TEX: Test-Time Scaling Testing Agents via Execution-based Cross-Validation

Akash Gokul[1*†], Srijan Bansal[2*], Yingbo Zhou[3†], Semih Yavuz[2]
[1]NVIDIA, [2]Salesforce Research, [3]Recursive
[*]Equal contribution. [†]Work done while at Salesforce Research.
Correspondence: srijanbansal@salesforce.com

Abstract

Test-case generation is critical but remains challenging for language models. Existing test-time scaling methods either isolate candidates or aggregate traces, which do not scale to long-horizon tasks. We present TEX, a collaborative test-time scaling framework based on execution-driven cross-validation. TEX parallelizes agents to generate tests and patches, cross-executing each test against every patch to form a $K \times K$ feedback matrix. Agents use their matrix slices for refinement, enabling objective validation without verbose trace sharing. On SWT-Bench Verified, TEX achieves *86.4%*, establishing a new state-of-the-art and improving +9.0% over single-agent baselines. Moreover, even random selection from TEX outputs (84.4%) surpasses the previous SOTA (84.0%), showing collaborative execution feedback improves solution quality, not just best-case samples. Our results highlight execution-grounded feedback as an effective scaling mechanism for long-horizon agentic tasks.

1 Introduction

Software engineering agents have made remarkable progress in bug-fixing (Jimenez et al., 2024; Deng et al., 2025), yet *generating high-quality tests* remains challenging. Test-case generation demands both diversity (exploring different test designs) and validation (identifying which tests correctly detect bugs). Current approaches struggle to provide both: isolated generation lacks diversity, while parallel sampling faces a verification gap when no oracle tests exist (Li et al., 2026).

Test-time scaling improves large language model (LLM) performance through inference-time computation (OpenAI, 2024b; Guo et al., 2025; Snell et al., 2025). However, existing methods have limitations. Serial scaling limits diversity and suffers from overthinking (Zhou et al., 2026). Parallel scaling (Best-of-N) requires verifiers (Xi et al., 2026; Jin et al., 2026). Hybrid approaches (Venkatraman et al., 2025; Madaan et al., 2025) share information across agents but either concatenate full traces (prohibitive at 100K+ tokens) or summarize them (losing details). Moreover, multi-agent collaboration often underperforms single agents under equal budgets (Tran and Kiela, 2026), suggesting inefficient aggregation.

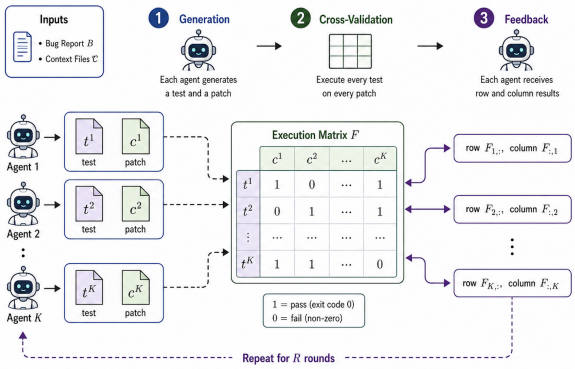

Figure 1: TEX overview. At each round, K agents are sampled in parallel and each agent outputs a test file and code patch. We perform cross-validation by running all generated code patches against all generated test files. Each agent receives execution feedback from running its code & test against all other tests & code, then continues sampling in the next round.

For test generation, these limitations are acute. Verifier-based methods work when correctness is verifiable, but test quality verification requires oracle code execution. Self-execution frameworks (Zheng et al., 2024; Ding et al., 2024) run agents' self-generated tests on their own code, providing limited supervision, as failures do not disambiguate between incorrect tests and incorrect code.

We hypothesize that this can be addressed through *collaborative validation*: agents generate diverse solutions and learn from each other's execution results. We present TEX, which achieves this via *execution-based collaborative learning*. TEX samples K agents in parallel; each generates both test scripts and code patches across multiple

Findings of the Association for Computational Linguistics: EMNLP 2026, pages 46889–46894
October 24-29, 2026 ©2026 Association for Computational Linguistics

rounds. Rather than sharing reasoning traces, we execute every agent's tests against every agent's code, producing a compact $K \times K$ feedback matrix (Figure 1). Each agent receives execution results for cross-candidate refinement. This provides: *(i)* objective verification without learned reward models or oracle verifiers; *(ii)* richer signals than self-execution alone; and *(iii)* compact feedback. TEX achieves **86.4%** on SWT-Bench Verified, improving **+9.0%** over single-agent baselines while maintaining diversity (Best-of-4 stable at ∼89%). Effective test-time scaling needs the *right signal*: execution feedback, not more context.

