MOLDCODER: AUTOMATED CONSTRUCTION OF CODE INSTRUCTION DATA USING REAL TEST EXECUTION AS FUNCTIONAL SUPERVISION

Anonymous authors
Paper under double-blind review

ABSTRACT

High-quality code instruction-tuning data are an important foundation for improving the code generation capabilities of pretrained language models. Beyond data scale, task complexity, and data diversity, code instruction-tuning data also require that the code correctly implement the functional requirements specified by its corresponding instruction. Many execution-based data construction methods use model-generated tests to verify synthetic code, with execution outcomes guiding sample filtering or code repair. However, the reliability of this functional supervision remains constrained by the quality of the generated tests. A central question is therefore how to obtain more reliable functional supervision and leverage it more effectively. To this end, we introduce Mold-Instruct, an automated code instruction data construction method that uses real-test execution as functional supervision: it first recovers repository-dependent real tests into standalone executable test–code pairs, then uses functional requirements from the tests to guide instruction construction and execution outcomes for verification and heuristic capability diagnosis, thereby constructing target-specific training data. By adjusting its extensible configurations, Mold-Instruct constructs 46.7K high-quality test–code pairs from open-source code repositories, which are then used to construct training data for two base models and fine-tune the corresponding MoldCoder models. MoldCoder improves all nine evaluated metrics on Qwen3-4B-Base and all seven on SmolLM3-3B-Base, increasing their respective average scores from 45.7 to 51.6 and from 33.1 to 36.9. Further controlled experiments show that real tests outperform target-model-generated tests in our setting, functional constraints and execution verification each provide gains, and capability diagnosis guides supervision selection that further improves performance.

1 INTRODUCTION

High-quality code instruction-tuning data are an important foundation for improving the code generation capabilities of pretrained language models (Luo et al., 2024; Wei et al., 2024b;a). Prior work has improved code instruction-tuning data along dimensions such as data scale, task complexity, and data diversity through instruction evolution, code-grounded task synthesis, and structured code analysis (Luo et al., 2024; Wei et al., 2024b; Wang et al., 2025). However, beyond these dimensions, high-quality code instruction-tuning data also require the code to correctly implement the functional requirements specified by its corresponding instruction. To improve such functional correctness, recent methods use model-generated tests to verify synthesized instruction–code examples and execution feedback to guide sample filtering or code repair (Wei et al., 2024a; Ahmad et al., 2025; Lei et al., 2024; Wang et al., 2025). However, the reliability of such functional supervision remains constrained by the generated tests themselves, while its use is typically limited to post-hoc verification of generated code. A key question is therefore how to obtain more reliable functional supervision and leverage it more effectively.

First, open-source code repositories offer a growing source of real tests written and maintained by developers in real-world development workflows, making them a promising source of functional supervision. However, real tests typically depend on complex repository contexts for execution. Retaining entire repositories introduces substantial irrelevant content and execution overhead, whereas

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.

extracting only local tests and code often breaks executability because of heterogeneous test configurations, complex runtime environments, and missing cross-file dependencies. **Second,** more effective use of functional supervision should go beyond execution verification. Functional constraints encoded in real tests can directly guide instruction construction, helping instructions capture the functional requirements exercised by the tests. Meanwhile, execution outcomes can support heuristic capability diagnosis, further guiding target-specific data selection. Figure 1 summarizes the two central questions considered in this work.

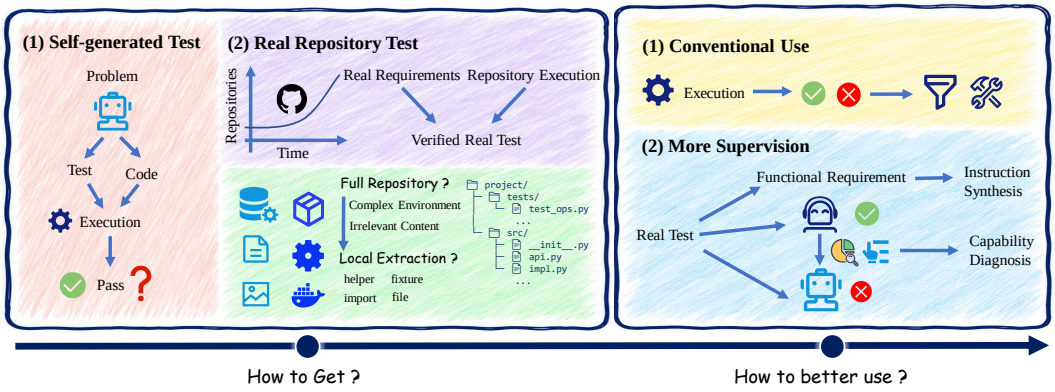


Figure 1: Motivation of Mold-Instruct: how to obtain more reliable functional supervision and how to leverage it more effectively.

To address these challenges, we introduce Mold-Instruct, a two-stage method for automatically constructing code instruction-tuning data with real-test execution as functional supervision. The first stage mines high-confidence real tests from open-source code repositories, localizes their code under test, and recovers cross-file code, relative imports, and file resources missing after local extraction, producing standalone executable test–code pairs through real-test execution and multi-stage refinement. The second stage injects functional requirements from real tests into instruction synthesis as functional constraints and verifies regenerated code by executing the original tests; for capability-gap examples where the target model fails but a stronger model succeeds, cross-model code regeneration enables heuristic capability diagnosis to guide target-specific supervision selection. We use Mold-Instruct to construct training data for target models and fine-tune them, collectively referring to the resulting models as MoldCoder.

By adjusting its extensible configurations, we use Mold-Instruct to obtain 102.8K candidate test–code examples from open-source code repositories, retaining 46.7K test–code pairs after execution-guided recovery, multi-stage refinement, and subsequent cleaning. Using these test–code pairs, we construct target-specific training data for Qwen3-4B-Base and SmolLM3-3B-Base and fine-tune the corresponding MoldCoder models. MoldCoder improves all nine evaluated metrics on Qwen3-4B-Base and all seven on SmolLM3-3B-Base, increasing their respective average scores from 45.7 to 51.6 and from 33.1 to 36.9. Further controlled experiments show that real tests outperform target-model-generated tests in our setting, functional constraints and execution verification each provide gains, and capability diagnosis guides supervision selection that further improves performance. Our main contributions are as follows:

- We introduce Mold-Instruct, an automated code instruction data construction method that uses real-test execution as functional supervision.
- Under one extensible configuration, Mold-Instruct constructs 46.7K high-quality test–code pairs from open-source code repositories.
- In our experiments, real tests provide more effective functional supervision than target-model-generated tests.
- Using data constructed by Mold-Instruct, MoldCoder improves both base models across multiple code generation metrics.

2 Related Work

Existing code instruction synthesis methods have improved the scale, complexity, and diversity of training data through large-scale self-generation, instruction evolution, code-based construction, and structured synthesis (Chaudhary, 2023; Luo et al., 2024; Muennighoff et al., 2023; Wei et al., 2024b; Yu et al., 2024; Majumdar et al., 2025; Wu et al., 2024; Xing et al., 2025; Luo et al., 2025). These methods generally do not use test execution results as functional supervision signals.

Several methods use generated tests and execution results for sample scoring, quality analysis, and data selection: XCoder (Wang et al., 2024) scores responses by the number of tests they pass, Self-CodeAlign (Wei et al., 2024a) executes responses against tests and retains passing examples, Open-CodeInstruct (Ahmad et al., 2025) generates tests for synthesized samples and records execution outcomes for sample-quality analysis and filtering, and X-Coder (Wu et al., 2026) uses execution-based verification between tests and candidate solutions to select reliable solutions for supervised fine-tuning, while using verified tests for reinforcement learning. AutoCoder (Lei et al., 2024) and EpiCoder (Wang et al., 2025) further leverage test execution feedback to iteratively debug and refine generated code.

Further methods incorporate generated tests and execution information into the construction of training instructions or tasks. Beyond execution-based verification, KodCode (Xu et al., 2025) uses verified solutions and tests to rewrite natural-language coding questions into completion-style tasks with function signatures and examples, thereby expanding its post-training data. CodeEvo (Sun et al., 2026) uses execution and semantic feedback to adjust the difficulty of subsequent instructions. AutoCodeGen (Zhou et al., 2026) generates code solutions and test inputs, obtains test outputs through execution, and then generates programming problems from the solutions and tests, using public tests as examples and private tests for verification before sampling and quality filtering.

Unlike these methods, MoldCoder recovers developer-written tests and their executable contexts from real repositories, refines them through multiple stages, and uses them as functional supervision to constrain instruction synthesis, while using cross-model reconstruction results for heuristic capability diagnosis to guide training-data selection. Related uses of tests in other code tasks and application settings are discussed in Appendix A.

3 Method

3.1 Executable Test–Code Pair Mining, Recovery, and Refinement

Given a large-scale code corpus from open-source repositories $\mathcal{D}_{\text{raw}}$, our goal is to construct an executable test–code pair dataset $\mathcal{D}_{\text{exec}}$. Real tests are tightly coupled with repository code, dependencies, and resources, so local extraction must address three challenges: localizing test–code relationships, recovering executable context, and ensuring reliable functional supervision. As illustrated in Figure 2, we address these challenges through six stages: Mining and Inlining localize and recover compact test–code relationships, Execution checks executability, Completion recovers missing execution context through execution verification, Refinement improves the reliability of functional supervision, and Cleaning removes contaminated and redundant samples (Appendix B.6); Execution is further reused during Completion and Refinement.

Mining. Rather than mining isolated code snippets, we start from high-confidence real tests and localize their corresponding code, establishing functional supervision directly from real test–code relationships. To scale this process, we use lightweight AST analysis with framework-specific structural signals at multiple granularities, followed by shallow code localization from direct calls in the tests. Same-file calls are resolved to the corresponding code with required local dependencies, while calls through repository-internal imports are retained as cross-file relations for subsequent recovery during Inlining. The complete Mining procedure, its algorithm, Framework Registry, Import Whitelist, and Behavior Blacklist are detailed in Appendix B.1, Algorithm 1, Box B.1, Box B.2, and Box B.3, respectively.

Inlining. The cross-file relations retained by Mining only reference implementations in the original repository rather than self-contained code, while preserving entire related files would introduce

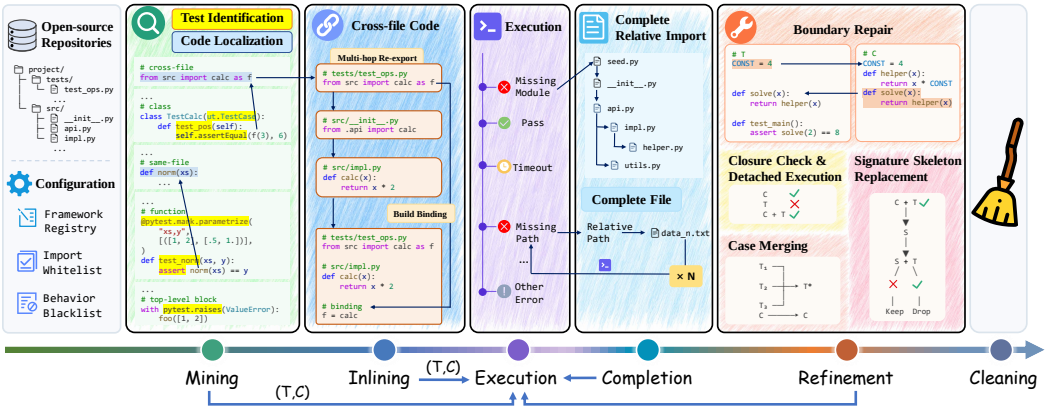


Figure 2: Overview of the executable test–code pair construction pipeline.

substantial irrelevant content. We therefore convert these cross-file relations into explicit and compact test–code pairs. Specifically, we resolve target symbols from import bindings and test calls, follow explicit multi-hop re-export chains when needed, and extract only the target implementations and their required local dependencies. Relocating implementations may break the alias or namespace access paths used by real tests, so we construct corresponding alias or namespace bindings to bridge the tests and inlined implementations. The Inlining procedure and its binding rules are provided in Appendix B.2, Algorithm 2, and Table 6. The downstream benefit of recovered cross-file supervision is evaluated in Appendix E.2.

Execution. Execution is used throughout subsequent stages to continuously verify functional consistency between code and tests. We construct localized execution environments and recover the necessary context when required to preserve package-import semantics or working-directory-dependent file-access semantics. The execution environment is detailed in Appendix B.3.

Completion. Completion recovers relative-import dependencies and file resources severed by local extraction. Relative-import dependencies are recursively traceable through static analysis, so we construct their bounded dependency closure in a single pass. File resources are instead recovered iteratively from missing files exposed by successive executions. Completed candidates are further verified through execution; both procedures are detailed in Appendix B.4.

Refinement. We observe that some test–code pairs still exhibit misaligned boundaries, fragmented functional constraints, or tests that do not genuinely constrain the code implementation. We therefore repair test–code boundaries and merge multiple valid tests for the same code to consolidate fragmented functional constraints. Before merging, we combine static closure checks with detached execution to verify that the code is independently closed and that the tests genuinely depend on the code. After merging, we replace the code with an implementation-stripped interface signature skeleton and re-execute the tests, filtering out spurious supervision that does not require the actual implementation. Detailed refinement procedures are provided in Appendix B.5.

3.2 Instruction–Code Pair Construction

Given the executable test–code pair dataset $\mathcal{D}_{\text{exec}}$ from the first stage and a target model M_t, our goal is to construct a target-specific instruction–code dataset $\mathcal{D}_{\text{inst}}^{(t)}$ using functional supervision from real tests. As illustrated in Figure 3, we inject functional requirements into instruction construction, verify functionality through execution, and select supervision through capability diagnosis. Finally, Cleaning filters low-quality instructions and performs benchmark decontamination, yielding training data for the target model (Appendix B.8).

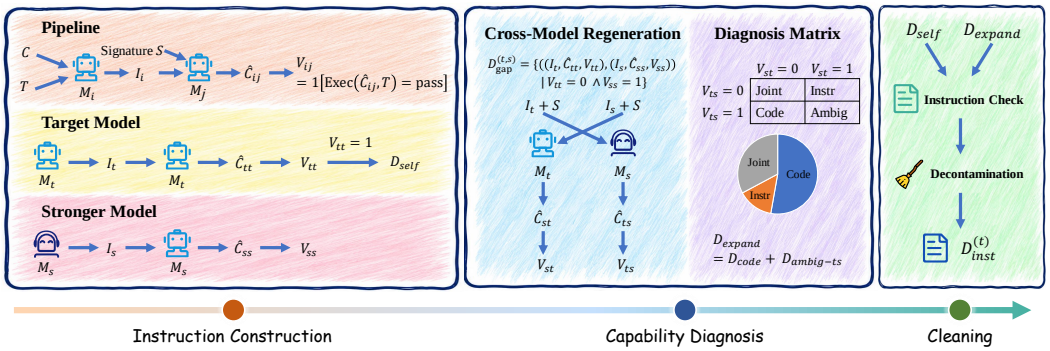


Figure 3: Overview of the instruction–code pair construction pipeline.

Functionality-Constrained Instruction Synthesis. Instructions synthesized from code alone may omit expected behaviors, constraints, and edge cases inferred from tests. Therefore, for each executable test–code pair (T, C), the target model M_t synthesizes an instruction I_t for C, treating T as an additional functional constraint. M_t generates code from I_t and the interface signature skeleton S, allowing at most one execution-guided repair; the final code $\hat{C}_{tt}$ is verified by executing T. Verified instruction–code pairs $(I_t, \hat{C}_{tt})$ form the self-generated dataset $\mathcal{D}^{(t)}_{\text{self}}$, while failures proceed to capability diagnosis. Prompts are provided in Appendix F.

