EXECUTION-SIGNATURE RECYCLING: DEDUPLICATING UNIT-TEST FAILURE FEEDBACK FOR TEST-TIME CODE SCALING

FARS
Analemma
fars@analemma.ai

ABSTRACT

Test-time scaling improves code generation by sampling multiple candidates and using execution feedback to guide selection or refinement. However, when multiple candidates fail for similar reasons, providing redundant feedback may waste the model's context. We propose Execution-Signature Recycling (ESR), a training-free method that clusters candidates by their execution signatures—the set of failing tests and error types—and conditions subsequent generations on a deduplicated failure bank. We evaluate ESR on HumanEval+ with Qwen2.5-Coder-7B-Instruct under a fixed 16-generation budget. While ESR achieves the highest mean Pass@1 (87.60%), it does not significantly outperform the simpler Self-Debug baseline (86.99%), with a 95% confidence interval of $[-0.81, 2.24]$ that includes zero. This negative result suggests that for strong code generation models, per-sample self-debugging may be sufficient, and cross-sample feedback aggregation does not provide reliable additional benefits.

WARNING: This paper was generated by an automated research system. The code is publicly available.[1]

1 INTRODUCTION

Large language models (LLMs) have demonstrated remarkable capabilities in code generation, and recent work shows that allocating additional inference-time compute—known as test-time scaling—can substantially improve performance (Snell et al., 2024; Brown et al., 2024). A common approach is best-of-N sampling: generate multiple candidate programs, execute them against unit tests, and select the best-performing candidate. This simple strategy can significantly boost Pass@1 accuracy compared to single-sample generation.

Execution feedback provides a powerful signal for improving code generation. Self-debugging methods (Chen et al., 2023) use error messages and tracebacks to iteratively refine failing candidates, while approaches like Reflexion (Shinn et al., 2023) and Self-Refine (Madaan et al., 2023) maintain memory of past failures to guide future attempts. However, these methods typically operate on individual candidates, refining each based on its own execution feedback without sharing information across candidates.

We hypothesize that under best-of-N sampling, multiple candidates often fail for similar reasons—the same edge case, the same type error, or the same off-by-one bug. If failure modes cluster strongly, then providing deduplicated feedback about common failures could help the model avoid redundant exploration and focus on novel solutions. To test this hypothesis, we propose **Execution-Signature Recycling (ESR)**, a training-free test-time scaling method that clusters candidates by their execution signatures (the set of failing tests and error types) and conditions subsequent generations on a compact, deduplicated failure bank.

We evaluate ESR on HumanEval+ with Qwen2.5-Coder-7B-Instruct under a fixed 16-generation budget. While ESR achieves the highest mean Pass@1 (87.60%), it does not significantly outperform the simpler Self-Debug baseline (86.99%), with a 95% confidence interval of $[-0.81, 2.24]$

[1]https://gitlab.com/fars-a/unit-test-grounded-rse

that includes zero. This negative result suggests that for strong code generation models, per-sample self-debugging may be sufficient, and cross-sample feedback aggregation does not provide reliable additional benefits.

Our contributions are:

- We propose ESR, a method that aggregates execution feedback across candidates using signature-based clustering and provides deduplicated failure information to guide subsequent generations.
- We conduct a rigorous evaluation with pre-registered decision rules and paired bootstrap confidence intervals across multiple random seeds.
- We report a negative result: ESR does not significantly outperform per-sample self-debugging on HumanEval+, providing evidence that simpler feedback approaches may be sufficient for strong models.

2 Method

We propose **Execution-Signature Recycling (ESR)**, a training-free test-time scaling method for code generation that aggregates and deduplicates execution feedback across multiple candidate programs. ESR operates under a fixed generation budget split into two rounds: an exploration phase that identifies common failure modes, followed by an exploitation phase that conditions new generations on a compact, deduplicated failure bank.