2 Methodology: TEX

TEX enables collaborative test-case generation through execution-based cross-validation. Rather than sharing reasoning traces, agents share execution results—a grounded signal for long-horizon tasks. Our core hypothesis is that test-case generation benefits from two mechanisms: (1) *active code generation*—agents that write both tests and bug fixes produce better tests than test-only approaches, as understanding the fix helps identify what to test; and (2) *cross-candidate learning*—agents refine their solutions more effectively when they can see how their tests and code patches perform against their peers', rather than in isolation.

Method Overview. Given a bug report, TEX runs K agents in parallel over R rounds (Algorithm 1). Each round proceeds in three phases. **Generation:** each agent i independently generates test script T_i and code patch C_i (up to $M = 8$ steps each) based on the bug report, relevant context files, and execution feedback from the previous round ($\mathbf{F}^{r-1}_{i,:}, \mathbf{F}^{r-1}_{:,i}$). Tests aim to reproduce the bug (fail on buggy code); patches aim to fix it. **Cross-Validation:** we execute every agent's test against every agent's code patch, building execution feedback matrix $\mathbf{F} \in \{0,1\}^{K\times K}$ where $F_{ij} = 1$ if test T_i passes on code C_j (exit code 0), else $F_{ij} = 0$ (non-zero exit). This captures all K^2 pairwise interactions—does agent i's test correctly validate agent j's fix?—providing multi-source validation unavailable in self-execution. **Feedback:** each agent i receives row $\mathbf{F}_{i,:}$ (how its test performed on all K patches) and column $\mathbf{F}_{:,i}$ (how its code performed against all K tests), totaling $2K$ compact execution results. If agent i's test fails on all patches, it may be overly strict; if its code fails most tests, it likely doesn't fix the bug. This signal lets agents distinguish test errors from code errors and refine both in the next round.

Algorithm 1 TEX: Test-Case Generation via Execution Feedback

Input: Bug report B, context files $\mathcal{C}$, ensemble size K, rounds R
for round $r = 1$ to R **do**
  // Generation Phase
  for agent $i = 1$ to K **in parallel do**
    $T_i^r \leftarrow$ GenerateTest($B, \mathcal{C}, \mathbf{F}^{r-1}_{i,:}, \mathbf{F}^{r-1}_{:,i}$)
    $C_i^r \leftarrow$ GeneratePatch($B, \mathcal{C}, \mathbf{F}^{r-1}_{i,:}, \mathbf{F}^{r-1}_{:,i}$)
  end for
  // Cross-Validation Phase
  for $i, j = 1$ to K **do**
    $F_{ij}^r \leftarrow$ Execute(T_i^r, C_j^r) *// 1 if pass, 0 if fail*
  end for
  // Feedback Phase: $\mathbf{F}^r$ *provided to agents in next round*
end for
// Selection Strategy
Return: MajorityVote($\{T_1^R, \ldots, T_K^R\}, \{C_1^R, \ldots, C_K^R\}$)

Selection Strategy. After R rounds, we select the final test via majority voting over execution results. We first filter tests that fail to detect the bug (exit code $= 0$ on pre-patch code), then select the one that passes (exit code $= 0$) most frequently across all K generated patches—frequent passing indicates robustness and agreement with multiple independent fixes. If no valid test remains, we randomly select one. This execution-based selection avoids learned verifiers while leveraging code patches as natural validation signals.