Capability Diagnosis via Cross-Model Code Regeneration. The size of the preceding self-generated dataset provides a natural proxy for the target model's current capability. To perform heuristic capability diagnosis and select expansion supervision, we use a stronger model M_s to repeat the preceding process, obtaining I_s and $\hat{C}_{ss}$. Samples that fail target-model verification but pass stronger-model verification are identified as capability-gap samples. However, since M_s generates both a new instruction and new code, it remains unclear whether the gap arises from instruction synthesis or code generation. We therefore exchange the synthesized instructions for cross-model code regeneration. The stronger model M_s generates $\hat{C}_{ts}$ from I_t, while the target model M_t generates $\hat{C}_{st}$ from I_s; both are verified by executing T. Our heuristic capability diagnosis groups samples into Code, Instr, Joint, and Ambiguous categories based on the two cross-model outcomes. These categories characterize observed failure patterns under the generation-and-repair procedure. The relative sizes of the first three categories form a diagnostic profile, with the largest category determining the supervision direction. We select supervision along this direction and assign Ambiguous samples the corresponding supervision form. Detailed diagnosis and supervision-selection rules are provided in Appendix B.7.

4 Experimental Setup

4.1 Data Construction

Executable Test–Code Pairs. Mold-Instruct is an extensible and configurable automated pipeline. We use the Python subset of The Stack Dedup (Kocetkov et al., 2022)[1] as the raw code corpus. Under the default configuration, after mining and completing repository retrieval and code inlining for cross-file candidates, we select 102.8K candidates for real-test execution. Execution validation and resource recovery retain 67.1K executable examples, and subsequent refinement yields 52.0K examples. Finally, we decontaminate the data against six code-generation benchmarks—HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021), APPS (Hendrycks et al., 2021), DS-1000 (Lai et al., 2022), BigCodeBench (Zhuo et al., 2024), and LiveCodeBench (Jain et al., 2024)—and obtain 46.7K executable test–code pairs after deduplication. Construction configurations governing data yield are detailed in Appendix C.1. The construction funnel, resource recovery, and

[1] https://huggingface.co/datasets/bigcode/the-stack-dedup

refinement analysis are provided in Appendix D.1, D.2, and D.3. Throughout the data construction process, we set a 10-second timeout for every real-test run. The trade-off between execution yield and cost under different timeout settings is analyzed in Appendix E.1.

Instruction–Code Pairs. We use Qwen3-4B-Base (Yang et al., 2025a)[2] and SmolLM3-3B-Base (Bakouch et al., 2025)[3] as target models and GPT-5.6-Luna[4] as the stronger model for cross-model capability diagnosis. Complete data statistics are provided in Appendix D.4.

4.2 Model Training and Evaluation

Baselines. We compare against eight public code instruction datasets: Magicoder-OSS-Instruct (Wei et al., 2024b), XCoder (Wang et al., 2024), SelfCodeAlign (Wei et al., 2024a), Epi-Coder (Wang et al., 2025), KodCode (Xu et al., 2025), OpenCodeInstruct (Ahmad et al., 2025), AutoCodeInstruct (Zhou et al., 2026), and CodeEvo (Sun et al., 2026).

Training Details. We perform full-parameter fine-tuning on Qwen3-4B-Base and SmolLM3-3B-Base and refer to the models fine-tuned on Mold-Instruct as MoldCoder. Qwen3-4B-Base provides a native chat template, so training and evaluation on this model use chat serialization; SmolLM3-3B-Base has no native chat template, so the corresponding experiments use completion serialization. In addition to results reported by the original works, for unified comparisons we fine-tune each released baseline dataset with at most 75K examples using the same base models and training configuration as MoldCoder; full training details are provided in Appendix C.2. Dataset conversion details are provided in Appendix C.4.

Evaluation Protocol. We evaluate HumanEval, HumanEval+, MBPP, and MBPP+ using EvalPlus (Liu et al., 2023); for BigCodeBench, we evaluate the Full and Hard subsets under the Complete and Instruct splits; we additionally evaluate on LiveCodeBench release_v6. We use vLLM for local inference and follow the official evaluation pipelines, executing BigCodeBench in the environment provided by its official Docker image. Across all benchmarks, we set the temperature to 0, generate one sample per problem, and report Pass@1.

5 Results

5.1 Code Generation Performance

Table 1 first summarizes published results and supplementary evaluations of prior work, and then compares different code instruction datasets under the same target backbones, training configuration, and evaluation protocol.

Under the unified fine-tuning setting, MoldCoder improves all evaluated metrics on both target models. On Qwen3-4B-Base, MoldCoder increases the average score from 45.7 to 51.6, achieving the highest average among all training datasets evaluated with this backbone. Although most per-benchmark best scores are achieved by KodCode, OpenCodeInstruct, and AutoCodeInstruct, MoldCoder remains competitive across benchmarks. On SmolLM3-3B-Base, MoldCoder likewise improves all seven evaluated metrics, increasing the average score from 33.1 to 36.9.

5.2 Does Capability Diagnosis Extend Self-Generated Supervision?

Self supervision only covers functionalities that the target model can reconstruct on its own. A stronger model can provide additional capability signals, but whether such supervision can be directly added remains unclear. We first examine naive stronger-model augmentation and then test whether capability diagnosis can select more suitable supervision.

As shown in Table 2, Stronger and Stronger Gap achieve Average scores of 48.9 and 48.8, respectively, both clearly below Self at 51.2, indicating that stronger-model supervision is not suitable as a

[2] https://huggingface.co/Qwen/Qwen3-4B-Base

[3] https://huggingface.co/HuggingFaceTB/SmolLM3-3B-Base

[4] https://openai.com/index/gpt-5-6/

Table 1: Code generation performance (Pass@1, %). Our fine-tuning results are means over repeated runs (Appendix C.2). Avg. is the unweighted mean of nine scores for Qwen3 and seven for SmolLM3. Within each backbone block, bold and underlined values mark the top two distinct scores, with displayed ties sharing a mark; "–" denotes unavailable results.

Baseline Model or Training Data	Size	HE		MBPP		BCB Complete		BCB Instruct		LCB	Avg.
		Base	Plus	Base	Plus	Full	Hard	Full	Hard	V6	
Magicoder-DS-6.7B (Wei et al., 2024b)	75K	66.5	60.4	75.4	61.9	48.7[a]	17.6[a]	38.0[a]	13.5[a]	19.2[a]	44.6
Magicoder-S-DS-6.7B (Wei et al., 2024b)	185K	76.8	70.7	75.7	64.4	47.6[b]	12.8[b]	36.2[b]	13.5[b]	19.1[a]	46.3
StarCoder2-15B-Instruct-v0.1 (Wei et al., 2024a)[c]	50K	72.6	63.4	75.2	61.2	45.1[b]	14.9[b]	37.6[b]	12.2[b]	4.5[a]	43.0
XCoder-8B (Wang et al., 2024)	80K	57.3	53.0	62.2[a]	53.2[a]	32.9[a]	8.8[a]	29.7[a]	10.8[a]	10.6[a]	35.4
EpiCoder-Qwen-7B (Wang et al., 2025)[d]	380K	89.0	82.3	84.1	71.4	51.9	27.7	43.8	22.3	–[d]	–
EpiCoder-DS-6.7B (Wang et al., 2025)[d]	380K	80.5	76.8	81.5	68.3	50.6	19.6	37.9	12.8	–[d]	–
EpiCoder-DS-6.7B-Sample75k (Wang et al., 2025)[d]	75K	78.0	73.2	79.4	68.8	48.2	18.4	35.6	12.8	–[d]	–
KODCODE-32B-SFT-50K (Xu et al., 2025)[e]	50K	92.7	85.4	89.9	76.2	59.8	37.8	51.1	32.4	–[e]	–
KODCODE-32B-SFT-Hard-18K (Xu et al., 2025)[e]	18K	90.9	86.6	89.2	77.0	59.7	37.2	50.5	31.1	–[e]	–
OCI-Qwen2.5-Coder-3B (Ahmad et al., 2025)[f]	5M	84.8	79.7	81.0	69.3	–[f]	–[f]	38.1	–[f]	–[f]	–
OCI-Qwen2.5-Coder-7B (Ahmad et al., 2025)[f]	5M	87.8	84.1	86.8	74.9	–[f]	–[f]	43.6	–[f]	–[f]	–
AutoCodeInstruct-7B-SFT (Zhou et al., 2026)[g]	14K	84.8[g]	79.3[g]	86.2[g]	73.8[g]	53.6[g]	27.0[g]	42.5[g]	22.3[g]	29.4[g]	55.4
CodeEvo-Qwen2.5-Coder-7B-Inst (Sun et al., 2026)	100K	85.7	81.1	84.7	72.5	52.1	24.1	42.8	22.0	28.1	54.8
CodeEvo-Qwen3-8B (Sun et al., 2026)	100K	86.4	81.7	86.2	73.8	55.1	26.1	44.9	17.6	46.9	57.6
Qwen3-4B-Base	–	76.8	70.7	78.0	65.1	47.5	18.9	29.9	14.9	9.6	45.7
+ Magicoder	75K	78.7	72.6	77.5	66.4	45.7	18.2	37.1	13.5	25.8	48.4
+ SelfCodeAlign	50K	76.8	67.7	68.0	59.0	41.3	11.5	35.4	10.8	21.5	43.6
+ XCoder	75K	77.4	71.3	78.6	68.5	45.8	17.6	36.8	14.2	25.1	48.4
+ EpiCoder	75K	79.9	78.0	63.2	53.4	48.5	16.2	39.6	15.5	27.8	46.9
+ KodCode	75K	83.5	77.4	64.0	55.6	47.0	23.0	38.2	**20.9**	24.5	48.2
+ OpenCodeInstruct	75K	82.3	77.4	**80.4**	68.3	**49.6**	18.2	**41.5**	16.2	27.6	51.3
+ AutoCodeInstruct	14K	**88.4**	**82.3**	70.9	60.6	49.5	**25.0**	39.7	18.2	**27.9**	51.4
+ CodeEvo	75K	79.3	72.6	78.6	66.4	48.3	21.6	39.1	18.9	26.8	50.2
+ MoldCoder	32K	84.1	77.4	79.1	**69.3**	48.2	23.6	38.6	16.9	26.8	**51.6**
SmolLM3-3B-Base	–	37.8	31.7	64.0	51.3	29.7	6.1	–	–	10.9	33.1
+ Magicoder	75K	44.5	37.8	63.8	51.1	33.0	4.1	–	–	11.4	35.1
+ SelfCodeAlign	50K	41.5	35.4	64.6	52.1	29.4	6.1	–	–	11.8	34.4
+ XCoder	75K	45.1	38.4	67.2	54.5	32.5	6.1	–	–	10.8	36.4
+ EpiCoder	75K	45.1	39.6	59.0	47.1	34.3	6.8	–	–	10.8	34.7
+ KodCode	75K	47.6	**43.9**	65.1	54.5	**34.6**	9.5	–	–	9.6	37.8
+ OpenCodeInstruct	75K	45.7	41.5	**67.7**	**57.1**	32.4	**10.8**	–	–	**12.6**	**38.3**
+ AutoCodeInstruct	14K	**48.2**	41.5	63.8	53.2	34.4	6.8	–	–	8.8	36.7
+ CodeEvo	75K	39.6	34.8	67.2	56.1	**34.6**	7.4	–	–	8.8	35.5
+ MoldCoder	27K	44.5	39.6	64.6	56.6	30.3	**10.8**	–	–	11.8	36.9

[a] Supplementary evaluations use the authors' publicly released model weights and follow our unified evaluation settings. Unmarked results are from the corresponding original papers or official model releases.
[b] Results from the official BigCodeBench leaderboard.[5]
[c] SelfCodeAlign-CQ-7B does not release its training data or model weights; we therefore report the publicly released StarCoder2-15B-Instruct-v0.1. Its reported LiveCodeBench score uses an earlier release and is therefore omitted.
[d] EpiCoder does not release the reported model weights and does not provide the corresponding training hyperparameters, such as the learning rate and number of training epochs, in either the paper or the released code.
[e] KODCODE does not release the reported model weights and does not specify thresholds for filters such as too long and too short in the paper or released code, preventing exact reconstruction of the 50K and 18K training subsets. KODCODE reports release_v5 LiveCodeBench results only by Easy, Medium, and Hard difficulty, so the LCB column is left unavailable.
[f] OpenCodeInstruct does not release the reported model weights. Its reported models are trained on the full 5M dataset, making reproduction difficult and costly at this scale. OCI-Qwen2.5-Coder-3B and OCI-Qwen2.5-Coder-7B report 31.1 and 39.7, respectively, on release_v4, so these values are not entered in the release_v6 LCB column.
[g] AutoCodeInstruct-7B-SFT is fine-tuned from Qwen2.5-Coder-7B-Instruct using the authors' released 14K AutoCodeInstruct data and reported training hyperparameters with our training pipeline, and evaluated under our unified evaluation protocol. The original paper reports a LiveCodeBench release_v6 score of 17.7.

Table 2: Naive stronger-model augmentation on Qwen3-4B-Base. Size denotes the cleaned training-set size, and Average is the unweighted mean of the nine evaluation scores. Stronger Gap contains stronger-model pairs only for target-failed but stronger-passed cases.

Training Data	Size	HE	HE+	MBPP	MBPP+	BCB-C F	BCB-C H	BCB-I F	BCB-I H	LCB	Avg.
+ Self	20K	85.4	78.7	81.7	69.8	45.4	18.2	37.8	16.9	26.7	51.2
+ Stronger	41K	81.1	73.8	77.0	69.3	47.3	17.6	37.3	13.5	22.9	48.9
+ Stronger Gap	19K	80.5	72.6	76.5	68.3	45.4	18.2	37.9	14.9	24.5	48.8
+ Self + Stronger Gap	39K	81.1	75.6	80.4	69.6	47.1	21.6	38.8	17.6	26.5	50.9

direct replacement for Self. Adding Stronger Gap to Self only slightly decreases the Average to 50.9;

[5] https://bigcode-bench.github.io/

although HE, HE+, and MBPP decrease, all four BCB scores improve, showing that stronger-model gaps contain capability signals not covered by Self but need to be used selectively.

Table 3: Effect of capability diagnosis across supervision directions on Qwen3-4B-Base; Size denotes the cleaned training-set size, and Average is the unweighted mean of the nine evaluation scores.