2.1 Problem Formulation

Given a programming task with natural language specification p and a set of unit tests $\{t_j\}_{j=1}^{m}$, the goal is to generate a program y that passes all tests. Under a fixed generation budget of N candidates, we seek to maximize the probability that the selected candidate passes all evaluation tests (Pass@1).

2.2 Execution Signatures

For a candidate program y executed against unit tests $\{t_j\}_{j=1}^{m}$, we define its **execution signature** as the set of failing test outcomes:

$$\sigma(y) = \{(j, \text{errtype}_j) : t_j \text{ fails on } y\} \tag{1}$$

where errtype_j denotes the error type (e.g., `AssertionError`, `TypeError`, `Timeout`). Passing tests are excluded to keep signatures compact. Two candidates share the same execution signature if and only if they fail on exactly the same tests with the same error types.

2.3 Two-Round Procedure

ESR divides the generation budget $N = 16$ into two rounds of 8 candidates each, as illustrated in Figure 1.

Round 1 (Exploration). We sample 8 candidate programs from the language model and execute each against the base unit tests. For each candidate, we record its execution signature $\sigma(y_i)$.

Failure Bank Construction. We cluster the Round 1 candidates by exact signature equality. For each cluster c with signature σ_c and size $|c|$, we create a bank entry containing: (1) the count $|c|$ indicating how many candidates failed this way, (2) the list of failing test IDs, and (3) for up to $K = 2$ failing tests, the input, expected output, and observed output or exception traceback. We select the top-$M = 3$ clusters by frequency and serialize them into a failure bank B with a maximum token budget (900 tokens).

Round 2 (Exploitation). We prompt the model with the original task specification p, the failure bank B, and an instruction to generate a correct solution that avoids the documented failure modes. We then sample 8 new candidates conditioned on this augmented prompt.

Selection. We pool all 16 candidates from both rounds and select the one that maximizes the base-test pass rate, with ties broken by fewer failing tests.

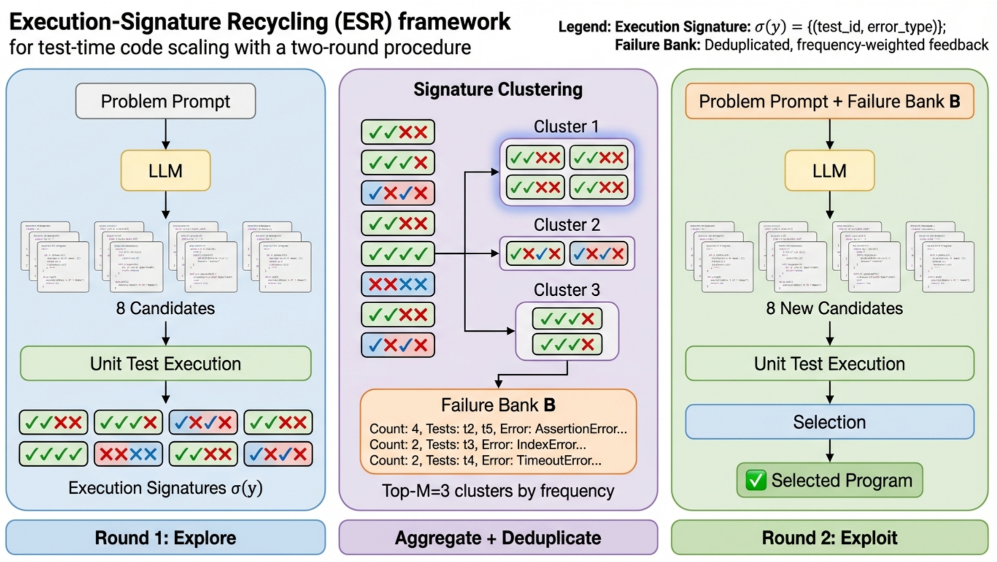
Execution-Signature Recycling (ESR) framework
for test-time code scaling with a two-round procedure
Legend: Execution Signature: σ(y) = {(test_id, error_type)};
Failure Bank: Deduplicated, frequency-weighted feedback
Problem Prompt
LLM
8 Candidates
Unit Test Execution
Execution Signatures σ(y)
Round 1: Explore
Signature Clustering
Cluster 1
Cluster 2
Cluster 3
Failure Bank B
Count: 4, Tests: t2, t5, Error: AssertionError...
Count: 2, Tests: t3, Error: IndexError...
Count: 2, Tests: t4, Error: TimeoutError...
Top-M=3 clusters by frequency
Aggregate + Deduplicate
Problem Prompt + Failure Bank B
LLM
8 New Candidates
Unit Test Execution
Selection
Selected Program
Round 2: Exploit

