**153.5**	**49.1**	274.0	47.3	281.9
EffiBench-X	86	55.8	299.9	60.5	**117.0**	51.2	221.8	**64.0**	144.6
Mercury	54	98.1	11.5	98.1	**9.1**	98.1	12.5	**100.0**	54.6
MBPP	50	**2.0**	40.0	**2.0**	10.6	**2.0**	11.6	**2.0**	**10.3**
xCodeEval	50	52.0	1372.8	54.0	1098.3	**56.0**	1025.1	48.0	**525.9**
BIRD	50	56.0	11.5	54.0	**8.0**	**60.0**	19.3	58.0	9.0
BigCodeBench	82	64.6	20.6	65.9	**20.4**	**68.3**	27.1	67.1	21.5
ODEX	89	43.8	17.0	44.9	11.1	**46.1**	46.6	44.9	**10.5**
Overall	516	53.1	215.6	54.5	150.2	54.3	182.1	**55.0**	**117.9**

and harness adaptation. The 45.3% reduction in overall zero-shot token use accompanies an increase in task success, demonstrating a favorable aggregate balance between effectiveness and efficiency. The gains on EffiBench-X and Mercury further support transfer of the jointly updated system beyond the repository-repair tasks used for training.

4.3 RQ2: How Does TMT-SFT Compare with Trajectory-Level SFT?

Training volume. Table 6 in Appendix E.2 shows that TMT-SFT uses 50.3% as many full-sequence tokens and 3.5% as many supervised tokens as Trajectory-Level SFT (Raw-SFT).

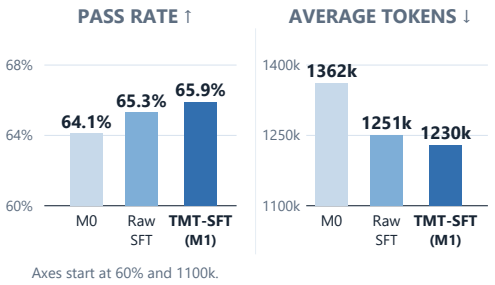

Figure 4: **SFT ablation on SWE-bench.** Fixed H_0; mean tokens (k). Axes are truncated.

Results on SWE-bench. Figure 4 compares the initial checkpoint, Raw-SFT, and TMT-SFT with H_0 fixed. Their pass rates are 64.1%, 65.3%, and 65.9%, respectively, and their mean rollout token counts are 1362k, 1251k, and 1230k. TMT-SFT exceeds Raw-SFT by 0.6 percentage points and uses 1.7% fewer rollout tokens.

Results on zero-shot benchmarks. Table 3 reports the fixed-H_0 comparison across the eight benchmarks. Raw-SFT and TMT-SFT both achieve 54.5% overall pass rate, compared with 53.1% for the initial checkpoint. TMT-SFT uses 150.2k rollout tokens per task overall, compared with 164.3k for Raw-SFT and 215.6k for the initial model. TMT-SFT attains the best or joint-best pass rate on five benchmarks and uses the fewest tokens on seven.

Analysis. TMT-SFT exceeds Raw-SFT's SWE-bench pass rate and matches its overall zero-shot pass rate, with lower mean rollout token use in both settings. Achieving these results with a smaller training corpus supports locally verified methods as an efficient source of supervision. TMT retains successful local work even from rollouts that fail overall and focuses training targets on actions supported by passing tests. This provides a plausible explanation for its data efficiency and connects the record-level design to the observed gains in model performance.

Table 3: **Zero-shot SFT ablation under fixed H_0.** Pass rates are percentages; tokens are per-task means (k). Bold and underlining mark the best and second-best distinct values, including ties. Overall aggregates follow Table 2.

		Initial M_0		Raw-SFT		TMT-SFT (M_1)	
Benchmark	n	Pass	Tok./task	Pass	Tok./task	Pass	Tok./task
DSCodeBench	55	47.3	189.1	47.3	293.4	**49.1**	**153.5**
EffiBench-X	86	55.8	299.9	59.3	140.1	**60.5**	**117.0**
Mercury	54	**98.1**	11.5	**98.1**	14.3	**98.1**	**9.1**
MBPP	50	**2.0**	40.0	0.0	24.9	**2.0**	**10.6**
xCodeEval	50	52.0	1,372.8	50.0	**1,002.4**	**54.0**	1,098.3
BIRD	50	**56.0**	11.5	**56.0**	13.3	54.0	**8.0**
BigCodeBench	82	64.6	20.6	**68.3**	27.9	65.9	**20.4**
ODEX	89	43.8	17.0	**47.2**	17.2	44.9	**11.1**
Overall	516	53.1	215.6	**54.5**	164.3	**54.5**	**150.2**

4.4 RQ3: Does the Updated Harness Generalize Across Models?

Results across frozen external models. Figure 5 compares H_0 and H_1 with DeepSeek-V4.1-Flash and MiniMax-M2.7. The models remain frozen, with no model-specific harness selection. For DeepSeek-V4.1-Flash, H_1 raises pass rate from 71.8% to 73.5% and reduces mean rollout tokens from 781k to 743k. For MiniMax-M2.7, pass rate increases from 67.1% to 68.2%, while tokens increase from 906k to 1032k.

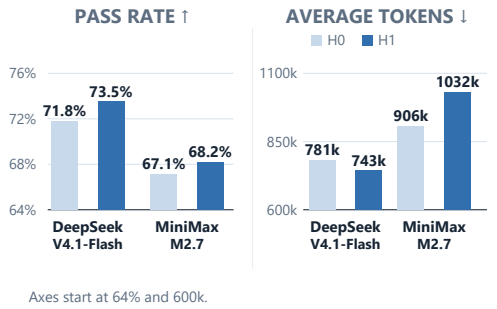

Figure 5: **Harness generalization on SWE-bench.** Mean tokens (k). Axes are truncated.