Training Data	Size	HE	HE+	MBPP	MBPP+	BCB-C F	BCB-C H	BCB-I F	BCB-I H	LCB	Avg.
+ Self	20K	85.4	78.7	81.7	69.8	45.4	18.2	37.8	16.9	26.7	51.2
+ Self + Code	26K	84.1	77.4	79.6	68.3	48.4	21.6	37.7	14.2	27.7	51.0
+ Self + Joint	24K	83.5	76.8	80.4	69.0	47.3	15.5	39.0	18.2	26.5	50.7
+ Self + Instr	21K	84.1	77.4	81.5	69.0	45.3	15.5	37.9	15.5	26.5	50.3
+ Self + Code + Ambig-TS	32K	84.1	77.4	79.1	69.3	48.2	23.6	38.6	16.9	26.8	51.6
+ Self + Joint + Ambig-SS	30K	84.8	77.4	81.2	69.8	46.4	17.6	40.1	16.2	26.6	51.1
+ Self + Instr + Ambig-ST	28K	83.5	76.8	80.7	67.7	44.2	14.9	37.9	16.9	25.8	49.8
+ Self + All Diagnosis-TS	38K	82.9	77.4	79.6	70.1	48.2	20.9	39.2	16.9	26.6	51.3

The raw diagnosis counts in Table 22 show that Qwen3-4B-Base contains 7,591 Code, 4,719 Joint, 2,074 Instr, and 6,959 Ambiguous cases. Code is the largest diagnostic category, with the first three categories ordered Code $>$ Joint $>$ Instr. Adding Code, Joint, and Instr yields Average scores of 51.0, 50.7, and 50.3. Code alone improves BCB-Complete Full/Hard by 3.0/3.4 points over Self, although its Average is 0.2 points lower. Routing the same Ambiguous pool through TS, SS, and ST yields Averages of 51.6, 51.1, and 49.8. The Code-directed combination exceeds Self by 0.4 points overall and 2.8/5.4 points on BCB-Complete Full/Hard, respectively. Further adding Instr and Joint to Self + Code + Ambig-TS yields Self + All Diagnosis-TS; despite using more training data (38K vs. 32K), the Average decreases from 51.6 to 51.3. These results support heuristic capability diagnosis for supervision selection; differing training-set sizes prevent uniquely attributing the gains to diagnostic categories.

SmolLM3-3B-Base contains 14,027 Code, 9,491 Joint, 2,051 Instr, and 6,421 Ambiguous cases, showing a diagnosis profile broadly consistent with Qwen3, with Code again being the largest category. Applying the same Code-directed recipe improves all seven metrics over Self on SmolLM3-3B-Base and increases the Average from 34.3 to 36.9; full results are provided in Appendix E.3.

5.3 What Makes Real-Test Functional Supervision Effective?

To examine how real tests provide effective functional supervision, we conduct three controlled studies on execution verification, functional constraints, and test source, with results summarized in Table 4. To avoid confounding from the stronger-model supervision introduced by capability diagnosis, all studies use the Self setting on Qwen3-4B-Base, whose effectiveness was established in Section 5.2. We further control for code source and style on matched Self data to rule out these factors as alternative explanations for the performance gains (Appendix E.4). Exact dataset sizes before and after cleaning for the experimental settings are reported in Table 23(a–b).

Execution Verification. To test whether real-test execution provides a useful data-selection criterion, we compare Self with Failures Only (failed pairs only), Self + Failures (both passed and failed pairs), and a Random Subset randomly sampled from Self + Failures to match the size of Self. As shown in Table 4(a), Self averages 51.2, substantially above the similarly sized Failures Only set at 49.1, and outperforms it on all nine benchmarks. Self + Failures nearly doubles the training data to 38K, yet its Average drops to 49.5 and it underperforms Self on all nine benchmarks. With training-set size controlled, the 20K Random Subset averages only 49.1 and also trails Self on all nine benchmarks, further showing that data quantity alone does not explain Self's advantage. Together, these results show that real-test execution provides a useful data-selection signal for retaining more reliable functional supervision.

Functional Constraints. To isolate the contribution of test-derived functional constraints, we keep real-test verification unchanged and remove the test input only from instruction synthesis in the w/o Test Injection control; the corresponding instruction-construction settings are detailed in Table 12.

Table 4: Controlled studies of real-test supervision on Qwen3-4B-Base: (a) execution verification, (b) functional constraints, and (c) test source. Self and Real-Full refer to the same shared baseline. Full uses the complete data from each test source; Intersection matches cases by ID and file relation. Size denotes the cleaned training-set size, and Avg. is the mean of the nine benchmark scores.

Training Data	Size	HE	HE+	MBPP	MBPP+	BCB-C F	BCB-C H	BCB-I F	BCB-I H	LCB	Avg.
Self / Real-Full	20K	85.4	78.7	81.7	69.8	45.4	18.2	37.8	16.9	26.7	51.2
(a) Execution Verification											
Failures Only	19K	83.5	76.8	77.0	67.5	44.1	16.2	37.4	13.5	25.9	49.1
Self + Failures	38K	84.1	78.0	79.9	68.5	44.2	13.5	36.8	14.2	26.0	49.5
Random Subset	20K	82.9	75.6	79.6	67.7	44.9	14.9	37.5	13.5	25.6	49.1
(b) Functional Constraints											
w/o Test Injection	20K	82.9	76.8	80.7	67.5	46.2	21.6	37.3	15.5	26.9	50.6
(c) Test Source											
Gen-Full	9K	84.1	77.4	81.0	68.8	47.2	18.2	37.0	14.9	26.7	50.6
Real-Intersection	6K	82.9	76.2	81.7	69.6	47.3	20.3	38.0	18.9	26.4	51.3
Gen-Intersection	6K	82.9	76.8	81.0	68.5	46.6	18.2	36.9	15.5	26.8	50.4

Table 4(b) shows that Self outperforms w/o Test Injection on most benchmarks, increasing the Average from 50.6 to 51.2. This additional overall gain indicates that test-derived functional constraints complement execution verification.

Test Source. To compare test sources, we replace real tests with target-model-generated tests, verify them on the original code, and follow the same construction pipeline; we report both the complete data from each source and Intersection data strictly matched by case ID and file relation. In the full pipeline, Real Test yields 20K usable examples with an Average of 51.2, compared with 9K examples and 50.6 for Gen Test; on the matched 6K Intersection sets, it still leads by 0.9 points (51.3 vs. 50.4). Thus, the advantage of real tests is not explained only by their higher data yield.

To further compare the functional checking strength of the tests themselves, we mutate the original code for the 6,466 strictly matched cases, sample up to 20 mutants per case, and execute the same mutants with both test sources. Real Test reaches an 81.95% mutation score, 8.63 points above Gen Test at 73.32%, while the Real-only kill rate is 14.91% versus 6.28% Gen-only. These paired results show that real tests detect more mutations missed by target-model-generated tests in this setting. Detailed experimental settings and complete results for the paired mutation analysis are provided in Appendix E.5.

6 Conclusion

In this work, we introduce Mold-Instruct, an automated code instruction data construction method that uses real-test execution as functional supervision. By recovering standalone executable test–code pairs, Mold-Instruct uses functional constraints to guide instruction synthesis and combines execution verification with capability diagnosis to construct target-specific training data. Our experiments show that MoldCoder improves code generation across base models, real tests outperform target-model-generated tests in our setting, and functional constraints and execution verification each provide gains. Furthermore, heuristic capability diagnosis guides target-specific supervision selection, yielding additional gains especially on BigCodeBench-Complete. These findings highlight that effective code instruction data construction depends not only on the source and utilization of functional supervision, but also on its alignment with the target model's capability needs.

7 AI Use Statement

AI tools were used in two distinct roles in this work. First, large language models are explicit components of the proposed method, including functionality-constrained instruction synthesis, code generation and execution-guided repair, and cross-model capability diagnosis; test generation and

repair are additionally used in the corresponding generated-test control experiments. These models and their specific uses are explicitly described in Section 3 and Section 4.

Second, AI was used as a research assistant for literature retrieval, coding assistance, experimental data analysis, and manuscript polishing. The core research ideas and methodological design of this work were independently conceived by the authors. AI was used to assist with literature retrieval, while the relevant papers were independently read and verified by the authors. During the validation and implementation of the proposed ideas, AI was selectively used to assist with coding. AI-assisted code was developed under strict coding standards and was manually reviewed, verified, and refactored when necessary by the authors. AI was used to assist with the organization and analysis of experimental data, while the reported results and research conclusions were checked and confirmed by the authors against the original experimental data. The manuscript was drafted by the authors, with AI used only for local language polishing.

8 Ethics Statement

This work does not involve human subjects or intentionally collect private user data. We use the Python subset of The Stack Dedup (Kocetkov et al., 2022) as the raw code corpus and access corresponding public source-repository snapshots only when required for cross-file inlining and resource completion. For accessed repository snapshots, our mining pipeline preserves repository-level provenance metadata, including the repository name, exact revision, and repository-relative file path. These metadata support source traceability, attribution, and license-aware release decisions. Because source code remains subject to the licenses of its originating repositories, any release of derived datasets or redistributed source-derived artifacts will follow the applicable source licenses and the terms of the underlying datasets.

The Mold-Instruct data-construction pipeline executes tests and code recovered from public repositories, so running untrusted third-party code can introduce security risks. Before execution, samples are statically filtered using an import allowlist and behavior blacklist; during execution, each case is subject to a timeout, after which the corresponding process group is terminated and cleaned up. These mechanisms are intended to reduce execution risks and improve runtime robustness, but they do not constitute a complete security containment boundary. Reproduction of the data-construction pipeline should therefore be conducted in an isolated environment, avoid exposing sensitive credentials to executed code, and restrict file-system access and other operating-system privileges.

MoldCoder may inherit limitations and unsafe behaviors from its base models and source-code corpus, while LLM-based data construction may also introduce model-dependent biases. Our current study is limited to Python code generation and supervised instruction tuning of 3B–4B base models; whether the findings generalize to other programming languages, software-engineering tasks, larger model scales, or other training paradigms remains to be established.

9 Reproducibility Statement

During the review period, we provide an anonymous repository containing the complete code for executable test–code pair construction, instruction–code pair construction, model fine-tuning, and model evaluation, together with a subset of the data. The anonymous repository is available at `https://anonymous.4open.science/r/moldcoder-anno-repo/`. After the review process, we plan to publicly release the project as open source.

We provide detailed implementation descriptions of the proposed method in the appendices, including the key algorithms and technical procedures. Experimental details, including data-construction configurations, generation parameters, training and evaluation settings, and computing environments, are also documented in the main paper and appendices.

Cross-model capability diagnosis depends on GPT-5.6-Luna accessed through an API, so exact reproduction of its generations may be affected if the server-side model implementation or version changes over time. In addition, large language models rely on probabilistic generation, so generated outputs may vary across runs even under the same model and generation configuration. Therefore, steps involving LLM generation may not be exactly reproducible at the individual-sample level.

REFERENCES

Wasi Uddin Ahmad, Aleksander Ficek, Mehrzad Samadi, Jocelyn Huang, Vahid Noroozi, Somshubra Majumdar, and Boris Ginsburg. Opencodeinstruct: A large-scale instruction tuning dataset for code llms, 2025. URL https://arxiv.org/abs/2504.04030.

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.

Elie Bakouch, Loubna Ben Allal, Anton Lozhkov, Nouamane Tazi, Lewis Tunstall, Carlos Miguel Patiño, Edward Beeching, Aymeric Roucher, Aksel Joonas Reedi, Quentin Gallouédec, Kashif Rasul, Nathan Habib, Clémentine Fourrier, Hynek Kydlicek, Guilherme Penedo, Hugo Larcher, Mathieu Morlon, Vaibhav Srivastav, Joshua Lochner, Xuan-Son Nguyen, Colin Raffel, Leandro von Werra, and Thomas Wolf. SmolLM3: smol, multilingual, long-context reasoner. https://huggingface.co/blog/smollm3, 2025.

Sahil Chaudhary. Code alpaca: An instruction-following llama model for code generation. https://github.com/sahil280114/codealpaca, 2023.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. Codet: Code generation with generated tests, 2022. URL https://arxiv.org/abs/2207.10397.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021. URL https://arxiv.org/abs/2107.03374.

Kevin Han, Siddharth Maddikayala, Tim Knappe, Om Patel, Austen Liao, and Amir Barati Farimani. Tdflow: Agentic workflows for test driven development, 2026. URL https://arxiv.org/abs/2510.23761.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring coding challenge competence with apps. *NeurIPS*, 2021.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint*, 2024.

Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, Leandro von Werra, and Harm de Vries. The stack: 3 tb of permissively licensed source code. *Preprint*, 2022.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Scott Wen tau Yih, Daniel Fried, Sida Wang, and Tao Yu. Ds-1000: A natural and reliable benchmark for data science code generation. *ArXiv*, abs/2211.11501, 2022.

Bin Lei, Yuchen Li, and Qiuwu Chen. Autocoder: Enhancing code large language model with AIEV-INSTRUCT, 2024.

Junlong Li, Daya Guo, Dejian Yang, Runxin Xu, Yu Wu, and Junxian He. Codei/o: Condensing reasoning patterns via code input-output prediction. *arXiv preprint arXiv:2502.07316*, 2025a.

Wei Li, Xin Zhang, Zhongxin Guo, Shaoguang Mao, Wen Luo, Guangyue Peng, Yangyu Huang, Houfeng Wang, and Scarlett Li. Fea-bench: A benchmark for evaluating repository-level code generation for feature implementation, 2025b. URL https://arxiv.org/abs/2503.06680.

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. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=1qvx610Cu7.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. Wizardcoder: Empowering code large language models with evol-instruct. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=UnUwSIgK5W.

Ziyang Luo, Kaixin Li, Hongzhan Lin, Yuchen Tian, Mohan Kankanhalli, and Jing Ma. Tree-of-evolution: Tree-structured instruction evolution for code generation in large language models. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 297–316, Vienna, Austria, July 2025. Association for Computational Linguistics. ISBN 979-8-89176-251-0. URL https://aclanthology.org/2025.acl-long.14/.

Yichuan Ma, Yunfan Shao, Peiji Li, Demin Song, Qipeng Guo, Linyang Li, Xipeng Qiu, and Kai Chen. UnitCoder: Scalable code synthesis from pre-training corpora. In Christos Christodoulopoulos, Tanmoy Chakraborty, Carolyn Rose, and Violet Peng (eds.), *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 5623–5641, Suzhou, China, November 2025. Association for Computational Linguistics. ISBN 979-8-89176-332-6. doi: 10.18653/v1/2025.emnlp-main.286. URL https://aclanthology.org/2025.emnlp-main.286/.

Somshubra Majumdar, Vahid Noroozi, Mehrzad Samadi, Sean Narenthiran, Aleksander Ficek, Wasi Uddin Ahmad, Jocelyn Huang, Jagadeesh Balam, and Boris Ginsburg. Genetic instruct: Scaling up synthetic generation of coding instructions for large language models, 2025. URL https://arxiv.org/abs/2407.21077.

Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro von Werra, and Shayne Longpre. Octopack: Instruction tuning code large language models. *arXiv preprint arXiv:2308.07124*, 2023.

Yunfan Shao, Linyang Li, Yichuan Ma, Peiji Li, Demin Song, Qinyuan Cheng, Shimin Li, Xiaonan Li, Pengyu Wang, Qipeng Guo, et al. Case2code: Scalable synthetic data for code generation. *arXiv preprint arXiv:2407.12504*, 2024.

Bo Shen, Jiaxin Zhang, Taihong Chen, Daoguang Zan, Bing Geng, An Fu, Muhan Zeng, Ailun Yu, Jichuan Ji, Jingyang Zhao, Yuenan Guo, and Qianxiang Wang. Pangu-coder2: Boosting large language models for code with ranking feedback, 2023. URL https://arxiv.org/abs/2307.14936.

Qiushi Sun, Jingyang Gong, Lei Li, Qipeng Guo, and Fei Yuan. Codeevo: Interaction-driven synthesis of code-centric data through hybrid and iterative feedback. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 9668–9687, 2026.