3 Experiments

Setup. We use the 433-instance subset of SWE-Bench Verified (OpenAI, 2024a; Jimenez et al., 2024) introduced by Mündler et al. (2024) for bug-fixing (TEX-C) and SWT-Bench Verified (Mündler et al., 2024) for test-case generation (TEX-T). This subset excludes instances whose gold test suites are flaky or otherwise unreliable; because such issues originate in the benchmark's gold tests and are often unsolved (as seen in SWE-Bench's GitHub issues), they inject noise into any submission's score. We extend the CodeMonkeys scaffold (Ehrlich et al., 2025) to support cross-candidate execution, using Claude Sonnet 4 (temperature=0.5, $K = 4$ agents, $R = 3$ rounds, 8 steps per test and code generation respectively). We intentionally use a simple scaffold—no bash, internet, or kernel—to isolate TEX's contribution from tool richness. Context files are from Agentless (Xia et al., 2025). Baselines: (1) *Single-agent* (R0): prompting without cross-candidate feedback; (2) *Best-of-N*: oracle se-

Method	Pass@1	Best-of-4
Baselines		
Test-Only	71.3%	–
Test + Code (R0)	77.4%	88.7%
TEX-T		
Round 1	84.1%	88.7%
Round 2	84.1%	88.9%
Round 3	84.4%	89.2%
TEX-T + Majority Vote	**86.4%**	–
Previous SOTA (L*Agent v1)	84.0%	–

Table 1: SWT-Bench Verified results.

Method	Pass@1	Best-of-4
Baseline		
Code Generation (R0)	57.8%	68.4%
TEX-C		
Round 1	58.9%	67.7%
Round 2	59.4%	67.8%
Round 3	59.8%	67.9%

Table 2: SWE-Bench Verified results.

lection upper bound; (3) *Self-feedback*: agents refine using only their own test/code pairs.

SWT-Bench: State-of-the-art test generation. TEX-T achieves **86.4%** on SWT-Bench Verified, establishing a new state-of-the-art and improving +9.0% over single-agent baselines (Table 1). We observe two findings supporting our hypotheses. First, **active code generation provides substantial gains**: the test-only baseline achieves 71.3%, while adding code generation improves performance to 77.4% (+6.1%). This validates hypothesis 1—agents generating both tests and fixes produce better tests because understanding the fix informs what to test. Second, **cross-candidate learning provides immediate value**: Pass@1 improves from 77.4% (R0) to 84.1% after one round (+6.7% from cross-candidate feedback alone), then plateaus at 84.4% by R3. This shows that multi-source validation against diverse peer solutions provides richer signal than isolated refinement.

Critically, TEX achieves 84.4% with *random* selection, surpassing the previous SOTA of 84.0% (L*Agent v1; LogicStar AI, 2025). This shows cross-candidate learning reduces variance and improves quality, not just best-case outputs. Majority voting, selecting tests that pass most frequently across code patches, improves to 86.4%, using execution results as validation signals *without* LLM judges or learned verifiers. Best-of-4 performance remains stable across rounds (88.7%→89.2%), confirming TEX preserves diversity.

SWE-Bench: Generalization to bug-fixing. To verify execution-based collaboration generalizes beyond test generation, we evaluate TEX-C on bug-fixing. On SWE-Bench Verified, TEX-C improves Pass@1 from 57.8% to 59.8% across three rounds (Table 2), showing consistent gains each round (R1: 58.9%, R2: 59.4%, R3: 59.8%). This progression differs from SWT-Bench's plateau after R1, suggesting bug-fixing benefits from sustained refinement while test generation converges faster. This aligns with intuition: fixing bugs requires iterating on complex logic and edge cases, whereas writing tests involves clearer specifications that converge more quickly. The steady improvement validates that cross-candidate feedback continues providing value even when initial gains are smaller. Our baseline (57.8%) is 7.1% below mini-swe-agent (Yang et al., 2024) (64.9%) due to our simple scaffold lacking bash, internet, and kernel access. This isolates TEX's gains (+2.0% absolute) from scaffold improvements, showing cross-candidate learning is independent of tool richness. This isolation is valuable for understanding the method's contribution and suggests future work: combining TEX with more capable scaffolds could reach 66%–67% simply by using TEX with mini-swe-agent's harness.

Cost-Quality tradeoffs. TEX exchanges inference compute for solution quality: running K agents for R rounds costs 12× a single agent (as we use $K = 4$, $R = 3$) and buys +9.0%. The exchange compares favorably to Best-of-4 (4×), which reaches 88.7% only under oracle selection; TEX attains 86.4% using majority voting over execution signals, recovering most of that gap without LLM judges or learned verifiers. The curve is strongly concave: performance plateaus after Round 1 on SWT-Bench (84.1%→84.4%), so stopping at $R = 1$ saves 67% of the cost for 0.3% quality loss, and is still well above the single-agent baseline (84.1% vs. 77.4%).