Analysis. The pass-rate gains on both frozen external models support harness generalization beyond the student. With no external-model fine-tuning or model-specific harness selection, these gains demonstrate reuse across model backbones. DeepSeek-V4.1-Flash also achieves lower token use, improving effectiveness and efficiency together. These results support harness adaptation as a reusable way to improve coding agents without an additional model update.

5 Conclusion

We present TMT, a framework that uses local test feedback to improve both a coding model and its harness. TMT links subtasks, methods, and admitted tests to guide two complementary updates. PASS records provide model supervision, including successful work from rollouts that fail overall; failure evidence guides harness aggregation, diagnosis, and repair by the updated student. After one teacher-assisted cycle, the jointly updated system achieves higher pass rates and lower mean rollout token use than the initial system and either update alone on SWE-bench and overall across eight zero-shot benchmarks. TMT-SFT exceeds Raw-SFT's SWE-bench pass rate and matches its overall zero-shot pass rate with fewer training and rollout tokens. The updated harness also improves task success for both frozen external models. These findings establish local test records as useful evidence for model–harness co-evolution, connecting data-efficient learning with reusable harness improvements.

6 Limitations

TMT is used during data generation, not during final benchmark inference. We evaluate only one update cycle. We have not tested later cycles in which the student model replaces the teacher to generate TMT records.

AI USE STATEMENT

Generative AI assisted language polishing and the refinement of experimental analyses in this manuscript. Training data for the first update cycle were generated with AI assistance. The authors take full responsibility for the submitted content.

ETHICS STATEMENT

Coding agents execute generated programs and may inherit unsafe or biased behavior from their training data. The experimental setup therefore requires isolated execution, restricted access to private evaluation material, and review of task and repository licenses before redistribution. Improved benchmark correctness does not establish that generated code is secure or suitable for unattended deployment. The work studies software tasks and does not include a human-subject study. Any later human annotation or release of derived data will require a corresponding update to this statement.

REPRODUCIBILITY STATEMENT

Section 3 defines TMT and the seven harness components. Appendices A and B give update and supervision details. Appendices C and D provide prompts and checked admission examples. Appendix E gives the data sources and training and inference settings.

REFERENCES

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, 2021. URL https://arxiv.org/abs/2108.07732.

Kent Beck. *Test Driven Development: By Example*. Addison-Wesley Professional, 2002. ISBN 9780321146533. URL https://www.pearson.com/en-us/subject-catalog/p/Beck-Test-Driven-Development-By-Example/P200000009421/9780321146533.

Mingju Chen, Can Lv, Guibin Zhang, Heng Chang, and Shiji Zhou. HarnessForge: Joint harness and policy evolution for adaptive agent systems, 2026. URL https://arxiv.org/abs/2606.01779.

Mingzhe Du, Luu Anh Tuan, Bin Ji, Qian Liu, and See-Kiong Ng. Mercury: A Code Efficiency Benchmark for Code Large Language Models. In *Advances in Neural Information Processing Systems*, volume 37, pp. 16601–16622, 2024. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/1df1df43b58845650b8dada00fca9772-Paper-Datasets_and_Benchmarks_Track.pdf.

Tianyu Fan and Chao Huang. HELIX: Model-harness co-evolution for recursive self-improvement, 2026. URL https://arxiv.org/abs/2608.13951.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. In *The Tenth International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=nZeVKeeFYf9.

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? In *The Twelfth International Conference on Learning Representations*, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/edac78c3e300629acfe6cbe9ca88fb84-Abstract-Conference.html.

Mohammad Abdullah Matin Khan, M Saiful Bari, Xuan Long Do, Weishi Wang, Md Rizwan Parvez, and Shafiq Joty. XCodeEval: An execution-based large scale multilingual multitask benchmark for code understanding, generation, translation and retrieval. In *Proceedings of the*

62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), 2024. URL https://aclanthology.org/2024.acl-long.367/.

Hyunin Lee, Jinglue Xu, Jeffrey Seely, Donghyun Lee, Matei Zaharia, and Yujin Tang. Recursive harness self-improvement, 2026. URL https://arxiv.org/abs/2607.15524.

Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, Xuanhe Zhou, Chenhao Ma, Guoliang Li, Kevin Chang, Fei Huang, Reynold Cheng, and Yongbin Li. Can LLM already serve as a database interface? a BIg bench for large-scale database grounded Text-to-SQLs. In *Advances in Neural Information Processing Systems*, volume 36, 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/83fc8fab1710363050bbd1d4b8cc0021-Abstract-Datasets_and_Benchmarks.html.

Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://arxiv.org/abs/2305.20050.

Shuyin Ouyang, Dong Huang, Jingwen Guo, Zeyu Sun, Qihao Zhu, and Jie M. Zhang. DSCodeBench: A realistic benchmark for data science code generation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pp. 32628–32636, 2026. doi: 10.1609/aaai.v40i38.40540. URL https://ojs.aaai.org/index.php/AAAI/article/view/40540.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. Training software engineering agents and verifiers with SWE-Gym. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 47717–47737, 2025. URL https://proceedings.mlr.press/v267/pan25g.html.

Yuhao Qing, Boyu Zhu, Mingzhe Du, Zhijiang Guo, Terry Yue Zhuo, Qianru Zhang, Jie M. Zhang, Heming Cui, Siu Ming Yiu, Dong Huang, See-Kiong Ng, and Anh Tuan Luu. EffiBench-X: A multi-language benchmark for measuring efficiency of LLM-generated code. In *Advances in Neural Information Processing Systems*, volume 38, 2025. URL https://proceedings.neurips.cc/paper_files/paper/2025/hash/6b8198f777d1ba87a8250368db2ea16a-Abstract-Datasets_and_Benchmarks_Track.html.