Yaoxiang Wang, Haoling Li, Xin Zhang, Jie Wu, Xiao Liu, Wenxiang Hu, Zhongxin Guo, Yangyu Huang, Ying Xin, Yujiu Yang, Jinsong Su, Qi Chen, and Scarlett Li. Epicoder: Encompassing diversity and complexity in code generation, 2025. URL https://arxiv.org/abs/2501.04694.

Yejie Wang, Keqing He, Dayuan Fu, Zhuoma GongQue, Heyang Xu, Yanxu Chen, Zhexu Wang, Yujia Fu, Guanting Dong, Muxi Diao, Jingang Wang, Mengdi Zhang, Xunliang Cai, and Weiran Xu. How do your code LLMs perform? empowering code instruction tuning with really good data. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (eds.), *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 14027–14043, Miami, Florida, USA, November 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.emnlp-main.777. URL https://aclanthology.org/2024.emnlp-main.777/.

Yuxiang Wei, Federico Cassano, Jiawei Liu, Yifeng Ding, Naman Jain, Zachary Mueller, Harm de Vries, Leandro von Werra, Arjun Guha, and Lingming Zhang. Selfcodealign: Self-alignment for code generation. *arXiv preprint arXiv:2410.24198*, 2024a.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. Magicoder: Empowering code generation with OSS-instruct. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 52632–52657. PMLR, 21–27 Jul 2024b. URL https://proceedings.mlr.press/v235/wei24h.html.

Jie Wu, Haoling Li, Xin Zhang, Jiani Guo, Jane Luo, Xuewei Yang, Steven Liu, Yangyu Huang, Ruihang Chu, Scarlett Li, and Yujiu Yang. X-coder: Advancing competitive programming with synthetic tasks, solutions, and tests, 2026. URL https://arxiv.org/abs/2601.06953.

Yutong Wu, Di Huang, Wenxuan Shi, Wei Wang, Lingzhe Gao, Shihao Liu, Ziyuan Nan, Kaizhao Yuan, Rui Zhang, Xishan Zhang, Zidong Du, Qi Guo, Yewen Pu, Dawei Yin, Xing Hu, and Yunji Chen. Inversecoder: Self-improving instruction-tuned code llms with inverse-instruct, 2024. URL https://arxiv.org/abs/2407.05700.

Wenjing Xing, Wenke Lu, Yeheng Duan, Bing Zhao, Zhenghui kang, Yaolong Wang, Kai Gao, and Lei Qiao. Infinite-instruct: Synthesizing scaling code instruction data with bidirectional synthesis and static verification, 2025. URL https://arxiv.org/abs/2505.23177.

Zhangchen Xu, Yang Liu, Yueqin Yin, Mingyuan Zhou, and Radha Poovendran. Kodcode: A diverse, challenging, and verifiable synthetic dataset for coding, 2025. URL https://arxiv.org/abs/2503.02951.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025a.

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. In *Proceedings of the 39th Annual Conference on Neural Information Processing Systems (NeurIPS 2025 D&B Spotlight)*, 2025b. URL https://arxiv.org/abs/2504.21798. arXiv:2504.21798, accepted at NeurIPS 2025 (Spotlight).

Zhaojian Yu, Xin Zhang, Ning Shang, Yangyu Huang, Can Xu, Yishujie Zhao, Wenxiang Hu, and Qiufeng Yin. WaveCoder: Widespread and versatile enhancement for code large language models by instruction tuning. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 5140–5153, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.280. URL https://aclanthology.org/2024.acl-long.280/.

Lei Zhang, Jiaxi Yang, Min Yang, Jian Yang, Mouxiang Chen, Jiajun Zhang, Zeyu Cui, Binyuan Hui, and Junyang Lin. Swe-flow: Synthesizing software engineering data in a test-driven manner, 2025. URL https://arxiv.org/abs/2506.09003.

Changzhi Zhou, Ao Liu, Yuchi Deng, Zhiying Zeng, Tao Zhang, Haotian Zhu, Jianwei Cai, Yue Mao, Chenchen Zhang, Lingyun Tan, ZiyanXU, Bohui Zhai, HengyiLIu, Speed Zhu, Wiggin Zhou, and Fengzong Lian. Autocodebench: Large language models are automatic code benchmark generators. In C. Vondrick, B. Hariharan, C. Raffel, L. Pinto, D. Yang, and A. Faust (eds.), *International Conference on Learning Representations*, volume 2026, pp. 79116–79148, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/file/7fb7cbd43263f65f485ae12cb3698a7c-Paper-Conference.pdf.

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

A Additional Related Work

Uses of Tests in Related Tasks. CodeT (Chen et al., 2022) is an early work that uses generated tests to assist code generation, executing candidate solutions against generated tests and selecting solutions based on execution agreement at inference time. PanGu-Coder2 (Shen et al., 2023) further incorporates test feedback into training, ranking candidate responses using test results and teacher feedback and optimizing the model with ranking feedback. CodeI/O (Li et al., 2025a) constructs input-output prediction tasks from code and test cases, using natural-language CoT to learn general reasoning patterns embedded in programs. Case2Code (Shao et al., 2024) generates program inputs and obtains outputs through execution, training models to infer code implementations from the resulting input-output behaviors. UnitCoder (Ma et al., 2025) generates unit tests for code from pretraining corpora, iteratively refines code based on execution results, and constructs completion-style post-training data. These works use tests and execution-derived behaviors as functional supervision for different code tasks, but do not directly target the construction of code instruction-tuning data.

Uses of Tests in Repository-Level Software Engineering. FEA-Bench (Li et al., 2025b) constructs feature-implementation evaluation tasks from real GitHub pull requests and pairs them with relevant unit tests for verification. SWE-smith (Yang et al., 2025b) synthesizes bug-fix tasks in real codebases by introducing changes that break existing tests, validates them through execution, and uses them to train software-engineering agents. SWE-Flow (Zhang et al., 2025) treats unit tests as high-level requirements, infers incremental development steps from them, and constructs verifiable TDD tasks. TDFlow (Han et al., 2026) formulates repository-scale software engineering as solving human-written tests and iteratively debugs and revises code patches based on test feedback. These works leverage real tests from open-source repositories for functional verification, but their execution typically relies on relatively complete repository-level software-engineering environments and primarily targets the training and evaluation of repository-level software-engineering tasks rather than the construction of instruction-tuning data for code generation.

B Method Details

B.1 Mining

The complete Mining procedure is summarized in Algorithm 1, followed by details of its key components.

High-Confidence Test Identification. First, we perform lightweight file-level screening based on registered testing-framework imports or native assertions. Structural signals such as test base classes, decorators, and calls are first resolved through the file's import bindings and then matched against the import_startswith rules configured in the framework registry. Candidates with native assertions but no matched framework are handled by the builtin entry. The framework registry separates framework-specific configuration from the generic Mining procedure by maintaining import-matching rules, test-candidate selection rules, and execution-entry templates in a unified interface, enabling the same procedure to support heterogeneous testing frameworks. Representative entries

Algorithm 1 Executable Test–Code Pair Mining.

Require: Source file S, repository context R, Framework Registry $\mathcal{F}$ ▷ Box B.1
Import Whitelist $\mathcal{W}$, Behavior Blacklist $\mathcal{B}$ ▷ Box B.2, Box B.3
Ensure: Mined test–code pairs $\mathcal{P}$

$\mathcal{P} \leftarrow \emptyset$
$I \leftarrow \textsc{ImportBindings}(S)$
$\mathcal{T} \leftarrow \textsc{IdentifyTests}(S, I, \mathcal{F})$ ▷ Test Identification
$\mathcal{D} \leftarrow \textsc{DefinitionMap}(S, \mathcal{T})$
for each $t \in \mathcal{T}$ **do**
  $(c, r) \leftarrow \textsc{LocalizeCode}(t, \mathcal{D}, I, R, \mathcal{W})$ ▷ Code Localization
  $u \leftarrow \textsc{BuildRunner}(t, \mathcal{F})$
  $(c, t, I) \leftarrow \textsc{RecoverContext}(c, t, I, r)$ ▷ Context Recovery
  $I \leftarrow \textsc{PruneImports}(I, c, t, u)$
  if $\neg\textsc{AllowedImports}(I, \mathcal{W})$ **then continue** ▷ Dependency Filtering
  if $\textsc{ResolveCalls}(c, t, I) \cap \mathcal{B} \neq \emptyset$ **then continue** ▷ Behavior Filtering
  $\mathcal{P} \leftarrow \mathcal{P} \cup \{\textsc{Assemble}(I, c, t, u, r)\}$
end for
return $\mathcal{P}$

are shown in Box B.1. **Second**, to cover heterogeneous test organization in real repositories, candidates are extracted at class, function, and top-level block granularities. Blocks are formed from contiguous top-level statements, allowing top-level assertions and test logic that are not wrapped in test functions to remain discoverable.

Box B.1: Representative Entries of the Framework Registry

```
"unittest": {
    "name": "unittest",
    "import_startswith": ("unittest", "unittest2"),
    "test_rules": {
        "enable_name_filter": True,
        "class_prefixes": ("Test",),
        "method_prefixes": ("test",),
        "function_prefixes": ("test",),
        "enable_decorator_ignore": True,
        "decorator_ignore": ("skip", "skipIf", "skipUnless", "expectedFailure"),
    },
    "runner_templates": {
        "class": "unittest.defaultTestLoader.loadTestsFromTestCase(...)",
        "function": "unittest.FunctionTestCase(...)",
        "block": "unittest.defaultTestLoader.loadTestsFromModule(...)",
    },
},
"pytest": {...},
"nose": {...},
"nose2": {...},
"hypothesis": {...},
"testtools": {...},
"twisted.trial": {...},
"ward": {...},
"builtin": {
    "name": "builtin",
    "import_startswith": (),
    "test_rules": {
        "enable_name_filter": True,
        "class_prefixes": (),
        "method_prefixes": ("test", "check", "verify"),
        "function_prefixes": ("test", "check", "verify"),
        "enable_decorator_ignore": False,
        "decorator_ignore": (),
    },
    "runner_templates": {
        "class": "{class_name}(); ...; _fn()",
        "function": "{func_name}()",
        "block": "pass",
    },
},
```

Test-Guided Code Localization. We localize the corresponding code under test from direct calls in each test. We intentionally restrict this stage to first-order localization based on direct calls, without constructing a complete call graph or performing deeper program analysis. This lightweight strategy is suitable for large-scale corpus mining, while the required local dependencies and functional validity are addressed by subsequent context recovery and real execution validation, respectively. For same-file candidates, direct calls locate code in the same file after definitions belonging to the identified test candidate are excluded. For cross-file candidates, direct calls through imports identify repository-internal imports, including relative imports and absolute imports resolvable within the source repository, while the actual code is recovered during subsequent Inlining. The exact rules are summarized in Table 5.

Local-Context Recovery. To recover the local context required by the test and code, we recursively retrieve same-file supporting statements based on unresolved names in the currently resolved source. For same-file candidates, local dependencies are recovered for the code and test in sequence. For cross-file candidates, test-side dependencies are recovered from the test file before Inlining, while the same procedure is applied after Inlining to recover code-side dependencies. Finally, imports are pruned according to the names actually used, retaining only the required imports.

Table 5: Same-file and cross-file code-localization rules.

Relation	Mining output	Evidence	Filtering criteria
Same-file	Same-file code	Direct calls to local definitions	At least one matched code.
Cross-file	Repository-internal imports	Direct calls through imports	1. Empty same-file code. 2. At least one repository-internal import. 3. Whitelisted external imports are ignored; other external imports cause the candidate to be skipped.

Dependency and Behavior Filtering. Beyond basic syntactic validity, we further constrain execution dependencies and potentially unsafe behavior through import-level and call-level filtering. Import-level filtering applies the Import Whitelist to the retained imports, allowing only dependencies supported by the unified execution environment. Call-level filtering examines actual function calls and resolves aliases through import bindings before exact matching against the Behavior Blacklist. Representative configurations of the Import Whitelist and Behavior Blacklist are shown in Boxes B.2 and B.3, respectively.

Box B.2: Representative Entries of the Import Whitelist

```
IMPORT_WHITELIST = {
    # 1) Core foundations and algorithms
    "typing", "collections", "math", "json", "pathlib", "ast", ...
    # 2) Concurrency and scheduling
    "threading", "queue", "concurrent", "multiprocessing", "asyncio", "sched", ...
    # 3) Network communication
    "socket", "select", "ssl", "urllib", "http", "email", ...
    # 4) Data science and machine learning
    "numpy", "pandas", "scipy", "sklearn", "statsmodels", "sympy", ...
    # 5) Deep learning and artificial intelligence
    "torch", "torchvision", "torchaudio", "transformers", "datasets", "huggingface_hub",
...
    # 6) Natural language processing and text processing
    "bs4", "soupsieve", "lxml", "regex", "chardet", "wordninja", ...
    # 7) Image processing and visualization
    "PIL", "cv2", "skimage", "imageio", "matplotlib", "seaborn", ...
    # 8) Modern engineering tools
    "dateutil", "pytz", "pydantic", "tqdm", "yaml", "packaging", ...
    # 9) Testing frameworks
    "unittest", "pytest", "nose", "hypothesis", "twisted", "ward", "mock", ...
}
```

Box B.3: Representative Entries of the Behavior Blacklist

```
BEHAVIOR_BLACKLIST = {
    # 1) Core foundations and algorithms
    "input", "os.remove", "os.system", "pathlib.Path.unlink", "shutil.rmtree", ...
    # 2) Concurrency and scheduling
    # No active entries.
    # 3) Network communication
    "socket.socket", "socket.create_connection", "urllib.request.urlopen",
    "http.client.HTTPConnection", "twisted.internet.reactor.listenTCP", ...
    # 4) Data science and machine learning
    "pandas.read_sql", "pandas.read_clipboard", "sklearn.datasets.fetch_openml",
    "statsmodels.datasets.get_rdataset", ...
    # 5) Deep learning and artificial intelligence
    "transformers.AutoModel.from_pretrained", "datasets.load_dataset",
    "huggingface_hub.hf_hub_download", "torch.hub.load", ...
    # 6) Natural language processing and image processing
    "cv2.VideoCapture", "matplotlib.pyplot.show", "seaborn.load_dataset", ...
    # 7) Python 2 legacy
    "raw_input", "xrange", "basestring", "execfile", "dict.iteritems", "operator.div", ...
}
```

B.2 INLINING

The complete Inlining procedure is summarized in Algorithm 2.

Algorithm 2 Cross-File Code Inlining.

Require: Test t, Cross-file code c, Repository context R
Ensure: Code c', Binding b

$I \leftarrow \textsc{ImportBindings}(c)$
$K \leftarrow \textsc{TestCalls}(t)$
$\mathcal{X} \leftarrow \textsc{ResolveTargets}(I, K)$ ▷ $x \in \mathcal{X}$: module, symbol, access type, and access path
$\mathcal{Z} \leftarrow \emptyset$
for each target $x \in \mathcal{X}$ **do**
    $S \leftarrow \textsc{LocateSource}(x.\text{module}, R)$
    $y \leftarrow x.\text{symbol}$
    while y is not defined in S **do**
        $(S, y) \leftarrow \textsc{FollowReExport}(S, y, R)$
    end while
    $\mathcal{Z} \leftarrow \mathcal{Z} \cup \{(S, y, x)\}$
end for
$r \leftarrow$ same-file
$\mathcal{C} \leftarrow \emptyset$
for each $S \in \textsc{SourceSet}(\mathcal{Z})$ **do**
    $Y_S \leftarrow \textsc{SymbolSet}(\mathcal{Z}, S)$
    $d_S \leftarrow \textsc{Definitions}(S, Y_S)$
    $I_S \leftarrow \textsc{ImportBindings}(S)$
    $(c_S, _, I_S) \leftarrow \textsc{RecoverContext}(d_S, \emptyset, I_S, r)$ ▷ Context Recovery
    $\mathcal{C} \leftarrow \mathcal{C} \cup \{c_S\}$
end for
$c' \leftarrow \textsc{Merge}(\mathcal{C})$
$b \leftarrow \textsc{BuildBindings}(\mathcal{Z})$ ▷ Table 6
return (c', b)

Using the repository location of the original test file as an anchor, the corresponding source file of the target module is located, either as a module file or a package initializer. If the target symbol is not directly defined in that source, tracing follows only matching explicit top-level from-import re-exports.