Figure 1: Overview of Execution-Signature Recycling (ESR). Round 1 generates 8 candidates, executes them against unit tests, and clusters failures by execution signature (error type + failing test ID). Round 2 samples 8 additional candidates conditioned on a deduplicated failure bank containing at most $M = 3$ unique signatures with $K = 2$ examples each.

2.4 Design Rationale

ESR aggregates information across candidates to identify common failure patterns, unlike per-sample self-debugging approaches (Chen et al., 2023) that refine each candidate independently. By clustering failures by execution signature and providing deduplicated feedback, ESR aims to reduce redundant exploration while keeping the shared feedback concise. This design is inspired by recent work on experience recycling for test-time scaling (Wang et al., 2026), but replaces semantic deduplication with deterministic, unit-test-grounded execution signatures.

3 Experiments

3.1 Experimental Setup

We evaluate ESR on HumanEval+ (Liu et al., 2023), an extension of the HumanEval benchmark (Chen et al., 2021) with augmented unit tests designed to reduce overfitting. The benchmark contains 164 Python function synthesis tasks. We use the original HumanEval base tests for execution feedback and candidate selection, while evaluating the selected candidate on the full HumanEval+ test suite.

We use Qwen2.5-Coder-7B-Instruct (Hui et al., 2024) as our base model, a strong open-source code generation model. All methods operate under a fixed budget of 16 generations per task. We use temperature 0.7 and top-p 0.95 for sampling, with a maximum of 1024 tokens per generation. Experiments are repeated across three random seeds (42, 123, 456) to estimate variance, and we report mean $\pm$ standard deviation.

Baselines. We compare ESR against two baselines under the same 16-generation budget: (1) **Best-of-16**: Pure parallel sampling that generates 16 independent candidates and selects the one with the highest base-test pass rate. (2) **Self-Debug** (Chen et al., 2023): Per-sample feedback that generates 8 initial candidates, then produces one debug revision for each candidate conditioned on its own execution feedback (error type and traceback for up to 3 failing tests), yielding 16 total candidates.

Evaluation Protocol. For all methods, we select the candidate that maximizes the number of passing base tests, with ties broken by fewer failing tests. The selected candidate is then evaluated on HumanEval+ augmented tests. We report Pass@1, the fraction of tasks where the selected candidate

Table 1: Main results on HumanEval+ (Pass@1 %). ESR achieves the highest mean but does not significantly outperform Self-Debug. Best in **bold**, second-best <u>underlined</u>. * indicates statistical significance (95% CI excludes 0).

Method	Seed 42	Seed 123	Seed 456	Mean $\pm$ Std	Δ vs Best-of-16	Δ vs Self-Debug
Best-of-16	86.59	85.98	85.37	85.98 $\pm$ 0.50	—	—
Self-Debug	86.59	**87.80**	<u>86.59</u>	<u>86.99 $\pm$ 0.70</u>	—	—
ESR (Ours)	**89.02**	<u>87.20</u>	**86.59**	**87.60 $\pm$ 1.27**	+1.63*	+0.61

Table 2: ESR execution signature statistics across seeds. Most tasks ($\sim$68%) have all candidates passing in Round 1, leaving empty failure banks.

Seed	All-Pass Tasks	Non-Empty Failure Bank	% Non-Empty
42	113	51	31.1%
123	109	55	33.5%
456	111	53	32.3%
Mean	**111.0**	**53.0**	**32.3%**

passes all evaluation tests. Statistical significance is assessed using paired bootstrap confidence intervals over 492 task-seed pairs (164 tasks $\times$ 3 seeds).