Zhiruo Wang, Shuyan Zhou, Daniel Fried, and Graham Neubig. Execution-based evaluation for open-domain code generation. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, 2023. URL https://aclanthology.org/2023.findings-emnlp.89/.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems*, volume 37, 2024. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/5a7c947568c1b1328ccc5230172e1e7c-Abstract-Conference.html.

John Yang, Kilian Lieret, Carlos E. Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. SWE-smith: Scaling data for software engineering agents, 2025. URL https://arxiv.org/abs/2504.21798.

Hangfan Zhang, Shao Zhang, Kangcong Li, Chen Zhang, Yang Chen, Yiqun Zhang, Lei Bai, and Shuyue Hu. Self-Harness: Harnesses that improve themselves, 2026a. URL https://arxiv.org/abs/2606.09498.

Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. Darwin Gödel machine: Open-ended evolution of self-improving agents. In *The Fourteenth International Conference on Learning Representations*, 2026b. URL https://arxiv.org/abs/2505.22954.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xiong-Hui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, Bingnan Zheng, Bang Liu, Yuyu Luo, and Chenglin Wu. AFlow: Automating agentic workflow generation. In *The Thirteenth International Conference on Learning Representations*, 2025a. URL https://openreview.net/forum?id=z5uVAKwmjf.

Lei Zhang, Jiaxi Yang, Min Yang, Jian Yang, Mouxiang Chen, Jiajun Zhang, Zeyu Cui, Binyuan Hui, and Junyang Lin. Synthesizing software engineering data in a test-driven manner. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 76518–76540, 2025b. URL https://proceedings.mlr.press/v267/zhang25cn.html.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, Simon Brunner, Chen Gong, James Hoang, Armel Zebaze, Xiaoheng Hong, Wen-Ding Li, Jean Kaddour, Ming Xu, Zhihan Zhang, Prateek Yadav, Naman Jain, Alex Gu, Zhoujun Cheng, Jiawei Liu, Qian Liu, Zijian Wang, Binyuan Hui, Niklas Muennighoff, David Lo, Daniel Fried, Xiaoning Du, Harm de Vries, and Leandro Von Werra. BigCodeBench: Benchmarking Code Generation with Diverse Function Calls and Complex Instructions. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/a6a90bcc2aa470c3871b2d39a67d26e8-Paper-Conference.pdf.

A First-Cycle Update Procedure

Algorithm A.1 details the first update cycle in Section 3. The initial pair is (M_0, H_0), with Qwen3.6-27B as the student and a SWE-agent-based harness. A single teacher-assisted collection supplies both model supervision and harness-repair evidence. Harness selection uses $\mathcal{D}_{\text{dev}}$; held-out evaluation tasks do not select candidate versions.

Algorithm A.1 Full first-cycle TMT update

1. **Input:** (M_0, H_0); SWE-smith training tasks $\mathcal{D}_{\text{train}}$; cold-start teacher DeepSeek-V4-Flash-0731; $\mathcal{D}_{\text{dev}}$; admission rules and comparison budgets.
2. **Collect TMT evidence.** Under H_0, the teacher assists task decomposition, test construction, and method rollout. Admit and freeze local tests before implementation. Tasks that fail admission follow the non-TDD fallback. These rollouts are excluded from TMT-SFT; their failures under whole-task evaluation are used for harness adaptation.
3. **Partition records.** Store the collected local records as $\mathcal{D}_0^{\text{TMT}}$. Retain eligible PASS records for SFT selection. Let $\mathcal{F}_0$ contain whole-task failures from fallback rollouts, and set $\mathcal{D}_0^- = \{z \in \mathcal{D}_0^{\text{TMT}} : y(z) = \textsc{Fail}\} \cup \mathcal{F}_0$. Preserve each attempt's original test binding and label.
4. **Update the model.** Construct the TMT selected-action corpus with its public contexts, deduplication, and loss masks. Fine-tune M_0 to obtain M_1 and freeze the resulting checkpoint.
5. **Propose a harness update.** Using both types of FAIL records in $\mathcal{D}_0^-$, run aggregation, diagnosis, and repair with the same frozen student model under role-specific prompts and separate contexts. Produce a candidate H_{cand} targeting evidence-supported defects in H_0.
6. **Select the harness.** Compare (M_1, H_{cand}) with (M_1, H_0) on $\mathcal{D}_{\text{dev}}$ under the same evaluation settings. Accept the candidate only if it satisfies the selection criterion; otherwise retain H_0 as H_1.
7. **Output:** (M_1, H_1), the selected training data, proposed changes, and development decision.

The teacher does not perform harness aggregation, diagnosis, repair, or candidate validation. TMT FAIL records come from $\mathcal{D}_0^{\text{TMT}}$; fallback failures are collected in the same data-collection stage, without an additional collection rollout. The procedure can repeat with new evidence in subsequent cycles; the experiments in this paper cover only the first cycle.

A.1 Role handoffs and candidate evaluation

Aggregation first groups sanitized local FAIL records and whole-task failures from fallback rollouts within partitions, then reconciles provisional clusters across the completed batch. All supplied failures must be covered without duplication. Diagnosis distinguishes harness defects from coding defects, test defects, infrastructure failures, and uncertain causes. A proposed harness defect must be supported by evidence of a violation of the frozen harness contract. Repair consumes this report and proposes a bounded change to the relevant components in Table 1. These are prompted roles of one model, not independently trained agents. Appendix C includes both aggregation stages and the diagnosis and repair prompts.