4 Analysis

4.1 Why Cross-Candidate Learning Works

To isolate whether TEX's gains stem from cross-candidate signals rather than extra compute, we compare against *self-feedback*, where agents receive only their own test/code results. Both use identical compute ($K = 4$, $R = 3$), differing only in feedback scope. Figure 2 shows results on a

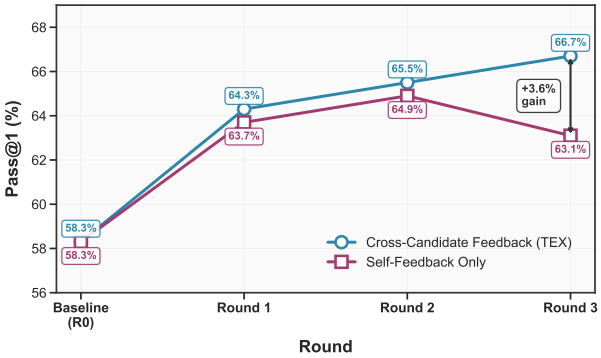

Figure 2: Ablation on SWT-Bench subset shows cross-candidate feedback (TEX) outperforms self-feedback under identical compute ($K = 4$, $R = 3$).

42-instance SWT-Bench subset. Self-feedback improves initially but plateaus at Round 2 (64.9%), then drops to 63.1% at Round 3, indicating overfitting without external validation. In contrast, cross-candidate feedback sustains gains across rounds (58.3% at R0 $\rightarrow$ 66.7% at R3).

Three mechanisms explain this advantage. (1) *Multi-source validation*: when agent i's test fails on agent j's code but passes on agent k's, it learns the test may be overly strict; self-feedback lacks this triangulation and cannot distinguish "my test is wrong" from "my code is wrong." (2) *Exposure to alternatives*: cross-candidate feedback avoids local optima by exposing agents to peers' approaches through execution results. (3) *Scalable aggregation*: TEX shares $O(K)$ pass/fail results versus $O(K \cdot L)$ tokens for trace concatenation (Venkatraman et al., 2025; Madaan et al., 2025), where L often exceeds 100K for software agents.

Difficulty stratification explains performance patterns. Full SWT-Bench plateaus after Round 1 (84.1%$\rightarrow$84.4%), while the ablation continues improving. We hypothesize easy instances converge quickly, while harder ones require more rounds. This suggests adaptive stopping: stop at $R = 1$ for converged instances (67% cost reduction), and continue where agents disagree.

5 Background and Related Work

Test-Time Compute Scaling. Recent work explores allocating more computation at inference time (Snell et al., 2025). Recursive self-aggregation (Venkatraman et al., 2025) refines outputs by randomly subsampling and aggregating across parallel samples. Iterative refinement methods (Madaan et al., 2025) improve solutions across multiple rounds. For agentic tasks, General AgentBench (Li et al., 2026) evaluates test-time scaling, identifying context ceiling and verification gap as bottlenecks. Process reward models (Xi et al., 2026) and self-evolving frameworks (Jin et al., 2026) address verification through learned value functions.

Execution-Based Code Generation. OpenCodeInterpreter (Zheng et al., 2024) and CYCLE (Ding et al., 2024) integrate code generation with execution feedback, where models iteratively refine outputs based on test results. These frameworks show execution provides objective validation signals superior to natural language feedback alone. However, they focus on single-agent self-correction rather than multi-agent collaboration.

Software Engineering Benchmarks and Agents. SWE-Bench (Jimenez et al., 2024) evaluates bug fixing, while SWT-Bench (Mündler et al., 2024) evaluates test generation. Prior methods use stronger scaffolds (SWE-Agent (Yang et al., 2024), SWE-Gym (Pan et al., 2025)) or verifier-based selection ((Shum et al., 2025; Xi et al., 2026)). TEX instead enables lightweight communication through cross-candidate execution outcomes.

Test Generation Methods. Code-T (Chen et al., 2023) first demonstrated iterative test generation through execution feedback loops. For SWT-Bench, published work includes e-Otter++ (Ahmed et al., 2026), which relies on off-the-shelf code patches to validate generated tests without agents solving the underlying code bug, and L*Agent v1 (LogicStar AI, 2025)—a closed-source approach.