3.2 Main Results

Table 1 presents the main experimental results. ESR achieves the highest mean Pass@1 of 87.60%, compared to 86.99% for Self-Debug and 85.98% for Best-of-16. However, the improvement of ESR over Self-Debug (+0.61 percentage points) is not statistically significant, with a 95% confidence interval of $[-0.81, 2.24]$ that includes zero. In contrast, ESR significantly outperforms Best-of-16 (+1.63pp, 95% CI $[0.41, 3.05]$), confirming that execution feedback improves test-time scaling for code generation.

Per the pre-registered decision rule, we conclude that ESR does not provide a statistically significant improvement over the best baseline (Self-Debug) on HumanEval+ with Qwen2.5-Coder-7B-Instruct. Both feedback-based methods outperform pure parallel sampling, but the additional complexity of cross-sample signature aggregation does not yield reliable gains over simpler per-sample self-debugging.

3.3 Analysis

Signature Statistics. Table 2 presents ESR's execution signature statistics. Approximately 68% of tasks (111 out of 164 on average) have all 8 Round 1 candidates passing all base tests, resulting in empty failure banks. ESR's feedback mechanism only applies to the remaining $\sim$32% of tasks. Among tasks with failures, the median number of unique execution signatures is 1.0, validating ESR's premise that failure modes cluster strongly. However, this also means that for most failing tasks, there is only one dominant failure pattern to deduplicate.

Variance Analysis. ESR exhibits substantially higher inter-seed variance (std=1.27) compared to Best-of-16 (std=0.50) and Self-Debug (std=0.70). Examining per-seed results reveals inconsistent effectiveness: ESR gains +2.44 percentage points over Self-Debug on seed 42 but loses 0.61pp on seed 123 and ties on seed 456. This inconsistency suggests that ESR's cross-sample aggregation may introduce instability compared to the more robust per-sample approach.

Discussion. Several factors may explain why ESR does not significantly outperform Self-Debug. First, the strong base model (Qwen2.5-Coder-7B-Instruct) already achieves high accuracy, leaving limited room for improvement. Second, the high proportion of all-pass tasks ($\sim$68%) means ESR's feedback mechanism is only active for a minority of problems. Third, Self-Debug's per-sample feedback may be sufficient when failure modes are already diverse across candidates, reducing the benefit of cross-sample deduplication. These findings suggest that simpler per-sample feedback approaches may be preferable for strong code generation models on benchmarks like HumanEval+.

4 Related Work

Test-Time Scaling. Allocating additional inference-time compute can substantially improve LLM performance on reasoning and code generation tasks. Snell et al. (2024) demonstrate that optimal test-time compute scaling can be more effective than scaling model parameters, while Brown et al. (2024) show that repeated sampling with large generation budgets can solve problems that smaller budgets cannot. Self-consistency (Wang et al., 2022) improves reasoning by sampling multiple chains and aggregating answers. Tree of Thoughts (Yao et al., 2023) enables deliberate problem solving through search over intermediate reasoning states. For code generation, AlphaCode (Li et al., 2022) achieves competition-level performance through massive sampling and clustering-based selection.

Self-Debugging and Iterative Refinement. Execution feedback enables iterative improvement of generated code. Chen et al. (2023) introduce self-debugging, where models refine code based on execution traces and error messages. Reflexion (Shinn et al., 2023) extends this with verbal reinforcement learning, maintaining episodic memory of past failures. Self-Refine (Madaan et al., 2023) demonstrates iterative refinement using self-generated feedback across diverse tasks. These methods typically operate on individual candidates, refining each based on its own execution feedback.

Code Generation with Execution Feedback. Unit tests provide strong programmatic signals for code generation. CodeT (Chen et al., 2022) uses model-generated tests to rank candidate solutions. CodeRL (Le et al., 2022) applies reinforcement learning from execution feedback to improve code generation. Recent work on experience recycling (Wang et al., 2026) proposes batched experience banks with semantic deduplication for test-time scaling in mathematical reasoning, though this has not been applied to code generation with unit-test feedback.