Candidate and baseline share the frozen checkpoint, development tasks, tool permissions, and inference budgets. A rejected proposal leaves the selected harness unchanged. Table 4 summarizes the first-cycle update and its development decision.

Table 4: Evidence-grounded first-cycle harness update from H_0 to H_1. The development comparison uses 300 SWE-smith tasks with M_1 fixed.

Update	Failure evidence	Diagnosis and change	Validation status
$H_0 \rightarrow H_1$	908 failure records from 490 tasks were grouped into 303 clusters. An exact-cover check verified that every source record appeared once.	Two harness defects were identified. Component V was updated. The proposed change to component S was outside the permitted editable scope and was therefore not applied.	With M_1 fixed, the candidate was compared with H_0 on 300 SWE-smith development tasks. This comparison selected H_1. On the same 82-test regression suite, the candidate introduced no new failures relative to H_0.

B Test Contracts and Supervision

B.1 Admission and access boundaries

TMT follows test-driven development: the model writes tests before implementing the corresponding behavior. In reference-assisted construction, the ordered checks in Box 7 compare the initial state C_0 with a private reference state C_{ref}. A target test must reach its declared assertion and fail on C_0, then pass on C_{ref}. Preservation tests must pass on both states. Import and setup failures do not establish the target behavior and cannot serve as successful admission evidence.

Admitted tests and their execution commands are frozen. Changing either requires a new admission check; the previous execution result cannot be reused for the changed contract. Later code edits and retries do not retroactively change an earlier record's verdict. A defective contract is excluded from supervision.

Reference material is available only to the reference-assisted construction and admission path. method rollout and harness-repair prompts receive public task information and sanitized execution evidence, not gold patches or private test source. Admission failure triggers the default non-TDD fallback. These fallback rollouts are excluded from TMT-SFT, but their failures under whole-task evaluation are used for harness adaptation. A rejected test candidate is not a solver FAIL TMT: local FAIL records arise only when already admitted tests are executed. Held-out evaluation follows the non-TDD route with benchmark verification.

B.2 TMT record selection and training masks

Each record binds a task, objective, a sequence of one or more methods, code states, frozen test, execution evidence, and local verdict. Each method comprises one or more model responses and their associated tool calls. A record is tied to a test execution. Separately available whole-task outcomes remain metadata.

The TMT corpus retains selected code edits and relevant investigation actions supported by local PASS evidence, together with selected repairs conditioned on earlier failures. Recovery selection requires the failure feedback to be visible in the original public input, followed by a retained edit and a passing execution of the same test version in the same environment. Recorded code states must connect the failure to the verified repair. An assistant's unsupported claim that a test failed is not sufficient evidence. In the selected corpus, repairs automatically trigger the local tests; no separate model-generated test call is synthesized.

Selected actions retain their original public message prefixes and tool-call representations. Standalone reasoning fields, private test source, and reference solutions are excluded. Prompts, tool outputs, failed attempts, unselected actions, and padding receive no positive loss. Public failure feedback can remain as context for a later successful repair. A target action is supervised once even if multiple tests support it; complete fallback trajectories are excluded. Contexts are not truncated to force inclusion: windows that cannot preserve their required public prefix within the length budget are excluded.

Task weighting. The TMT implementation applies the masked token likelihood in Equation 3 with per-task normalization. The task set $\mathcal{Q}_i$, selected target-token instances $\mathcal{A}_q$, and loss $\mathcal{L}^{(i)}_{\mathrm{TMT}}(\theta)$ have the same definitions here as in Section 3. Each task therefore has equal total weight, rather than receiving weight proportional to its retained token count. This weighting does not match the computational budgets of the two SFT routes.

Raw-SFT comparison. Raw-SFT and TMT-SFT use the same source tasks and cold-start teacher, and both fine-tuning runs start from M_0. Raw-SFT retains complete trajectories accepted by the benchmark verifier, whereas TMT-SFT selects actions supported by frozen local tests. The source tasks are shared, but the resulting trajectories and retained training examples differ. The comparison evaluates the complete data-construction pipelines rather than isolating selection or masking alone.

C Prompt Set

Boxes 1 and 2 show partial prompts, retaining the key instructions, inputs, and output structure. Ellipses mark omitted content. Their complete templates, output schemas, and feedback messages are provided in `prompts/sdd.txt` and `prompts/tdd.txt` in the source package. Boxes 3–6 retain the complete implementation templates, including both aggregation stages in Box 4. Double braces denote runtime values; JSON slots preserve their types, and array entries repeat for the supplied items. Box 3 uses Jinja syntax. The source snapshot and file hashes accompany the templates; these are parameterized instructions, not per-run request logs. The displayed coding prompt uses lowercase "method"; its source template is unchanged. This archived prompt uses one response per method, a special case of the definition in Appendix B.2. The archived harness prompts use the name FAIL-TMT for failure capsules; Section 3.4 distinguishes the two failure sources.

Box 1. Task decomposition (partial prompt)

```
SYSTEM (selected instructions)
Build a complete execution map, then break the supplied coding
    task into
fine-grained implementation Subtasks. Return exactly one JSON
   object and no commentary.
[...]
Each Subtask must describe one concrete, independently
   verifiable observable behavior.
[...]
Each Subtask must contain exactly one target acceptance
   criterion, and that target
criterion must cite exactly one supplied requirement source
   span.
[...]
Return Subtasks directly in execution order. Dependencies may
   refer only to supplied Subtask IDs;
do not add grouping layers around the Subtasks.

USER (selected fields)
{
  "task_text": {{task_text}},
  "c0_repository_notes": {{c0_repository_notes}},
  "task_contract": {
    "requirements": {{task.requirements}},
    "allowed_code_scopes": {{task.allowed_code_scopes}}
  },
  "previous_rejection_reasons": {{rejection_feedback}},
  [...]
}
```