6 Conclusion

We present TEX, a test-time scaling method achieving **86.4%** on SWT-Bench (**first place**) through collaborative learning via execution feedback. Instead of sharing verbose reasoning traces—impractical for long-horizon software tasks—TEX exchanges compact pass/fail outcomes, enabling scalable iterative refinement for agents generating hundreds of thousands of tokens per instance. Our analysis shows: (1) cross-candidate learning outperforms self-feedback through multi-source validation; (2) collaboration preserves solution diversity rather than causing convergence; and (3) adaptive stopping could reduce cost by 67% with minimal loss. TEX demonstrates that effective test-time scaling for complex agents requires not more context, but sharing the right structured signals—execution feedback instead of natural language traces.

Limitations

TEX's 12× cost ($K = 4$, $R = 3$) could be reduced via: (1) adaptive stopping at $R = 1$ (67% reduction, 0.3% loss), (2) dynamic ensemble sizing by instance difficulty, (3) partial cross-validation (computing a subset of the $K \times K$ matrix). Our simple scaffold isolates TEX's contribution but limits absolute SWE-Bench performance; richer scaffolds (bash, search tools, debugging kernels) could yield larger gains. Our Python-only evaluation requires validation on other languages (Java, C++, JavaScript) and domains beyond GitHub—while execution feedback is language-agnostic in principle, empirical generalization remains future work. We fixed $K = 4$ due to budget constraints; systematic exploration of $K \in \{2, 4, 8, 16\}$, together with confidence intervals over repeated runs, could reveal optimal tradeoffs but remains beyond our compute budget; all reported numbers are single-run results.

Potential risks. TEX amplifies the capabilities of LLM-based test generation agents, which raises concerns shared broadly by autonomous test-case generation systems: generated tests may be overly permissive (missing critical edge cases or security vulnerabilities) or overly strict (rejecting valid implementations), and these issues may be difficult to detect without domain expertise. Because TEX employs cross-candidate learning where agents observe each other's execution results, there is a risk that systematic biases or blind spots shared across the agent ensemble could be reinforced rather than corrected—if all agents miss the same class of bugs, cross-validation provides no corrective signal. Additionally, agreement-based selection could amplify common failure modes if most agents converge to similar but flawed test designs. We recommend that TEX-generated tests undergo the same review and validation processes applied to human-authored tests, including manual inspection for completeness, correctness, and security considerations before deployment in production systems.

Acknowledgments

We would like to thank Hiroaki Hayashi and Wenting Zhao for useful discussions. We would also like to thank Srinath Reddy for infrastructure support. The authors used LLM-based tools (Claude Code) to assist with manuscript editing, including rewording, restructuring sections, condensing verbose text, and typographical corrections. All scientific content, experimental design, implementation, analysis, and interpretation were conducted by the authors.

References

Toufique Ahmed, Jatin Ganhotra, Avraham Shinnar, and Martin Hirzel. 2026. Heterogeneous prompting and execution feedback for SWE issue test generation and selection. In *Proceedings of the 48th IEEE/ACM International Conference on Software Engineering (ICSE 2026)*. To appear.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2023. CodeT: Code generation with generated tests. In *The Eleventh International Conference on Learning Representations (ICLR 2023)*.

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, Karmini Sampath, Maya Krishnan, Srivatsa Kundurthy, Sean Hendryx, Zifan Wang, Vijay Bharadwaj, Jeff Holm, Raja Aluri, Chen Bo Calvin Zhang, and 3 others. 2025. SWE-Bench Pro: Can AI agents solve long-horizon software engineering tasks? *Preprint*, arXiv:2509.16941.

Yangruibo Ding, Marcus J. Min, Gail Kaiser, and Baishakhi Ray. 2024. Cycle: Learning to self-refine the code generation. *Proc. ACM Program. Lang.*, 8(OOPSLA1).

Ryan Ehrlich, Bradley Brown, Jordan Juravsky, Ronald Clark, Christopher Ré, and Azalia Mirhoseini. 2025. CodeMonkeys: Scaling test-time compute for software engineering. *Preprint*, arXiv:2501.14723.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, and 175 others. 2025. DeepSeek-R1 incentivizes reasoning in LLMs through reinforcement learning. *Nature*, 645(8081):633–638.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2024. SWE-bench: Can language models resolve real-world Github issues? In *The Twelfth International Conference on Learning Representations*.