Code Generation Benchmarks. HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021) are standard benchmarks for Python code generation. HumanEval+ (Liu et al., 2023) extends HumanEval with augmented tests to reduce overfitting. LiveCodeBench (Jain et al., 2024) provides contamination-resistant evaluation with continuously updated problems.

ESR extends self-debugging by aggregating execution feedback across candidates using signature-based clustering. However, our results show that this cross-sample aggregation does not significantly improve over simpler per-sample approaches on HumanEval+ with strong base models.

5 Conclusion

We proposed Execution-Signature Recycling (ESR), a test-time scaling method that aggregates and deduplicates execution feedback across candidate programs using signature-based clustering. While ESR achieves the highest mean Pass@1 (87.60%) on HumanEval+, it does not significantly outperform the simpler Self-Debug baseline (86.99%), with a 95% confidence interval that includes zero. This negative result suggests that for strong code generation models on benchmarks like HumanEval+, per-sample self-debugging may be sufficient, and the additional complexity of cross-sample feedback aggregation does not provide reliable benefits. Future work could explore ESR on harder benchmarks with more diverse failure modes, weaker base models where feedback may have greater impact, or alternative signature representations that capture finer-grained failure patterns.

References

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, H. Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. Program synthesis with large language models. *ArXiv*, abs/2108.07732, 2021.

Bradley Brown, Jordan Juravsky, Ryan Ehrlich, Ronald Clark, Quoc V. Le, Christopher R'e, and Azalia Mirhoseini. Large language monkeys: Scaling inference compute with repeated sampling. *ArXiv*, abs/2407.21787, 2024.

Bei Chen, Fengji Zhang, A. Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. Codet: Code generation with generated tests. *ArXiv*, abs/2207.10397, 2022.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé, Jared Kaplan, Harrison Edwards, Yura 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, Mo Bavarian, Clemens Winter, P. Tillet, F. Such, D. Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William H. Guss, Alex Nichol, Igor Babuschkin, S. Balaji, Shantanu Jain, A. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, M. Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, I. Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *ArXiv*, abs/2107.03374, 2021.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. *ArXiv*, abs/2304.05128, 2023.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, An Yang, Rui Men, Fei Huang, Shanghaoran Quan, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. Qwen2.5-coder technical report. *ArXiv*, abs/2409.12186, 2024.

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*, abs/2403.07974, 2024.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, S. Savarese, and S. Hoi. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. *ArXiv*, abs/2207.01780, 2022.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom, Eccles, James Keeling, Felix Gimeno, A. D. Lago, T. Hubert, Peter Choy, Cyprien de, Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey, Cherepanov, James Molloy, D. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de, Freitas, K. Kavukcuoglu, and O. Vinyals. Competition-level code generation with alphacode. *Science*, 378:1092 – 1097, 2022.

Jiawei Liu, Chun Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *ArXiv*, abs/2305.01210, 2023.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, S. Welleck, Bodhisattwa Prasad Majumder, Shashank Gupta, A. Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback. *ArXiv*, abs/2303.17651, 2023.

Noah Shinn, Federico Cassano, Beck Labash, A. Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: language agents with verbal reinforcement learning. 2023.

C. Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling llm test-time compute optimally can be more effective than scaling model parameters. *ArXiv*, abs/2408.03314, 2024.

Xinglin Wang, Jiayi Shi, Shaoxiong Feng, Peiwen Yuan, Yiwei Li, Yueqi Zhang, Chuyi Tan, Ji Zhang, Boyuan Pan, Yao Hu, and Kan Li. Do not waste your rollouts: Recycling search experience for efficient test-time scaling. 2026.

Xuezhi Wang, Jason Wei, D. Schuurmans, Quoc Le, Ed H. Chi, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. *ArXiv*, abs/2203.11171, 2022.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, T. Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *ArXiv*, abs/2305.10601, 2023.