Box 1. Task decomposition (partial prompt) (continued)

```
OUTPUT (selected fields)
subtasks: [{subtask_id, description, dependencies,
  acceptance_criteria: [{criterion_id, statement, kind,
    source_spans}], allowed_code_scopes, ...}]
deliverables: [...]

Full template: prompts/sdd.txt
```

Box 2. TDD construction and feedback (partial prompt)

```
SYSTEM (selected instructions)
Write one focused pytest Test for the one supplied acceptance
   criterion.
[...]
Required C0 behavior:
- target/bug_revealing: collect and set up, reach the declared
    assertion, and FAIL there;
- preservation/regression: PASS.
Import, fixture, setup, or unrelated runtime failures are
   invalid.
[...]

REFERENCE-ASSISTED EXTENSION (excerpt)
A target/bug_revealing Test must additionally PASS on C_ref;
a preservation/regression Test must PASS on both states.
[...]

USER (selected fields)
{
  "criterion_target": {{criterion_target}},
  "criterion_scoped_c0_evidence": {{criterion_evidence}},
  "authoring_mode": {{authoring_mode}},
  "previous_candidate": {{previous_candidate_or_null}},
  "previous_candidate_feedback": {{rejection_feedback}},
  [...]
}

AGENTIC OUTPUT AND RETRY (excerpt)
cat > .coevo_tests/test_behavior.py <<'PY'
<complete Python Test source>
PY
[...]
On rejection, rewrite the same file. Never modify project
   source.

Full template, output schema, and feedback: prompts/tdd.txt
```

Box 3. Coding method prompt

```
SYSTEM (Jinja template)
You are the Coding method model controlled by H0.
H0 follows a frozen SDD. Work only on the active Subtask and
   allowed code scopes.
```

Box 3. Coding method prompt (continued)

```
You may inspect the repository and change allowed project code
    or project-owned tests through
bash actions when the active Subtask requires it.
Do not create, modify, discover, or infer hidden/registered
    Test files.
Do not modify .git, Trust Core artifacts, experiment manifests
    , or frozen SDD data.
The Test Registry is frozen before you are queried; only its
    redacted contracts are visible.
The shell already starts in /testbed. Never use /workspace as
    a repository path.
Do not claim a Subtask is complete: only H0's visible Test
    feedback advances the scheduler.
Each response is one method and may contain at most {{
    max_actions_per_method}} actions.
Exploration methods do not execute Tests and do not consume
    repair opportunities.
A method becomes one complete RepairSubmission when it changes
    an allowed workspace path,
or when its current assistant response contains the exact
    marker CODE_REPAIR_SUBMIT, either
in response text or structured action arguments. Use that
    marker only when the current
workspace is ready to be tested without another code change.
In tool-call mode, submit an unchanged ready workspace with
    exactly one harmless bash action:
printf '%s\n' CODE_REPAIR_SUBMIT
Each registered Test is executed for at most {{
    max_test_attempts}} RepairSubmissions. A Test
that still fails on its final RepairSubmission is skipped;
    continue with remaining Tests.
Use compound read-only bash commands when inspecting related
    files. There is no method-count or
exploration-count boundary: pending Tests run only after a
    real code change or explicit
CODE_REPAIR_SUBMIT. Continue until the active Subtask is
    genuinely ready, subject only to the
Harness wall-clock and infrastructure safety timeouts.
Continue the existing investigation; never restart it. A read-
    only method is justified only when
a concrete fact needed for the repair is still missing. Once
    your reasoning identifies the defect
and a specific edit, your next method must apply that edit
    instead of rereading the same files.

USER (Jinja template, rendered for the active method)
<root_task>
{{task}}
</root_task>
<global_task_map>
{{global_task_map}}
</global_task_map>
<frozen_sdd id="{{sdd_id}}" hash="{{sdd_hash}}" registry_hash
    ="{{sdd_registry_hash}}">
Active Subtask: {{active_subtask_id}} -- {{
    active_subtask_title}}
```

Box 3. Coding method prompt (continued)

```
Description: {{active_subtask_description}}
Dependencies: {{active_dependencies | join(', ') if
    active_dependencies else 'none'}}
Acceptance criteria:
{% for criterion in active_acceptance_criteria %}- {{criterion
    }}
{% endfor %}Allowed code scopes: {{active_allowed_scopes |
    join(', ')}}
</frozen_sdd>
<frozen_test_registry hash="{{test_registry_hash}}" />
<visible_test_contracts>
{{visible_test_contracts}}
</visible_test_contracts>
<latest_test_feedback>
{{latest_test_feedback}}
</latest_test_feedback>
<budget>
method {{method_number}}; RepairSubmissions {{
    repair_submissions_used}};
tool calls {{tool_calls_used}}.
</budget>
Analyze the current state and issue bash actions for this
    Subtask. Explore as needed before
submitting a repair; do not use CODE_REPAIR_SUBMIT merely to
    request more diagnostics.
Continue from the previous method reasoning shown in the
    history. If it already names the defect
and a concrete repair, implement that repair now; do not begin
     another repository survey.
When no code change is needed and the current workspace is
    ready, issue exactly one bash
action: printf '%s\n' CODE_REPAIR_SUBMIT
Do not repeat repository inspections already present in the
    method history. Once the relevant
implementation and expected behavior are identified, make the
    smallest allowed code change.

Conversation history and tool observations are supplied by the
     harness.
```

Box 4. Aggregation and reconciliation prompts

```
STAGE 1: PARTITION THE COMPLETE FAILURE BATCH

SYSTEM
Cluster error-centered FAIL-TMT capsules from multiple coding
    tasks. Return one JSON object only. Do not diagnose or
    assign responsibility.

USER (JSON template)
{
  "prompt_version": "d-harness-global-error-aggregation-v1",
  "scope": "one deterministic partition of the complete
    D_harness batch",
  "capsules": [
```