The localized code recovered from each source cannot retain relative imports, because flattening multiple source files into a single local module cannot simultaneously preserve their original source paths and package contexts.

Inlining removes the original module hierarchy, so direct-alias or namespace bindings are constructed from the preserved test-side access paths to keep the original test calls unchanged. Representative binding patterns are shown in Table 6. For brevity, NS() denotes an empty namespace instance used to construct module hierarchies.

Table 6: Representative original imports, test calls, and constructed bindings after inlining.

Original Import	**Test Call**	**Constructed Binding**
from pkg.mod import f as g	g(...)	g = f
import pkg.mod as m	m.f(...)	m = NS() m.f = f
import pkg.mod	pkg.mod.C.foo(...)	pkg = NS() pkg.mod = NS() pkg.mod.C = C

B.3 Execution

Package-relative imports depend on the original package context, whereas relative file accesses depend on the execution working directory. The localized execution environment therefore preserves these two contexts separately. For candidates with relative imports, the executable entry is placed under its corresponding source path and launched through a lightweight wrapper in the runtime directory to preserve package semantics. File resources are instead placed relative to the runtime directory according to their original access paths. Figure 4 illustrates a candidate with both a relative import and a relative file access, whose resources are placed according to the source-package and runtime-directory contexts, respectively.

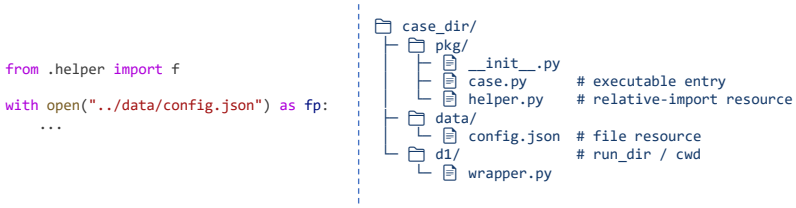


Figure 4: An example localized execution environment preserving both package context and working-directory context.

B.4 Completion

Relative-Import Resource Completion. Relative-import completion uses the original source file as the initial anchor. For each relative import, candidate repository paths are constructed from its level and import target, and the first available source file in the repository snapshot is located from each candidate group. Each newly recovered source file becomes the anchor for the next level of resolution until the queue is exhausted or the maximum depth, file count, or total size is reached. The complete procedure is given in Algorithm 3.

File-Resource Completion. The missing path reported by execution is first converted into a runtime-relative path using the actual working directory, and then combined with the original source

Algorithm 3 Bounded Completion of Relative-Import Resources.

Require: Test–code pair p, Source file S, Repository snapshot R
Max depth $D_{\max}$, Max files $N_{\max}$, Max total size $B_{\max}$
Ensure: Relative-import resources $\mathcal{R}_{\mathrm{RI}}$

$\mathcal{R}_{\mathrm{RI}} \leftarrow \emptyset$
$Q \leftarrow \{(S, ri, 0) \mid ri \in \textsc{RelativeImports}(p)\}$
$V \leftarrow \emptyset$, $b \leftarrow 0$
while $Q \neq \emptyset$ **do**
  $(\hat{S}, ri, d) \leftarrow \textsc{Pop}(Q)$
  if $d > D_{\max}$ or $(\hat{S}, ri) \in V$ **then continue**
  $V \leftarrow V \cup \{(\hat{S}, ri)\}$
  for each $\Pi \in \textsc{ImportPathCandidates}(\hat{S}, ri)$ **do**
    $S' \leftarrow \textsc{LocateFirst}(\Pi, R)$
    if $S' = \emptyset$ **then continue**
    if $|\mathcal{R}_{\mathrm{RI}}| \geq N_{\max}$ or $b + \textsc{Size}(S') > B_{\max}$ **then continue**
    $\mathcal{R}_{\mathrm{RI}} \leftarrow \mathcal{R}_{\mathrm{RI}} \cup \{S'\}$
    $b \leftarrow b + \textsc{Size}(S')$
    for each $ri' \in \textsc{RelativeImports}(S')$ **do**
      $\textsc{Push}(Q, (S', ri', d+1))$
    end for
  end for
end while
return $\mathcal{R}_{\mathrm{RI}}$

location to locate the corresponding file in the repository snapshot. For paths containing parent-directory traversals, the original relative-path relation is preserved by adjusting the runtime-directory depth and resource placement. File resources are recovered across successive executions until execution succeeds or completion can no longer proceed. The complete procedure is given in Algorithm 4.

Algorithm 4 Execution-guided Multi-round Completion of File Resources.

Require: Test–code pair p, Source file S, Repository snapshot R
Max rounds $M_{\max}$, Max parent traversals $U_{\max}$
Ensure: File resources $\mathcal{R}_{\mathrm{F}}$

$\mathcal{R}_{\mathrm{F}} \leftarrow \emptyset$
$d \leftarrow 0$
$e \leftarrow \textsc{ExecResult}(p)$
for $m = 1, \ldots, M_{\max}$ **do**
  $\rho \leftarrow \textsc{MissingPath}(e)$
  if $\rho = \emptyset$ **then break**
  $\rho \leftarrow \textsc{RuntimeRelativePath}(\rho, \textsc{RunDir}(e))$
  if $\rho = \emptyset$ **then break**
  $(u, \bar{\rho}) \leftarrow \textsc{SplitUpLevels}(\rho)$
  if $u > U_{\max}$ or $\bar{\rho}$ is unsafe or $\bar{\rho}$ is empty **then break**
  $d \leftarrow \max(d, u)$
  if $\rho \notin \textsc{Paths}(\mathcal{R}_{\mathrm{F}})$ **then**
    $\Pi \leftarrow \textsc{FilePathCandidates}(S, \rho)$
    $F \leftarrow \textsc{LocateFirst}(\Pi, R)$
    if $F = \emptyset$ **then break**
    $\mathcal{R}_{\mathrm{F}} \leftarrow \mathcal{R}_{\mathrm{F}} \cup \{(\rho, F)\}$
  end if
  $e \leftarrow \textsc{Execute}(p, \mathcal{R}_{\mathrm{F}}, d)$
  if $\textsc{Passed}(e)$ **then break**
end for
return $\mathcal{R}_{\mathrm{F}}$

B.5 REFINEMENT

Boundary Repair. Boundary repair first uses Ruff F821 diagnostics to detect undefined names in the code and moves the corresponding top-level definitions from the test back to the code, recovering code dependencies misplaced by the initial split. We then identify real tests and high-confidence test-support units among top-level units from both sides, and propagate test-side roles along definition–reference relations to move test-related units misplaced in the code back to the test. We retain only repairs consistent with the existing binding and whose repaired test–code pairs pass re-execution. The complete procedure is given in Algorithm 5.

Algorithm 5 Test–Code Boundary Repair.

Require: Test–Code pair $p = (T, C)$, Imports I, Binding b
Ensure: Repaired pair $p' = (T', C')$

$T_0 \leftarrow T$, $C_0 \leftarrow C$
while true **do**
  $\mathcal{N} \leftarrow \textsc{RuffF821}(I, C, b)$
  if $\mathcal{N} = \emptyset$ **then break**
  $n \leftarrow \textsc{FirstUndefined}(\mathcal{N})$
  $u \leftarrow \textsc{TopDefinition}(T, n)$
  if $u = \emptyset$ **then return** FAILURE
  $(T, C) \leftarrow \textsc{MoveToCode}(u, T, C)$
end while
$\mathcal{U} \leftarrow \textsc{TopUnits}(C, T)$
$\mathcal{A} \leftarrow \textsc{TestAnchors}(\mathcal{U})$
$\mathcal{H} \leftarrow \textsc{TestSupport}(\mathcal{U})$
$r \leftarrow \textsc{InitRoles}(\mathcal{U}, \mathcal{A}, \mathcal{H})$
F $\leftarrow$ true
while F **do**
  F $\leftarrow$ false
  $\mathcal{D}_T \leftarrow \textsc{Defs}(\{v \in \mathcal{U} \mid r(v) = \text{TEST}\})$
  $\mathcal{D}_C \leftarrow \textsc{Defs}(\{v \in \mathcal{U} \mid r(v) = \text{CODE}\})$
  for each $u \in \mathcal{U}$ with $r(u) \neq \text{TEST}$ **do**
    if $\textsc{Refs}(u) \cap (\mathcal{D}_T \setminus \mathcal{D}_C) \neq \emptyset$ **then**
      $r(u) \leftarrow \text{TEST}$
      F $\leftarrow$ true
    end if
  end for
end while
$r \leftarrow \textsc{FillUnknownRoles}(\mathcal{U}, r)$
$(T', C') \leftarrow \textsc{ComposeByRoles}(\mathcal{U}, r)$
if $\textsc{BindingConflict}(T_0, C_0, T', C', b)$ **then return** FAILURE
$e \leftarrow \textsc{Execute}(T', C')$
if not $\textsc{Passed}(e)$ **then return** FAILURE
return (T', C')

Import Splitting. Before splitting the imports, we first expand star imports into explicit imports. When symbol resolution depends on repository-local context, we resolve the exported symbols in the reconstructed localized execution environment. The joint imports are then independently combined with the code and the test, and Ruff F401 is used to remove imports unused on each side, yielding code-side and test-side imports.

Static Closure and Detached Execution. Static closure checking uses Ruff F821 to obtain the undefined-name sets of the code, the test, and their combination, denoted by U_C, U_T, and U_{CT}, respectively. We require the code to contain no unresolved names, the test to contain no unknown dependency that remains unresolved after combination, and at least one missing test-side name to be supplied by the code:

$$U_C = \emptyset, \qquad U_T \cap U_{CT} = \emptyset, \qquad U_T \setminus U_{CT} \neq \emptyset.$$

The static dependency is further verified through detached execution. The test must fail when the code is removed, while the code must execute successfully when the test is removed.

Case Merging. For validated test–code pairs from the same repository revision with identical normalized code, code-side imports, and binding, we attempt to consolidate their distributed real tests. Before merging, we check compatibility among test execution protocols, top-level test definitions, and recovered resources, and skip candidate groups containing conflicts. For compatible groups, we combine the tests, test-side imports, and runtime resources and reconstruct an execution entry covering all test entries. The merged case is re-executed against the real tests and retained only when execution succeeds; otherwise, the original independent test–code pairs are restored.

Signature Skeleton Extraction and Validation. We first use Ruff F821 to identify undefined names on the test side and match them against top-level code definitions to determine the entries exposed by the tests. Starting from these entries, we construct an implementation-stripped signature skeleton and recursively add required top-level definitions until the skeleton is statically closed. Finally, we temporarily replace the original code with the signature skeleton and re-execute the real tests. If the tests still pass, the case does not require the actual implementation and is filtered; otherwise, the original code and the extracted signature are retained. The complete procedure is given in Algorithm 6.

Algorithm 6 Signature Skeleton Extraction and Validation.

Require: Test–Code pair $p = (T, C)$, Code-side imports I_C, Test-side imports I_T, Binding b
Ensure: Signature skeleton S

$\mathcal{D} \leftarrow \textsc{TopDefs}(C)$
$\mathcal{N}_T \leftarrow \textsc{RuffF821}(I_T, \textsc{Bind}(T, b))$
$K \leftarrow \mathcal{N}_T \cap \textsc{Names}(\mathcal{D})$
if $K = \emptyset$ **then return** FAILURE
$A \leftarrow K$
while true **do**
  $S \leftarrow \textsc{Skeleton}(A, \mathcal{D})$
  $\mathcal{N}_S \leftarrow \textsc{RuffF821}(I_C, S)$
  $\mathcal{H} \leftarrow (\mathcal{N}_S \cap \textsc{Names}(\mathcal{D})) \setminus A$
  if $\mathcal{H} = \emptyset$ **then break**
  $A \leftarrow A \cup \mathcal{H}$
end while
if $\mathcal{N}_S \neq \emptyset$ **then return** FAILURE
$e \leftarrow \textsc{Execute}(T, S)$
if $\textsc{Passed}(e)$ **then return** FAILURE
return S

B.6 Test–Code Pair Cleaning

Decontamination. Before instruction synthesis, we decontaminate executable test–code pairs against evaluation benchmarks. Following the decontamination practices of Magicoder and Self-CodeAlign (Wei et al., 2024b;a), we collect reference strings from benchmark-specific fields listed in Table 7. We remove a test–code pair if any complete reference string occurs verbatim in its code, test, or associated imports.

Code Deduplication. After decontamination, we concatenate the normalized code-side imports and normalized code as the representation for near-duplicate detection. We convert this representation into character-level n-gram sets, use MinHash-LSH for candidate retrieval, and make the final duplicate decision using exact Jaccard similarity. Among retrieved candidates, we take the highest Jaccard similarity and remove the later sample if it reaches the deduplication threshold. We use character-level 5-grams and 128 MinHash permutations, with a threshold of 0.95 for both LSH candidate retrieval and exact Jaccard verification.

Table 7: Benchmark versions and reference fields used for test–code pair decontamination.

Benchmark	Version or Range	Reference Fields
HumanEval	test split	docstring from prompt, canonical_solution, test
MBPP	task IDs 11–510	text, code, test_setup_code, test_list, challenge_test_list
APPS	test split	question, solutions
DS-1000	Insertion problems	prompt, reference_code
BigCodeBench	v0.1.4	complete_prompt, instruct_prompt, canonical_solution, test
LiveCodeBench	release_v6	question_content

B.7 Cross-Model Diagnosis and Supervision Selection

For $i, j \in \{t, s\}$, let M_t and M_s denote the target and stronger models, respectively. Given an executable test–code pair (T, C), model M_i first synthesizes an instruction I_i from T and C. Model M_j then generates code $\hat{C}_{ij}$ from I_i and the interface signature skeleton S. Here, $\hat{C}_{ij}$ denotes the final code after initial generation and, if needed, at most one execution-guided repair attempt. The two generation stages are formalized as

$$(T, C) \xrightarrow{M_i} I_i, \qquad (I_i, S) \xrightarrow{M_j} \hat{C}_{ij}. \tag{1}$$

Finally, we execute $\hat{C}_{ij}$ against the paired tests T and define the verification outcome as

$$V_{ij} = \mathbf{1}\Big[\mathrm{Exec}(\hat{C}_{ij}, T) = \mathrm{pass}\Big]. \tag{2}$$

Together, instruction synthesis, code generation, and execution verification yield the verified instruction–code triple $(I_i, \hat{C}_{ij}, V_{ij})$.