Yiyang Jin, Kunzhao Xu, Hang Li, Xueting Han, Yanmin Zhou, Cheng Li, and Jing Bai. 2026. Reveal: Self-evolving code agents via reliable self-verification. In *International Conference on Learning Representations*, volume 2026, pages 149800–149824.

Xiaochuan Li, Ryan Ming, Pranav Setlur, Abhijay Paladugu, Andy Tang, Hao Kang, Shuai Shao, Rong

Jin, and Chenyan Xiong. 2026. Benchmark test-time scaling of general LLM agents. *Preprint*, arXiv:2602.18998.

LogicStar AI. 2025. SWT-Bench Verified – Best Test Generation at 84%. Blog post. Published 10 September 2025.

Lovish Madaan, Aniket Didolkar, Suchin Gururangan, John Quan, Ruan Silva, Ruslan Salakhutdinov, Manzil Zaheer, Sanjeev Arora, and Anirudh Goyal. 2025. Rethinking thinking tokens: LLMs as improvement operators. *Preprint*, arXiv:2510.01123.

Niels Mündler, Mark N Müller, Jingxuan He, and Martin Vechev. 2024. Swt-bench: Testing and validating real-world bug-fixes with code agents. *Advances in Neural Information Processing Systems*, 37:81857–81887.

OpenAI. 2024a. Introducing SWE-bench Verified. OpenAI Blog. Published 13 August 2024.

OpenAI. 2024b. OpenAI o1 system card. *Preprint*, arXiv:2412.16720.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. 2025. Training software engineering agents and verifiers with SWE-Gym. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pages 39532–39577.

Ka Lam Shum, Yichi Zhang, Xinrun Du, Yongchao Hao, Chengpeng Li, Xingyao Wang, Graham Neubig, Xuehai Pan, Nan Jiang, Jiacheng Liu, Yizhe Zhang, Qian Huang, and Jiayi Pan. 2025. SWE-RM: Execution-free feedback for software engineering agents. *Preprint*, arXiv:2512.21919.

Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. 2025. Scaling LLM test-time compute optimally can be more effective than scaling parameters for reasoning. In *The Thirteenth International Conference on Learning Representations (ICLR 2025)*.

Dat Tran and Douwe Kiela. 2026. Single-agent LLMs outperform multi-agent systems on multi-hop reasoning under equal thinking token budgets. *Preprint*, arXiv:2604.02460.

Siddarth Venkatraman, Vineet Jain, Sarthak Mittal, Vedant Shah, Johan Obando-Ceron, Yoshua Bengio, Brian R. Bartoldson, Bhavya Kailkhura, Guillaume Lajoie, Glen Berseth, Nikolay Malkin, and Moksh Jain. 2025. Recursive self-aggregation unlocks deep thinking in large language models. *Preprint*, arXiv:2509.26626.

Zhiheng Xi, Chenyang Liao, Guanyu Li, Zhihao Zhang, Wenxiang Chen, Binghai Wang, Senjie Jin, Yuhao Zhou, Jian Guan, Wei Wu, Tao Ji, Tao Gui, Qi Zhang, and Xuanjing Huang. 2026. Agentprm: Process reward models for llm agents via step-wise promise and progress. In *Proceedings of the ACM Web Conference 2026*, WWW '26, page 4184–4195, New York, NY, USA. Association for Computing Machinery.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2025. Agentless: Demystifying LLM-based software engineering agents. *Proceedings of the ACM on Software Engineering*, 2(FSE):801–824.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-computer interfaces enable automated software engineering. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.

Tianyu Zheng, Ge Zhang, Tianhao Shen, Xueling Liu, Bill Yuchen Lin, Jie Fu, Wenhu Chen, and Xiang Yue. 2024. OpenCodeInterpreter: Integrating code generation with execution and refinement. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 12834–12859, Bangkok, Thailand. Association for Computational Linguistics.

Shu Zhou, Rui Ling, Junan Chen, Xin Wang, Tao Fan, and Hao Wang. 2026. When more thinking hurts: Overthinking in LLM test-time compute scaling. In *Findings of the Association for Computational Linguistics: ACL 2026*, pages 23967–23977, San Diego, California, United States. Association for Computational Linguistics.