Box 4. Aggregation and reconciliation prompts (continued)

```
    {{item.model_dump(mode='json')}}
  ],
  "rules": [
    "cover every source_tmt_id exactly once and invent no ID",
    "merge only when evidence supports one shared cause;
   otherwise keep singleton",
    "describe observable commonality only; do not assign fault
    or propose repair",
    "do not infer hidden Test content or modify Test truth"
  ],
  "output_schema": {
    "clusters": [
      {
        "member_tmt_ids": [
          "exact supplied source_tmt_id"
        ],
        "title": "short title",
        "shared_failure_pattern": "shared observable pattern",
        "evidence_summary": "bounded evidence"
      }
    ]
  }
}

STAGE 2: RECONCILE PROVISIONAL CLUSTERS ACROSS PARTITIONS

SYSTEM
Reconcile provisional FAIL-TMT clusters across the completed
   batch. Return one JSON object only and preserve unsupported
    distinctions.

USER (JSON template)
{
  "prompt_version": "d-harness-global-reconciliation-v1",
  "error_family": {{family}},
  "provisional_clusters": {{compact}},
  "rules": [
    "cover every cluster_id exactly once and invent no ID",
    "merge across coding tasks only when one cause is
   supported",
    "preserve observable distinctions",
    "do not diagnose responsibility or propose a repair"
  ],
  "output_schema": {
    "clusters": [
      {
        "member_cluster_ids": [
          "exact supplied cluster_id"
        ],
        "title": "short title",
        "shared_failure_pattern": "shared pattern",
        "evidence_summary": "evidence"
      }
    ]
  }
}
```

Box 4. Aggregation and reconciliation prompts (continued)

```
PROVISIONAL CLUSTER INPUT (one object per supplied cluster)
[
  {
    "cluster_id": {{item.cluster_id}},
    "task_count": {{len(item.task_ids)}},
    "title": {{item.title[:200]}},
    "shared_failure_pattern": {{item.shared_failure_pattern
   [:500]}},
    "evidence_summary": {{item.evidence_summary[:500]}}
  }
]
```

Box 5. Diagnosis agent prompt

```
SYSTEM
You are the Diagnose Agent. Determine the cause of each
   aggregated coding failure cluster. Do not design the final
   Harness patch. Return one JSON object only.

USER (JSON template)
{
  "prompt_version": "d-harness-diagnose-v1",
  "frozen_harness_contract": {{harness_contract}},
  "global_error_clusters": [
    {
      "cluster_id": {{item.cluster_id}},
      "task_ids": {{item.task_ids}},
      "task_count": {{len(item.task_ids)}},
      "title": {{item.title}},
      "shared_failure_pattern": {{item.shared_failure_pattern
   }},
      "evidence_summary": {{item.evidence_summary}}
    }
  ],
  "cause_definitions": {
    "harness_defect": "controller, context assembly, tool
   protocol, scheduling, retry, feedback, or recovery behavior
    caused or repeated the failure",
    "coding_defect": "the task solution or repair logic is
   wrong",
    "test_defect": "the generated Test or its criterion
   binding is untrustworthy",
    "infrastructure_failure": "execution substrate or service
   failure",
    "uncertain": "the evidence cannot distinguish the cause"
  },
  "rules": [
    "diagnose every supplied cluster_id exactly once and
   invent no ID",
    "do not call a coding mistake a Harness defect",
    "attribute Harness fault only when it violates the frozen
   Hk contract",
    "use uncertain when causal evidence is insufficient",
```

Box 5. Diagnosis agent prompt (continued)

```
    "only harness_defect may name a component or repair
   direction",
    "do not infer hidden Test content"
  ],
  "output_schema": {
    "diagnoses": [
      {
        "cluster_id": "exact supplied cluster_id",
        "cause": "harness_defect|coding_defect|test_defect|
   infrastructure_failure|uncertain",
        "diagnosis": "causal diagnosis",
        "causal_evidence": "evidence supporting the cause",
        "counterevidence": "counterevidence or 'none observed
   '",
        "target_harness_component": "required only for
   harness_defect",
        "recommended_harness_change": "required only for
   harness_defect",
        "confidence": 0.0
      }
    ]
  }
}
```

Box 6. Repair agent prompt

```
DISPATCH
If there is no actionable diagnosis, the controller returns
   no_change
without a model call. Otherwise, send the following messages.

SYSTEM
You are the Repair Agent. Consume the frozen Diagnose Agent
   report and generate exactly one next-Harness candidate.
   Return one JSON object only.

USER (JSON template)
{
  "prompt_version": "d-harness-repair-v1",
  "source_harness_version": {{diagnosis.source_harness_version
   }},
  "frozen_harness_contract": {{harness_contract}},
  "diagnosis_report_hash": {{diagnosis_hash}},
  "actionable_harness_diagnoses": {{evidence}},
  "constraints": [
    "produce one coherent bounded Harness change or no_change
   ",
    "use only the Diagnose Agent report; do not infer raw TMT
   or Test content",
    "do not change Test truth, retry count, data split, or
   hidden-Test access",
    "do not modify the frozen source Harness in place",
    "acceptance checks must run on independent held-out data"
  ],
  "output_schema": {
```

Box 6. Repair agent prompt (continued)

```
    "decision": "no_change|propose_change",
    "diagnosis": "why the combined repair is justified",
    "target_component": "empty for no_change",
    "proposed_change": "empty for no_change",
    "acceptance_checks": [
      "empty for no_change"
    ]
  }
}

INPUT BINDING
evidence is the JSON serialization of the actionable diagnoses
   , bounded
by max_repair_evidence_chars. The other agents' conversations
   are excluded.
```