Capability diagnosis is restricted to samples for which target-model verification fails while stronger-model verification succeeds, i.e., $V_{tt} = 0$ and $V_{ss} = 1$. We define the paired verified triples from the two same-model paths as the capability-gap data

$$\mathcal{D}^{(t,s)}_{\mathrm{gap}} = \Big\{ \big((I_t, \hat{C}_{tt}, V_{tt}),\ (I_s, \hat{C}_{ss}, V_{ss})\big) \ \Big|\ V_{tt} = 0 \ \wedge\ V_{ss} = 1 \Big\}. \tag{3}$$

Our heuristic capability diagnosis uses the two cross-verification outcomes V_{ts} and V_{st} to classify samples under the fixed generation-and-repair procedure (Table 8). For $(V_{ts}, V_{st}) = (1, 0)$, the stronger model succeeds with the target instruction and we label the case Code; for $(0, 1)$, the target model succeeds with the stronger-model instruction and we label it Instr. Two failed cross outcomes are labeled Joint; two passing outcomes are labeled Ambiguous, with multiple verified supervision paths. These labels summarize observed outcomes, which can also reflect generation variability and instruction–model interactions, and therefore do not establish causal isolation of capability bottlenecks. The supervision pairs retained for the first three categories constitute $\mathcal{D}^{(t)}_{\mathrm{code}}$, $\mathcal{D}^{(t)}_{\mathrm{instr}}$, and $\mathcal{D}^{(t)}_{\mathrm{joint}}$, respectively.

Table 8: Heuristic capability diagnosis and supervision forms for capability-gap samples. A value of 1 denotes that the generated code passes the paired tests T, and 0 denotes failure.

V_{ts}	V_{st}	Diagnostic category	Supervision	Dataset
1	0	Code generation	$(I_t, \hat{C}_{ts})$	$\mathcal{D}^{(t)}_{\mathrm{code}}$
0	1	Instruction synthesis	$(I_s, \hat{C}_{st})$	$\mathcal{D}^{(t)}_{\mathrm{instr}}$
0	0	Joint	$(I_s, \hat{C}_{ss})$	$\mathcal{D}^{(t)}_{\mathrm{joint}}$
1	1	Ambiguous	TS / ST / SS	$\mathcal{D}^{(t)}_{\mathrm{ambig\text{-}ts/st/ss}}$

Within $\mathcal{D}^{(t,s)}_{\mathrm{gap}}$, ambiguous samples satisfy $V_{ts} = V_{st} = 1$, and the verified TS, ST, and SS paths provide three candidate supervision forms. We define them as $\mathcal{D}^{(t)}_{\mathrm{ambig\text{-}ts}}$, $\mathcal{D}^{(t)}_{\mathrm{ambig\text{-}st}}$, and $\mathcal{D}^{(t)}_{\mathrm{ambig\text{-}ss}}$,

respectively:

$$\begin{aligned}\mathcal{D}^{(t)}_{\text{ambig-ts}} &= \left\{(I_t, \hat{C}_{ts}) \mid V_{ts} = 1 \wedge V_{st} = 1\right\},\\ \mathcal{D}^{(t)}_{\text{ambig-st}} &= \left\{(I_s, \hat{C}_{st}) \mid V_{ts} = 1 \wedge V_{st} = 1\right\},\\ \mathcal{D}^{(t)}_{\text{ambig-ss}} &= \left\{(I_s, \hat{C}_{ss}) \mid V_{ts} = 1 \wedge V_{st} = 1\right\}.\end{aligned} \tag{4}$$

The relative sizes of the first three datasets form a diagnostic profile, and the largest category heuristically determines the supervision direction

$$d^* = \arg \max_{d \in \{\text{code,instr,joint}\}} \left|\mathcal{D}^{(t)}_d\right|. \tag{5}$$

Along the selected direction, we combine the corresponding category with aligned Ambiguous supervision to form the expansion supervision $\mathcal{D}^{(t)}_{\text{expand}}$:

$$\mathcal{D}^{(t)}_{\text{expand}} = \begin{cases}\mathcal{D}^{(t)}_{\text{code}} + \mathcal{D}^{(t)}_{\text{ambig-ts}}, & d^* = \text{code},\\ \mathcal{D}^{(t)}_{\text{instr}} + \mathcal{D}^{(t)}_{\text{ambig-st}}, & d^* = \text{instr},\\ \mathcal{D}^{(t)}_{\text{joint}} + \mathcal{D}^{(t)}_{\text{ambig-ss}}, & d^* = \text{joint}.\end{cases} \tag{6}$$

Finally, the self-generated supervision and expansion supervision form the final target-model instruction–code training dataset

$$\mathcal{D}^{(t)}_{\text{inst}} = \mathcal{D}^{(t)}_{\text{self}} + \mathcal{D}^{(t)}_{\text{expand}}. \tag{7}$$

B.8 Instruction–Code Pair Cleaning

Instruction Checking. We remove instructions that are too short, truncated, or excessively repetitive, as well as outputs that leak prompt structure, original code, or test content. For code and test leakage, we detect long contiguous overlaps using code-lexical tokens, including identifiers, literals, operators, and punctuation, avoiding false positives from isolated API names or common programming terms. The complete rejection criteria are summarized in Table 9.

Table 9: Instruction rejection criteria used during instruction–code pair cleaning.

Criterion	Rejection Condition
Insufficient content	Fewer than 40 characters or fewer than 8 words.
Generation truncation	Instruction synthesis reaches the model output-length limit.
Excessive repetition	Repeated sentences or non-overlapping 16- or 32-token blocks contribute at least 24 repeated tokens and account for at least 20% of the instruction; only sentences containing at least 8 tokens are considered.
Prompt leakage	A complete line reproduces an input or output field marker from the instruction-synthesis prompt.
Code leakage	The instruction and its original code share a contiguous 12-token code-lexical sequence spanning at least 48 characters.
Test leakage	The instruction and the test context used for instruction synthesis satisfy the same 12-token and 48-character overlap criterion.

Decontamination. Unlike the first-stage exact containment check for executable test–code pairs, second-stage decontamination applies word-level n-gram matching to the constructed instructions and code. Instructions are compared against benchmark task descriptions and code against reference implementations; after lowercasing and whitespace tokenization, a pair is removed if either side shares a contiguous 10-word n-gram with the corresponding benchmark reference. The reference fields are listed in Table 10.

Table 10: Benchmark reference content used for second-stage decontamination of generated instruction–code pairs.

Benchmark	Version or Range	Instruction References	Code References
HumanEval	test split	docstring from prompt	canonical_solution
MBPP	task IDs 11–510	text	code
APPS	test split	question	solutions
DS-1000	Insertion problems	prompt	reference_code
BigCodeBench	v0.1.4	complete_prompt, instruct_prompt	canonical_solution
LiveCodeBench	release_v6	question_content	–

C Experimental Details

C.1 Construction Configuration

The automated construction pipeline can be extended through test frameworks, import whitelists, and behavior rules, while its data yield can be adjusted through configurations for candidate selection, repository coverage, and resource recovery, as summarized in Table 11. For repository coverage, we rank repository–revision pairs by the number of cross-file cases they support, select a high-frequency head targeting 80% case coverage with at most 11,000 snapshots, and then add 8,000 snapshots from the remaining low-frequency tail; repository archives are limited to 100 MB.

Table 11: Configurable aspects of the automated data construction pipeline and their effects on data yield.

Configuration Area	Configurable Components	Effect on Data Yield
Candidate Selection	Framework registry (Box B.1); import whitelist (Box B.2); behavior blacklist (Box B.3)	Adjusts the test frameworks, dependencies, and behaviors admitted to downstream construction.
Repository Coverage	Head coverage target; head snapshot cap; tail snapshot count; repository archive-size limit	Adjusts the repository snapshots available for cross-file resolution and resource recovery.
Resource Recovery	Local dependency-completion bounds; relative-import resource depth, count, and total-size limits; file-resource completion rounds and path-depth limits	Adjusts the dependencies and runtime resources recovered for each candidate.

C.2 Implementation Details

Computing Environment. Data construction uses a single NVIDIA RTX 4090 GPU (24 GB), 16 vCPUs, 120 GB of RAM, and Python 3.10, while training and evaluation use a single NVIDIA A800 GPU (80 GB), 14 vCPUs, 120 GB of RAM, and Python 3.10.

Generation Configuration. We serve target models with vLLM (Kwon et al., 2023) and access both target and stronger models through OpenAI-compatible APIs. Across generation stages, we use one-shot prompting (one sampled demonstration per prompt) with temperature 0.2, top-p 0.95, a maximum generation length of 2,048 tokens, and seed 42.

Training Configuration. All models are full-parameter fine-tuned for 2 epochs with a maximum sequence length of 2,048, an effective batch size of 32, a learning rate of 1×10^{-5}, and zero weight decay. All fine-tuning runs use seed 42. We report mean scores over repeated runs (at least three per fine-tuning setting). Training uses the model's native chat template when available; consequently,

Qwen3-4B-Base uses chat serialization, whereas SmolLM3-3B-Base, which has no native chat template, uses completion serialization.

C.3 Controlled Construction Settings

Each controlled study changes only the corresponding component of Main. Execution Verification changes only the final supervision selection based on execution outcomes, whereas w/o Test Injection removes test T only from instruction synthesis. Gen Test replaces T with a generated test, which is first executed against source code C, may undergo one repair attempt after an initial failure, and is retained for subsequent instruction synthesis and execution verification only if it passes. The three instruction-construction settings are summarized in Table 12. The Test Source comparison strictly aligns the cleaned Main and Gen Test Self data by case identity and same-/cross-file relation.

Table 12: Controlled instruction-construction settings used in the functional-constraint-injection and test-source experiments.

Setting	Test Source	Test in Instruction Synthesis
Main	Real	Yes
w/o Test Injection	Real	No
Gen Test	Generated	Yes

C.4 Baseline Dataset Conversion

To enable controlled fine-tuning comparisons across public code instruction datasets, we convert the training sets of the eight baselines into a common instruction–code format without rewriting their original task content. For field-formatted datasets, we directly read the instruction and code fields listed in Table 13; for message-formatted XCoder and CodeEvo, we use the user message as the instruction and the assistant message as the response. For datasets requiring code extraction from the response, we use the first Markdown code block as the code target; for multilingual datasets, the language of Magicoder and XCoder is parsed from the code fence, while AutoCodeInstruct uses its released language field. For KodCode, we retain only samples with `style=instruct` and use `r1_solution` as the code target. For AutoCodeInstruct, we use all 13,940 examples from `autocodeinstruct_v3answer_qwen7b.jsonl`; for SelfCodeAlign, we randomly sample 50K examples; and for each remaining baseline, we randomly sample 75K examples. For baselines requiring random subsampling, we construct three without-replacement samples using data-sampling seeds 16, 42, and 64, fine-tune each with seed 42 and identical hyperparameters, and report the arithmetic mean of their scores in Table 1.

Table 13: Conversion of public baseline datasets into the unified instruction–code format used for controlled fine-tuning.

Baseline	Dataset	Sample Size	Instruction Field	Code Field
SelfCodeAlign	bigcode/self-oss-instruct- sc2-exec-filter-50k	50K	instruction	response
Magicoder	ise-uiuc/Magicoder- OSS-Instruct-75K	75K	problem	solution
XCoder	banksy235/ XCoder-80K	75K	user message	assistant message
EpiCoder	microsoft/EpiCoder- func-380k	75K	instruction	output
KodCode	KodCode/KodCode- V1-SFT-R1	75K	question	r1_solution
OpenCodeInstruct	nvidia/ OpenCodeInstruct	75K	input	output
AutoCodeInstruct	tencent/ AutoCodeBenchmark	14K	question	output
CodeEvo	QiushiSun/ CodeEvo-100K	75K	user message	assistant message

D Data Statistics

D.1 Executable Test–Code Pair Funnel

Figure 5 summarizes the overall construction process of same-file and cross-file executable test–code pairs. As shown in Figure 5, data construction is not a monotonic filtering process, as Resource

Completion adds initially failed candidates to the executable set, while Case Merging changes the counting unit by consolidating multiple validated samples into a single sample.

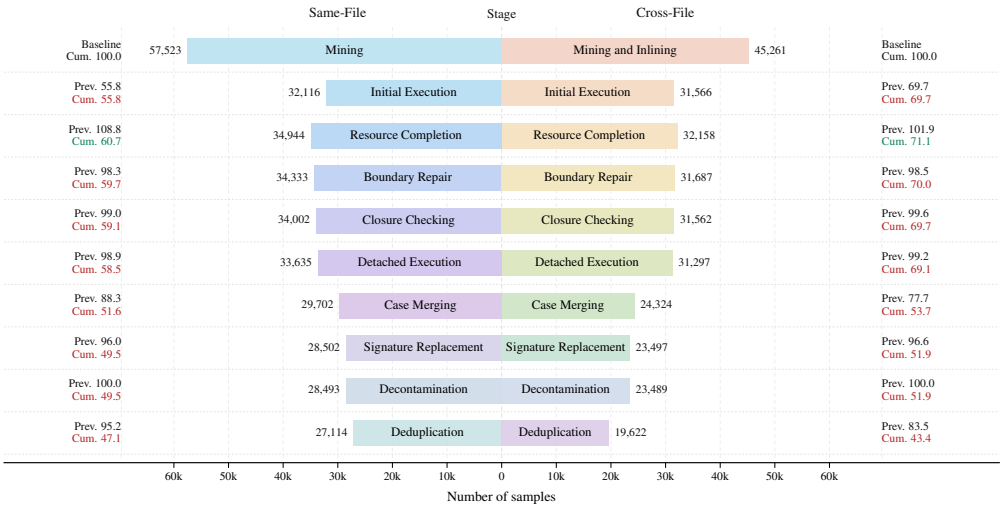


Figure 5: Construction funnel for same-file and cross-file executable test–code pairs.

D.2 Executable Test–Code Pair Recovery

Beyond the aggregate changes summarized by the funnel, we analyze how cross-file inlining and resource completion recover repository context disrupted by local extraction and quantify the executable cases recovered by these stages.

Cross-File Mining and Inlining. Among 322,614 mined cross-file candidates, 308,633 admit identifiable target code, corresponding to a target-identification rate of 95.67%. To balance frequent and long-tail repositories, snapshot selection covers 80.00% of the candidates with head snapshots and additionally selects 8,000 tail snapshots. Head and tail snapshots achieve comparable availability rates of 89.12% and 89.92%, respectively, allowing long-tail sampling to extend repository coverage without reducing snapshot availability. After repository retrieval and Inlining, we obtain 45,261 cross-file test–code pairs, among which 4,381 require at least one re-export-tracing step to locate the target implementation.

Resource Completion. Table 15 summarizes the routing and recovery of initial execution failures through relative-import and file-resource completion. Among 25,407 initial same-file failures, 7,633 are routed to relative-import completion, of which 1,242 are successfully recovered. For cross-file pairs, Inlining disallows relative imports in the inlined code, so the 120 cases entering this stage arise from test-side relative-import dependencies, of which 39 are successfully recovered. File-resource completion further recovers 1,586 of 3,724 same-file candidates and 553 of 1,159 cross-file candidates, corresponding to recovery rates of 42.59% and 47.71%, respectively. Table 16 shows that file-resource recovery converges rapidly, with the first two rounds accounting for 94.83% and 96.56% of all eventually recovered same-file and cross-file cases, respectively. Table 17 shows that file-resource completion is typically localized, with 79.12% of same-file and 88.58% of cross-file cases requiring only one recovered file, while relative-import completion also handles longer multi-file dependency chains. Remaining failures are dominated by missing local files, which account for 76.02% and 99.28% of relative-import errors and 54.63% and 75.99% of file-resource errors in the same-file and cross-file settings, respectively.