D ADMISSION EXAMPLES

The admission gate executes checks; it is not an LLM judge. Box 7 states the rules. Boxes 8 and 9 report two different SWE-Smith (Yang et al., 2025) tasks from recorded teacher-assisted cold-start runs. The python-docx test fails at its declared assertion on C_0 and passes on C_{ref}, so it is admitted. The DeepDiff candidate fails on both states, so it is rejected. Box 8 includes the complete registered test; Box 9 includes the assertion context preserved in the pytest traceback. Task and test IDs, timestamped execution evidence, and source hashes accompany the examples in the source package. These are admission outcomes; whole-task benchmark outcomes are not established by these records. Rejected test candidates do not become solver FAIL TMT records. Private reference material remains unavailable to the solver.

Box 7. Ordered TDD admission rule

```
INPUT
Candidate test, criterion binding, C0, and private C_ref.
The reference is used only in reference-assisted construction.

ORDERED CHECKS (executable rules, not an LLM verdict)
1. Discover the generated test on C0. Reject collection errors
   .
2. Run it on C0. A target must fail at its declared assertion;
   a preservation test must pass. Reject any other outcome.
3. Only after these checks, run on C_ref and require PASS.
4. Admit and freeze the exact test and command, or reject with
   the failing stage and reason. Revision returns to TDD.
Never treat an unrun stage as PASS.
```

Box 8. SWE-Smith: admitted python-docx test

```
BENCHMARK TASK (SWE-Smith; teacher cold start, 2026-08-30)
swe-smith:python-openxml__python-docx.0cf6d71f.
   func_basic__98yzu3no
Test ID: test-7b8fd6acbc314efc7e36e7cb

PUBLIC BEHAVIOR
Setting paragraph_format.space_before = Inches(0.5) must
serialize w:before as "720", without an extra "X".
```

Box 8. SWE-Smith: admitted python-docx test (continued)

```
REGISTERED TEST (complete source)
from docx import Document
from docx.oxml.ns import qn
from docx.shared import Inches


def test_optional_attribute_setter_preserves_value():
    doc = Document()
    paragraph = doc.add_paragraph()
    paragraph_format = paragraph.paragraph_format
    paragraph_format.space_before = Inches(0.5)

    pPr = paragraph_format._element.get_or_add_pPr()
    spacing = pPr.get_or_add_spacing()
    serialized = spacing.get(qn("w:before"))

    assert serialized == "720"


class DescribeOptionalAttributeSetter:
    def it_preserves_the_serialized_attribute_value(self):
        test_optional_attribute_setter_preserves_value()

RECORDED ADMISSION
C0 collection: PASS; 1 test collected; return code 0.
C0 execution: FAIL at assert serialized == "720";
  AssertionError: assert '720X' == '720'; return code 1.
C_ref execution: PASS; 1 passed; return code 0.
Decision: ACCEPT (accepted: true); register and freeze.

The later local PASS TMT is referenced by an exported SFT
sample. These local records do not determine whole-task
   success.
Evidence: examples/evidence/accepted/; admission_provenance.
   json.
```

Box 9. SWE-Smith: rejected DeepDiff test

```
BENCHMARK TASK (SWE-Smith; teacher cold start, 2026-08-30)
swe-smith:seperman__deepdiff.ed252022.lm_rewrite__axead9lx
Test ID: test-8c7fe2f81348212413b7afb2

PUBLIC BEHAVIOR
Applying the ignore-order Delta to t1 should produce t2,
including the required item positions and repetitions.

RECORDED PYTEST ASSERTION CONTEXT
    def test_delta_apply_ignore_order_with_repetition():
        t1 = [5, 1, 3, 1, 4, 4, 6]
        t2 = [7, 4, 4, 1, 3, 4, 8]
        diff = DeepDiff(t1, t2, ignore_order=True,
   report_repetition=True)
        delta = Delta(diff)
        try:
```

Box 9. SWE-Smith: rejected DeepDiff test (continued)

```
            result = t1 + delta
        except Exception:
            result = None
>       assert result == [7, 4, 4, 1, 3, 4, 8]

RECORDED ADMISSION (same candidate, 21:01:37-21:01:44 UTC)
C0 collection: PASS; 1 test collected; return code 0.
C0 execution: FAIL at the declared assertion;
  result is None; return code 1.
C_ref execution: FAIL at the same assertion;
  At index 0 diff: 8 != 7; return code 1.
Decision: REJECT; reference check did not pass.
Admission rule: CREF_EXPECTED_PASS.

A failure on C0 alone is insufficient for admission. The
candidate must also pass on C_ref. This is a rejected test
candidate, not a solver FAIL TMT for harness adaptation.
Evidence: examples/evidence/rejected/; admission_provenance.
    json.
```

E Experimental Settings

E.1 Data sources and splits

The zero-shot suite contains DSCodeBench for data-science code (Ouyang et al., 2026); EffiBench-X and Mercury for code efficiency (Qing et al., 2025; Du et al., 2024); MBPP for short Python functions (Austin et al., 2021); xCodeEval for multilingual code (Khan et al., 2024); BIRD for text-to-SQL (Li et al., 2023); BigCodeBench for library use (Zhuo et al., 2025); and ODEX for open-domain Python (Wang et al., 2023).

Table 5 distinguishes training, development selection, and held-out evaluation. SWE-smith supplies separate cold-start and development tasks; SWE-bench supports the main 2×2 evaluation, harness transfer, and in-domain SFT ablation. The eight additional benchmarks are used only for zero-shot 2×2 model–harness evaluation and fixed-H_0 model-update ablation, without benchmark-specific training or harness selection.

The counts refer to the training and evaluation subsets used in this work, not full dataset sizes. The development set $\mathcal{D}_{\text{dev}}$ contains 300 SWE-smith tasks. It has no overlap with the 1,000 cold-start tasks or the held-out test tasks: $\mathcal{D}_{\text{dev}} \cap \mathcal{D}_{\text{train}} = \varnothing$ and $\mathcal{D}_{\text{dev}} \cap \mathcal{D}_{\text{test}} = \varnothing$. It is used only for harness selection. All systems within a comparison are evaluated on the same test tasks.

E.2 Model training settings

Training volume. Raw-SFT and TMT-SFT start from M_0 and use the same cold-start teacher and optimizer family. Table 6 reports the realized corpus sizes. TMT-SFT uses approximately half as many full-sequence tokens and 3.5% as many supervised tokens as Raw-SFT. These reductions reflect corpus selection and loss masking; the two pipelines are compared at their realized training volumes.

Table 5: Data sources and their experimental roles. Cold-start collection uses 1,000 SWE-smith tasks, and development uses 300 separate SWE-smith tasks. The eight zero-shot subsets contain 516 tasks in total.

Source	Task domain	Role / counting unit	N
SWE-smith (Yang et al., 2025)	Repository repair	Training tasks ($\mathcal{D}_{\text{train}}$)	1,000
SWE-smith ($\mathcal{D}_{\text{dev}}$)	Repository repair	Harness candidate selection	300
SWE-bench (Jimenez et al., 2024)	Repository repair	Main, transfer, and ablation evaluation	170
DSCodeBench (Ouyang et al., 2026)	Data-science code	Zero-shot cross-evaluation	55
EffiBench-X (Qing et al., 2025)	Multilingual code efficiency	Zero-shot cross-evaluation	86
Mercury (Du et al., 2024)	Correctness and efficiency	Zero-shot cross-evaluation	54
MBPP (Austin et al., 2021)	Python function synthesis	Zero-shot cross-evaluation	50
xCodeEval (Khan et al., 2024)	Multilingual code generation	Zero-shot cross-evaluation	50
BIRD (Li et al., 2023)	Text-to-SQL	Zero-shot cross-evaluation	50
BigCodeBench (Zhuo et al., 2025)	Library composition	Zero-shot cross-evaluation	82
ODEX (Wang et al., 2023)	Open-domain Python generation	Zero-shot cross-evaluation	89

Table 6: **Realized SFT corpus size.** The ratio is TMT-SFT divided by Raw-SFT, expressed as a percentage and rounded to one decimal place. Full-sequence tokens count all tokens in the retained training sequences; supervised tokens are positions contributing to the loss.

Metric	TMT-SFT	Raw-SFT	TMT / Raw (%)
Full-sequence tokens	5,900,893	11,739,386	**50.3**
Supervised tokens	182,672	5,151,142	**3.5**

Section 4.3 compares task success and inference token use for these two training pipelines.

M_0 is Qwen3.6-27B and M_1 is the first TMT-SFT checkpoint. Table 7 summarizes the TMT training recipe associated with the corpus in Table 6. Raw-SFT and TMT-SFT have different realized training volumes; their token budgets are not matched.

Selection permits at most eight targets and 65,536 full-sequence tokens per task, one added repair or validation bundle per task, and 12 million full-sequence tokens overall. A recovery bundle permits at most four retained edits and is selected as a unit. The final corpus contains no truncated windows. These are data-construction budgets, not a token-budget match to Raw-SFT.

E.3 Harness and inference settings

The initial harness H_0 uses a mini-swe-agent-based control loop for task handling, tool execution, and trusted local-test checks. Aggregation, diagnosis, and repair use the same frozen student model under different prompts and separate contexts. DeepSeek-V4-Flash-0731 assists cold-start data collection but does not participate in harness adaptation.

Final evaluation follows the non-TDD route with benchmark verification. The student SWE-bench comparison uses non-thinking generation, temperature 1.0, top-p 0.95, top-k 20, a 262,144-token context limit, and an 80,000-token maximum per response. The verifier timeout is 600 seconds,

Table 7: TMT-SFT training recipe. Context budget is the allowed maximum; observed maximum is the longest retained window.

Setting	Configuration
Initial model	Qwen3.6-27B
Cold-start source tasks	1,000 SWE-smith tasks
Corpus	1,021 windows and unique supervised actions
Training framework	PyTorch 2.5.1+cu124; Transformers 5.14.1; PEFT 0.18.1
Distributed recipe	8 H100 GPUs; DeepSpeed 0.16.9; ZeRO-3 with CPU parameter offload
Trainable parameters	Language-model LoRA projection adapters; base weights frozen
LoRA rank / alpha / dropout	16 / 32 / 0.05
Epochs / effective batch	1 / 32 windows (8 GPUs $\times$ 1 window $\times$ 4 accumulation steps)
Optimizer updates	32 for the packaged corpus and batch size
Context budget / observed maximum	32,768 / 15,893 tokens; no truncation
Precision / optimizer	BF16 / AdamW
Peak learning rate / schedule	10^{-4}; 3% warmup; cosine decay to zero
Chat template	Non-thinking; original public message prefixes
Loss	Selected assistant tokens; equal total task weight

with no overall task deadline. Candidate and baseline share evaluation tasks, decoding settings, tool permissions, and budgets within each comparison.

The external-model transfer comparison covers frozen DeepSeek-V4.1-Flash under both H_0 and H_1. Settings are held fixed within each model comparison. Reported metrics are success rate and mean rollout tokens per task. All evaluation results follow the reporting precision in Section 4.1; training-token counts are reported as exact integers.