D.3 Executable Test–Code Pair Refinement

Refinement further corrects test–code boundaries, validates functional dependencies, consolidates fragmented supervision, and filters cases that do not depend on the actual implementation.

Table 14: Cross-file candidate preparation and inlining outcomes under the default configuration. Head and tail selection rates denote candidate coverage; all other rates are computed relative to the corresponding stage input.

Outcome	Count	Rate
Mining		
Mined cross-file candidates	322,614	100.00%
Target Identification		
Target-identifiable candidates	308,633	95.67%
Repository snapshots represented	39,845	–
Repository Snapshot Selection		
Head repository snapshots selected	10,878	80.00%
Tail repository snapshots selected	8,000	2.59%
Total repository snapshots selected	18,878	47.38%
Available head repository snapshots	9,694	89.12%
Available tail repository snapshots	7,194	89.92%
Available repository snapshots	16,888	89.46%
Retrieved candidates	199,541	64.65%
Code Inlining		
Successfully inlined	45,261	22.68%
Disallowed dependencies	73,984	37.08%
Disallowed behaviors	1,092	0.55%
Residual relative imports	43,977	22.04%
Target localization failure	34,698	17.39%
Name conflicts	529	0.27%

Table 15: Execution-failure routing and recovery across relative-import and file-resource completion.

Initial Execution Failures			**Relative-Import Completion**			**File-Resource Completion**		
Failure reason	*Same*	*Cross*	*Outcome*	*Same*	*Cross*	*Outcome*	*Same*	*Cross*
Missing dependency	9,546	2,449	Source pool	25,407	13,695	Source pool	24,165	13,656
Rate	37.57%	17.88%	Missing dependency	7,633	120	Missing-file cases	3,724	1,159
Missing file	3,628	1,155	Candidate rate	30.04%	0.88%	Candidate rate	15.41%	8.49%
Rate	14.28%	8.43%	Recovered	1,242	39	Recovered	1,586	553
Timeout	532	76	Recovery rate	16.27%	32.50%	Recovery rate	42.59%	47.71%
Rate	2.09%	0.55%	Still failed	6,391	81	Still failed	2,138	606
Other error	11,701	10,015						
Rate	46.05%	73.13%						
Total failed	25,407	13,695						

Table 16: Round-wise file-resource recovery for same-file and cross-file cases.

Round	Same-file Recovered	Same-file Cumulative	Cross-file Recovered	Cross-file Cumulative
1	1,256	1,256	482	482
2	248	1,504	52	534
3	39	1,543	9	543
4	14	1,557	3	546
5	6	1,563	3	549
6	13	1,576	2	551
7	3	1,579	1	552
8	1	1,580	1	553
9	2	1,582	0	553
10	2	1,584	0	553
11	0	1,584	0	553
12	2	1,586	0	553

Table 17: Resource-completion diagnostics for relative-import and file resources. The left panel reports the number of successfully recovered resources per case, and the right panel reports the reasons preventing further recovery.

(a) Recovered-resource count distribution

Resource Type	Count	Same-file	Cross-file
Relative import	1	1,943 (29.32%)	53 (46.09%)
	2	1,028 (15.51%)	3 (2.61%)
	3–5	1,040 (15.69%)	6 (5.22%)
	6–10	596 (8.99%)	0 (0.00%)
	11+	2,021 (30.49%)	53 (46.09%)
Total	All	6,628	115
File resource	1	1,660 (79.12%)	698 (88.58%)
	2	316 (15.06%)	60 (7.61%)
	3–5	83 (3.96%)	20 (2.54%)
	6–10	28 (1.33%)	8 (1.02%)
	11+	11 (0.52%)	2 (0.25%)
Total	All	2,098	788

(b) Resource-completion failure reasons

Resource Type	Failure Reason	Same-file	Cross-file
Relative import	repo snapshot missing	2,999 (17.28%)	0 (0.00%)
	all candidates failed	1,025 (5.91%)	0 (0.00%)
	local file missing	13,194 (76.02%)	138 (99.28%)
	max files exceeded	139 (0.80%)	1 (0.72%)
Total	All	17,357	139
File resource	missing path not found	63 (3.62%)	72 (19.00%)
	unsafe relative path	1 (0.06%)	0 (0.00%)
	max up-levels exceeded	160 (9.20%)	17 (4.49%)
	repo snapshot missing	556 (31.97%)	0 (0.00%)
	no repository candidates	9 (0.52%)	2 (0.53%)
	local file missing	950 (54.63%)	288 (75.99%)
Total	All	1,739	379

Boundary Repair. Table 18 summarizes the Boundary Repair outcomes and repair directions. Most pairs require no modification, and Boundary Repair ultimately retains 34,333 same-file and 31,687 cross-file pairs, corresponding to 98.25% and 98.54% of the inputs. Among successfully repaired pairs, 203 of 205 same-file cases and all 76 cross-file cases require only unidirectional boundary adjustment.

Table 18: Boundary Repair outcomes and repair directions for same-file and cross-file executable test–code pairs.

(a) Repair Outcomes

Outcome	Same-file	Cross-file
Input	34,944	32,158
Without repair	34,128	31,611
Code dependency absent from test	226	213
Bind-movement conflict	0	60
Semantically empty code	380	198
Failed post-repair execution	5	0
Successfully repaired	205	76
Output	34,333	31,687

(b) Repair Directions

Direction	Same-file	Cross-file
Test → Code only	66	0
Code → Test only	137	76
Bidirectional	2	0
Successfully repaired	205	76

Closure Checking and Detached Execution. Table 19 summarizes the Closure Checking and Detached Execution outcomes. Closure Checking retains 99.04% and 99.61% of same-file and cross-file pairs, respectively, while Detached Execution subsequently retains 98.92% and 99.16%. Together, the two stages retain 33,635 same-file and 31,297 cross-file pairs, corresponding to 97.97% and 98.77% of their inputs.

Case Merging. Table 20 summarizes the scale and execution-validation outcomes of Case Merging. Compatible merge groups achieve re-execution success rates of 99.34% and 99.67% in the same-file and cross-file settings, respectively. Case Merging ultimately reduces the corpus by 3,933 and 6,973 records through consolidation of validated test supervision. The effect of Case Merging on the strength of aggregated real-test supervision is further evaluated in Appendix E.6.

Signature Extraction and Replacement. Signature Replacement identifies 1,187 same-file and 728 cross-file pairs whose real tests still pass against the signature skeleton, filtering supervision that does not depend on the actual implementation. After this validation, 28,502 same-file and 23,497 cross-file test–code pairs remain, corresponding to retention rates of 95.96% and 96.60%, respectively.

Table 19: Static closure-checking and dynamic detached-execution outcomes for same-file and cross-file test–code pairs.

(a) Closure Checking

Outcome	Same-file	Cross-file
Input	34,333	31,687
Code not closed	3	0
Test contains unknown dependencies	48	20
No static test → code dependency	280	105
Removed	331	125
Passed	34,002	31,562

(b) Detached Execution

Outcome	Same-file	Cross-file
Input	34,002	31,562
Test passes without code only	367	259
Code fails without test only	0	6
Both	0	0
Removed	367	265
Passed	33,635	31,297

Table 20: Case Merging outcomes and successful merge-group sizes for same-file and cross-file test–code pairs.

(a) Merging Outcomes

Outcome	Same-file	Cross-file
Input pairs	33,635	31,297
Multi-pair groups	1,566	3,512
Pairs in multi-pair groups	6,059	13,335
Compatible merge candidates	1,372	2,722
Conflict groups retained unchanged	194	790
Execution-validated groups	1,363	2,713
Failed merge groups	9	9
Rollback children	40	38
Output pairs	29,702	24,324
Structural count reduction	3,933	6,973

(b) Successful Merge-Group Size

Child pairs per group	Same-file	Cross-file
2	715	1,399
3	281	538
4	142	288
5–8	182	357
> 8	43	131
Total groups	1,363	2,713
Mean	3.89	3.57

D.4 Instruction–Code Pairs

Instruction Synthesis and Verification. Using Qwen3-4B-Base and SmolLM3-3B-Base as the target models and GPT-5.6-Luna as the stronger model, we report the pass and failure counts obtained from instruction synthesis and code verification over the final executable test–code pairs. Among target-model verification failures, the stronger model further verifies 21,343 of 23,219 Qwen3-4B-Base cases and 31,990 of 33,934 SmolLM3-3B-Base cases, corresponding to 91.92% and 94.27%, respectively, and yielding the samples used for subsequent capability diagnosis.

Table 21: Instruction synthesis and verification results on the final executable test–code pairs.

Model	API Mode	Same-file		Cross-file		Overall	
		Pass	Fail	Pass	Fail	Pass	Fail
Qwen3-4B-Base	Completion	15,570	11,544	7,947	11,675	23,517	23,219
SmolLM3-3B-Base	Completion	8,849	18,265	3,953	15,669	12,802	33,934
GPT-5.6-Luna	Chat	26,215	899	18,518	1,104	44,733	2,003

Diagnosis and Cleaning. Table 22 summarizes the Self, Code, Instr, Joint, and Ambiguous data sizes before and after cleaning for both target models. Raw denotes the data before the two cleaning stages, Cleaned denotes the retained data afterward, and Overall combines the same-file and cross-file data. Ambig-TS, Ambig-ST, and Ambig-SS are three alternative supervision forms for the same Ambiguous pool and are therefore not additive. Across both target models and all data sources, cleaning retains 80.10%–93.69% of the pairs.

Table 22: Target-specific instruction–code pair composition before and after cleaning.

(a) Qwen3-4B-Base

Data Source	Same-file		Cross-file		Overall	
	Raw	Cleaned	Raw	Cleaned	Raw	Cleaned
Self	15,570	12,741	7,947	6,867	23,517	19,608
Code	3,825	3,228	3,766	3,134	7,591	6,362
Instr	1,067	937	1,007	890	2,074	1,827
Joint	2,120	1,880	2,599	2,180	4,719	4,060
Ambig-TS	3,711	3,225	3,248	2,806	6,959	6,031
Ambig-ST	3,711	3,226	3,248	2,965	6,959	6,191
Ambig-SS	3,711	3,422	3,248	3,007	6,959	6,429

(b) SmolLM3-3B-Base

Data Source	Same-file		Cross-file		Overall	
	Raw	Cleaned	Raw	Cleaned	Raw	Cleaned
Self	8,849	7,056	3,953	3,395	12,802	10,451
Code	8,227	6,598	5,800	4,637	14,027	11,235
Instr	1,079	955	972	883	2,051	1,838
Joint	4,308	3,901	5,183	4,486	9,491	8,387
Ambig-TS	3,795	3,140	2,626	2,197	6,421	5,337
Ambig-ST	3,795	3,314	2,626	2,439	6,421	5,753
Ambig-SS	3,795	3,546	2,626	2,470	6,421	6,016

Controlled Variants. For Qwen3-4B-Base, we report the dataset sizes used by the execution-verification, test-injection, and test-source controls. Random Subset is strictly size-matched to Self, while the Main and Gen Test intersection datasets are both matched at 6,466 pairs.

Table 23: Dataset sizes for execution-verification variants and test-injection/source settings on Qwen3-4B-Base.

(a) Execution-Verification Variants

Training Data	Raw	Cleaned
Self	23,517	19,608
Failures Only	23,219	18,850
Self + Failures	46,736	38,458
Random Subset	–	19,608

(b) Test Injection and Source Variants

Setting	Training Data	Cleaned
Main	Self	19,608
w/o Test Injection	Self	20,411
Gen Test	Self	9,231
Main	Intersection	6,466
Gen Test	Intersection	6,466

E Additional Results

E.1 Trade-offs in Execution Timeout

To characterize the yield–cost trade-off of execution timeout, we first perform a broad sweep over 1, 5, 10, 30, 60, and 180 seconds on same-file cases. As shown in Fig. 6(a–b) and Table 24(a), execution yield increases rapidly at short timeouts and clearly begins to saturate beyond 10 seconds. Based on this observation, we use a denser sweep over 1, 2, 3, 5, 7, and 10 seconds for cross-file cases, and add 15 seconds as a short extension between 10 seconds and the next point in the original coarse sweep at 30 seconds, yielding 1, 2, 3, 5, 7, 10, and 15 seconds in total, to test whether cross-file execution exhibits the same saturation trend (Fig. 6(c–d) and Table 24(b)). Each setting is repeated three times under the same execution configuration, and we report median [min, max].

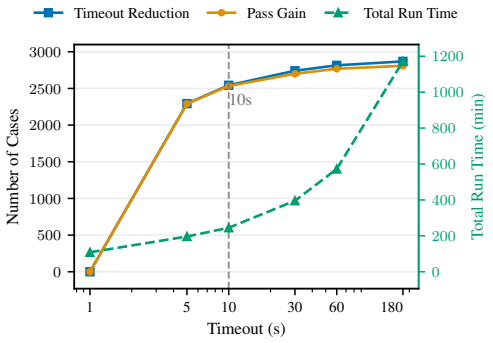
Let t_i denote the i-th timeout setting, and let P_i, Q_i, and C_i denote the median numbers of passed cases, timeout cases, and total run time in minutes, respectively, with $t_0 = 1$ second. Panels (a) and (c) report cumulative gains relative to the shortest setting,

$$\text{PassGain}_i = P_i - P_0, \qquad \text{TimeoutReduction}_i = Q_0 - Q_i. \tag{8}$$

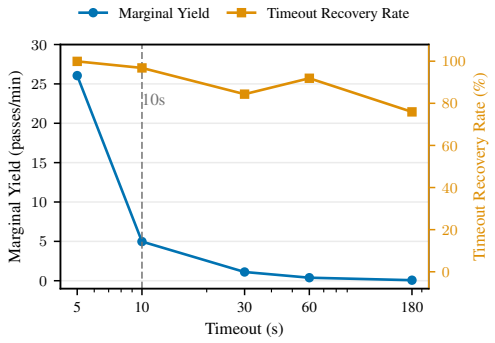
For adjacent settings $t_{i-1} < t_i$, Panels (b) and (d) report

$$\text{MarginalYield}_i = \frac{P_i - P_{i-1}}{C_i - C_{i-1}}, \qquad \text{RecoveryRate}_i = \frac{P_i - P_{i-1}}{Q_{i-1} - Q_i} \times 100\%. \tag{9}$$

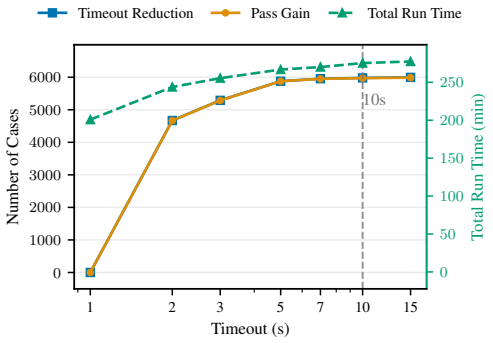
Marginal Yield measures additional passed cases per additional execution minute, while Timeout Recovery Rate measures the fraction of reduced timeout cases that are converted into additional passes.



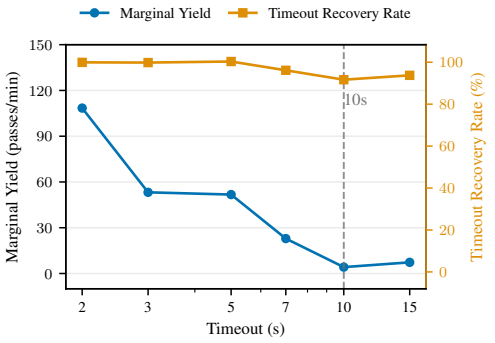
(a) Same-file: cumulative trade-off



(b) Same-file: marginal utility



(c) Cross-file: cumulative trade-off



(d) Cross-file: marginal utility

Figure 6: Execution yield–cost trade-offs under different timeout settings. Panels (a–b) report same-file cases, and Panels (c–d) report cross-file cases. Panels (a) and (c) show Pass Gain and Timeout Reduction relative to the 1-second setting, together with the absolute Total Run Time at each setting. Panels (b) and (d) show Marginal Yield and Timeout Recovery Rate between adjacent timeout settings, where Marginal Yield is the additional passed cases per additional minute of Total Run Time, and Timeout Recovery Rate is the fraction of reduced timeout cases converted into passed cases. The gray dashed line marks the selected 10-second timeout.

On same-file cases, most recoverable executions are obtained at short timeouts. Increasing the timeout from 5 to 10 seconds yields 242 additional passed cases for 48.65 additional execution minutes, whereas extending from 10 to 30 seconds yields only 167 additional passes despite 150.44 extra minutes. Marginal Yield consequently drops from 4.97 passes/min at 10 seconds to 1.11 at 30 seconds, and falls below 0.4 thereafter. These results identify 10 seconds as the onset of a clear diminishing-return regime.

The denser cross-file sweep exhibits the same saturation trend. The number of Passed Cases increases rapidly from 26,166 at 1 second to 32,041 at 5 seconds, but only gains another 97 cases

Table 24: Execution statistics under different timeout settings. Panel (a) reports same-file cases, and Panel (b) reports cross-file cases. Passed Cases, Remaining Timeouts, and Total Run Time are reported as median [min, max] over three repeated runs. Stable Rate denotes the proportion of cases with identical execution outcomes across all three runs.

(a) Same-file

Timeout (s)	Passed Cases	Remaining Timeouts	Total Run Time (min)	Stable Rate (%)
1	32,402 [32,401, 32,498]	3,164 [3,068, 3,165]	108.41 [107.04, 108.50]	98.78
5	34,693 [34,676, 34,701]	871 [856, 882]	196.35 [193.22, 201.14]	99.72
10	**34,935 [34,926, 34,946]**	**621 [607, 625]**	**244.99 [238.05, 246.02]**	**99.80**
30	35,102 [35,100, 35,102]	423 [421, 425]	395.44 [395.13, 396.94]	99.85
60	35,170 [35,169, 35,171]	349 [344, 349]	572.19 [566.55, 575.12]	99.89
180	35,211 [35,211, 35,216]	295 [289, 297]	1,171.40 [1,159.08, 1,171.46]	99.88

(b) Cross-file

Timeout (s)	Passed Cases	Remaining Timeouts	Total Run Time (min)	Stable Rate (%)
1	26,166 [26,117, 26,318]	6,061 [5,909, 6,113]	200.74 [198.91, 201.10]	94.57
2	30,833 [30,794, 30,901]	1,391 [1,322, 1,428]	243.78 [237.93, 244.45]	98.93
3	31,452 [31,423, 31,562]	771 [659, 799]	255.42 [242.53, 260.01]	99.20
5	32,041 [32,039, 32,058]	184 [162, 185]	266.80 [253.39, 269.56]	99.77
7	32,116 [32,099, 32,121]	106 [104, 107]	270.08 [268.85, 273.48]	99.88
10	**32,138 [32,094, 32,140]**	**82 [81, 85]**	**275.34 [272.22, 277.19]**	**99.81**
15	32,153 [32,148, 32,156]	66 [64, 69]	277.39 [277.16, 278.52]	99.93

from 5 to 10 seconds and 15 cases from 10 to 15 seconds. Meanwhile, the Timeout Recovery Rate remains consistently high across timeout intervals, indicating that most recovered timeout cases are converted into passed cases. The Marginal Yield at both 10 and 15 seconds is already low; therefore, increasing the timeout beyond 10 seconds provides only marginal additional coverage, consistent with the saturation trend observed for same-file cases.

E.2 Benefit of Cross-File Recovery

To assess whether cross-file recovery adds value beyond same-file supervision, we fine-tune Qwen3-4B-Base using Self pairs from same-file cases, cross-file cases, or both.

Table 25: Benefit of cross-file recovery using Self supervision on Qwen3-4B-Base. Avg. is the unweighted mean of the nine benchmark scores.

File Relation	Size	HE	HE+	MBPP	MBPP+	BCB-C F	BCB-C H	BCB-I F	BCB-I H	LCB	Avg.
Same-file	13K	84.8	77.4	81.5	69.0	45.5	16.9	36.6	14.2	26.6	50.3
Cross-file	7K	80.5	75.0	80.4	68.3	46.9	16.9	38.7	18.2	27.2	50.2
Overall	20K	85.4	78.7	81.7	69.8	45.4	18.2	37.8	16.9	26.7	51.2

As shown in Table 25, same-file Self training achieves an average score of 50.3, which rises to 51.2 when cross-file pairs are added. Same-file data are stronger on HumanEval and MBPP, while cross-file data reach a comparable average of 50.2 with only about half the number of cases, and are stronger on BCB and LCB. The union surpasses either source on HumanEval and MBPP, while also improving average BCB performance and LCB relative to same-file training alone. These gains show that recovered cross-file pairs provide complementary supervision and improve overall performance.

E.3 Cross-Model Corroboration of Capability Diagnosis

As shown in Table 26, applying the same Code-directed recipe as on Qwen3 improves all seven evaluated metrics over Self on SmolLM3-3B-Base, increasing the Average from 34.3 to 36.9.

Table 26: Cross-model corroboration of the diagnosis-derived selection rule on SmolLM3-3B-Base. Size denotes the cleaned training-set size, and Average is the unweighted mean of the seven evaluation scores.

Training Data	Size	HE	HE+	MBPP	MBPP+	BCB-C F	BCB-C H	LCB	Avg.
+ Self	10K	41.5	33.5	63.2	53.7	29.7	9.5	9.1	34.3
+ Self + Code + Ambig-TS	27K	44.5	39.6	64.6	56.6	30.3	10.8	11.8	36.9

E.4 Controlling for Code Source and Style

To rule out code source and style as alternative explanations for the performance gains, we construct matched controls on the Qwen3-4B-Base Self data with Original, Formatted, Rewritten, and Generated code targets. Original uses the original repository code, and Formatted applies Black to it. Rewritten uses code rewritten by Qwen3-4B-Base while preserving functionality and the external interface (Prompt F.7). Generated uses code produced from the synthesized instructions in Self and verified by the real tests. Formatting and rewriting retain 46,708 and 38,993 cases after verification against the original real tests, respectively. Intersecting these cases with Self by case ID and file relation yields 17,409 training examples shared by all four code targets.

Table 27: Performance under different code targets on the matched Qwen3-4B-Base Self subset. Avg. is the unweighted mean of the nine benchmark scores.

Code Target	Size	HE	HE+	MBPP	MBPP+	BCB-C F	BCB-C H	BCB-I F	BCB-I H	LCB	Avg.
Original	17K	68.9	62.2	74.6	64.8	36.7	10.8	32.2	11.5	19.8	42.4
Formatted	17K	66.5	59.8	74.6	64.0	35.8	8.1	32.2	10.8	20.1	41.3
Rewritten	17K	75.6	68.3	73.0	64.6	41.1	14.9	34.8	10.8	23.3	45.1
Generated	17K	82.3	75.6	80.7	68.8	45.2	16.2	36.3	14.2	26.4	49.5

The untuned Qwen3-4B-Base averages 45.7 (Table 1), whereas Original and Formatted reach only 42.4 and 41.3 on the matched data (Table 27), respectively. This negative transfer may stem from differences in implementation across repositories and a lack of a natural, fluent mapping from synthesized instructions to the corresponding code; surface formatting cannot resolve these issues. Rewritten recovers to 45.1, comparable to the untuned model's 45.7, consistent with model rewriting mitigating this mismatch. Generated averages 49.5, 4.4 points above the strongest control, Rewritten, and outperforms all three controls on every benchmark, suggesting that the gains of Mold-Instruct extend well beyond what code provenance or stylistic alignment can explain.

E.5 Differences Between Test Sources

Dynamic Mutation Analysis. Consistent with the test-source control in the main text, we use 6,466 Intersection cases strictly matched by case ID and file relation, comprising 4,264 same-file and 2,202 cross-file cases. For each case, we use `mutmut` to mutate the original code, fix the sampling seed by case ID and file relation, sample up to 20 mutants, and execute the same mutant code with Real Test and Gen Test. We retain only mutants for which neither test source times out, count non-timeout failures as kills, and macro-average the metrics computed within each case over cases with at least one comparable mutant, giving each valid case equal weight. Of the 6,466 input cases, 194 yield no mutants and one retains no comparable mutants after timeout filtering, leaving 6,271 valid cases. Of 71,510 sampled mutants, 827 are excluded due to timeouts, yielding 70,683 comparable mutants.

As shown in Table 28, Real Test achieves higher mutation scores than Gen Test in both same-file (+9.70 points) and cross-file (+6.50 points) cases. Across all valid cases, the Real-only kill rate (14.91%) is more than twice the Gen-only rate (6.28%). However, the nonzero Gen-only rate indicates that the detection capabilities of the two test sources are not strictly nested.

Table 28: Paired mutation-analysis results (%). All scores and kill categories are computed per case and macro-averaged over valid cases. Both, Real-only, Gen-only, and Neither partition the mutants comparable under both test sources.

File Relation	Cases	Real MS	Gen MS	Both	Real-only	Gen-only	Neither
Same-file	4,165	83.73	74.02	68.54	15.18	5.48	10.79
Cross-file	2,106	78.42	71.92	64.07	14.36	7.86	13.72
Overall	6,271	81.95	73.32	67.04	14.91	6.28	11.78

E.6 Effect of Case Merging on Test Strength

To examine whether case merging consolidates multiple real tests into stronger per-case functional constraints, we conduct a paired mutation analysis for each successfully merged group: the same mutants are generated from the shared code and executed against each child test and the merged test. For each group, we randomly sample up to 20 mutants from the shared code. Metrics use only mutants that do not time out in any execution, retaining 3,972 of 4,076 groups and 64,945 of 65,537 sampled mutants for the paired analysis. Table 29 reports the paired mutation-strength analysis.

Table 29: Mutation-strength comparison for case merging. Groups denotes retained comparable groups. Mean Child, Best Child, Union, and Merged are mutation scores in percentage, Merged–Best is the gain over Best Child in percentage points, and Merge–Union Jaccard measures the similarity between the Merged and Union kill sets.

Relation	Groups	Mean Child	Best Child	Union	Merged	Merged–Best (pp)	Merge–Union Jaccard
Same-file	1,309	61.9%	72.8%	78.0%	78.0%	+5.2	99.95%
Cross-file	2,663	48.0%	61.0%	68.8%	68.8%	+7.8	99.92%
Overall	3,972	52.6%	64.9%	71.8%	71.8%	+6.9	99.93%

The merged test matches the Union at an overall mutation score of 71.8%, with a Merge–Union Jaccard of 99.93%, while improving over the Best Child by 6.9 percentage points. Gains of 5.2 and 7.8 points on same-file and cross-file groups, respectively, show that merging nearly completely preserves the combined test strength and is stronger on average than any individual child.

F Prompt Templates

Each `{examples}` placeholder is filled with one sampled demonstration.

F.1 System Prompt

Prompt F.1: System Prompt

```
You are a coding assistant focused on functional reasoning and skilled in synthesis,
generation, and repair.
```

F.2 Instruction Synthesis

Prompt F.2: Instruction Synthesis

```
# Task
Synthesize a self-contained code instruction that captures the intended functionality
from the given Code, Test{extra_inputs}.

## Requirements
- The instruction must capture enough information from Code to regenerate an equivalent
implementation, without directly copying Code.
```

```
- The instruction must capture the expected behavior, constraints, and edge cases
inferred from Test that directly concern the functionality of Code, without leaking the
contents of Test.
{relative_imports_requirements}
{files_requirements}

## Output Format
Output exactly in the following format:

#### Instruction:

{{instruction}}

{examples}

## Current Query
### Input
#### Code:
```python
{code}
```

Test:
```python
{test}
```
{relative_imports_input}
{files_input}

Output
{completion_prefill}
```

F.3 CODE GENERATION

Prompt F.3: Code Generation

```
# Task
Generate self-contained and executable Code from the given Instruction and Signature.

## Requirements
- The Code must fully implement the functionality and details required by Instruction.
- The Code should preserve relevant symbols and interfaces from Signature.

## Output Format
Output exactly in the following format:

#### Code:
```python
{{code}}
```

{examples}

Current Query
Input
Instruction:

{instruction}

Signature:
```python
{signature}
```

Output
{completion_prefill}
```

F.4 CODE REPAIR

Prompt F.4: Code Repair

````
# Task
Repair the given Buggy Code based on the given Instruction and Signature.

## Requirements
- The repaired Code must fix the failure reported in Error.
- The repaired Code must fully implement the functionality and details required by
Instruction.
- The Code should preserve relevant symbols and interfaces from Signature.

## Output Format
Output exactly in the following format:

#### Code:
```python
{{code}}
```

{examples}

## Current Query
### Input
#### Instruction:

{instruction}

#### Buggy Code:
```python
{buggy_code}
```

#### Error:
```text
{error}
```

#### Signature:
```python
{signature}
```

### Output
{completion_prefill}
````

F.5 TEST GENERATION

Prompt F.5: Test Generation

````
# Task
Generate an executable Test for the given Code.

## Requirements
- The Test should verify the intended functionality of Code, such as expected behavior,
constraints, or edge cases.
- The Test must execute automatically when the Python file created by appending the Test
to Code is run.

## Output Format
Output exactly in the following format:

#### Test:
```python
{{test}}
```

{examples}

## Current Query
### Input
#### Code:
````

````
```python
{code}
```

### Output
{completion_prefill}
````

F.6 Test Repair

Prompt F.6: Test Repair

````
# Task
Repair the given Buggy Test for the given Code.

## Requirements
- The repaired Test must fix the failure reported in Error.
- The repaired Test should verify the intended functionality of Code, such as expected
behavior, constraints, or edge cases.
- The repaired Test must execute automatically when the Python file created by appending
the repaired Test to Code is run.

## Output Format
Output exactly in the following format:

#### Test:
```python
{{test}}
```

{examples}

## Current Query
### Input
#### Code:
```python
{code}
```

#### Buggy Test:
```python
{buggy_test}
```

#### Error:
```text
{error}
```

### Output
{completion_prefill}
````

F.7 Code Rewrite

Prompt F.7: Code Rewrite

````
# Task
Rewrite the given Code into a clean, robust, and consistent implementation.

## Requirements
- Freely revise internal implementation details, such as code structure and organization,
control flow and expressions, and comments and docstrings.
- The Rewritten Code must remain self-contained, preserve the original functionality, and
be compatible with the original external interface.

## Output Format
Output exactly in the following format:

#### Rewritten Code:
```python
````
````

```
{{rewritten_code}}
```

{examples}

Current Query
Input
Code:
```python
{code}
```

Output
{completion_prefill}