Weak-to-Strong Temporal Verification for Coding Agents

Junye Pan
junyep2@illinois.edu

Abstract

Repository-level software engineering agents are commonly scaled at test time by running multiple independent trajectories and then selecting one completed patch. This improves the chance that at least one branch solves the issue, but it is expensive because every branch consumes a full rollout budget even when its intermediate behavior is already unpromising. This report studies a more selective alternative: weak-to-strong trajectory verification. The system generates multiple SWE-agent trajectories for the same issue, renders their trajectory and patch evidence, compares candidate pairs with a weaker judge model, and aggregates the pairwise preferences into a single selected patch.

The current experiment is intentionally preliminary. It evaluates a 50-instance diagnostic subset of SWE-bench Verified, selected as a relatively difficult subset for studying candidate selection rather than as a random estimate of full-benchmark performance. The present run uses full-trajectory checkpoints rather than early partial checkpoints. On this subset, the pass@1 baseline solves 31/50 instances (62.0%), the weak-judge selected top-1 patch using `gpt-oss-120b` solves 33/50 instances (66.0%), and an oracle pass@4 baseline solves 35/50 instances (70.0%). Thus the verifier recovers two of the four additional solved instances available from four-rollout sampling, or roughly half of the observed pass@4 gap over pass@1. These results provide directional evidence that trajectory-level weak verification can improve single-patch selection, but the small sample size, hard-subset construction, use of only full trajectories, and missing compute-normalized accounting prevent stronger claims.

Project Repository: GitHub repository

Keywords

software engineering agents, test-time scaling, trajectory verification, SWE-bench, weak verification, pairwise ranking

1 Introduction

Repository-level issue resolution requires an agent to understand an unfamiliar codebase, localize relevant files, infer the root cause of a failure or missing behavior, edit the repository, and validate the proposed patch. SWE-bench introduced this setting using real GitHub issues and pull requests, and SWE-bench Verified further filtered the benchmark with human review to improve clarity, test correctness, and solvability [2, 5, 14]. Compared with isolated code-generation benchmarks, this setting stresses long-context repository understanding, tool use, multi-step debugging, and executable validation.

A common way to improve SWE-agent performance at test time is independent sampling. Given an issue, the system launches several complete trajectories, obtains multiple candidate patches, and then uses either an oracle, a verifier, a reviewer model, or a heuristic policy to select a final answer. Independent sampling is attractive because different trajectories can localize different files or explore different repair hypotheses. However, it is also compute-inefficient: every branch pays the cost of a full run, including branches that show weak localization, repeated failed commands, irrelevant exploration, or low-quality edits early in the trajectory.

This project asks whether a verifier can identify promising SWE-agent trajectories before or at completion and thereby recover some of the benefit of multi-rollout sampling while selecting only one final patch. The key distinction from ordinary patch verification is that the judge is not limited to the final diff. It sees trajectory evidence: files inspected, commands executed, test outputs, edits, failed hypotheses, final answer text, and the final patch. The central hypothesis is that this evidence contains useful signal about eventual correctness, and that a weaker or cheaper judge can exploit this signal even when it is not itself generating the patch.

The long-term target is *temporal* verification over partial trajectories. The current experiment is more limited: it evaluates full-trajectory checkpoints only. This limitation is important. The current result tests whether weak trajectory-aware selection improves top-1 patch choice, but it does not yet prove that early pruning improves the compute-success tradeoff.

2 Related Work

2.1 Benchmarks and SWE agents

SWE-bench frames software engineering as repository-level issue resolution: a model receives an issue description and a repository snapshot, then must submit a patch that passes hidden benchmark tests [5]. SWE-bench Verified is a 500-instance human-filtered subset intended to provide more reliable evaluation by checking task clarity, test correctness, and solvability [2, 14]. This report uses SWE-bench Verified because it remains a standard setting for comparing coding agents, while also treating it cautiously because recent work has raised concerns about benchmark saturation, contamination, and test-based validity [9, 11, 15].

SWE-agent showed that agent-computer interface design can substantially improve autonomous repository interaction by giving language-model agents more effective tools for navigation, editing, and test execution [17]. AutoCodeRover approaches the same problem from a software-engineering perspective, combining LLMs with code search, program structure, and fault-localization signals [20]. Agentless showed that a carefully structured localization-repair-validation workflow can remain competitive without a fully autonomous agent loop [16]. SWE-Gym further studies training environments and verifiers for software engineering agents, explicitly connecting agent trajectories, executable feedback, and inference-time selection [10]. These systems motivate this report's focus on trajectory evidence rather than only final diffs.

2.2 Trajectory analysis and failure modes

Recent trajectory analyses suggest that successful and unsuccessful SWE-agent runs often differ before termination. Bouzenia and

Pradel study thought-action-result trajectories and identify structural behavior patterns, anti-patterns, and feedback-use differences across agent runs [1]. This supports the premise that intermediate evidence can be useful for verification. If failure modes are visible in the sequence of search actions, commands, edits, and validation attempts, then a judge may be able to rank candidates before all compute has been spent.

2.3 Test-time scaling and verification

Inference-time scaling is now a central technique for improving reasoning and coding systems. In SWE settings, Satori-SWE studies evolutionary test-time scaling to make sampling more sample-efficient [19]; Agentic Rubrics proposes repository-grounded rubrics as contextual verifiers for candidate patches [12]; SWE-Replay studies efficient replay and branching from archived trajectories rather than naive resampling from scratch [3]; and R2E-Gym studies hybrid verification that combines execution-based and execution-free signals [4]. In broader reasoning, V_1 argues that pairwise verification and tournament-style ranking can be stronger and more efficient than pointwise scalar scoring [13]. This project is closest to the verification-and-selection branch of this literature: it uses a weak judge to compare candidate SWE trajectories and select a single final patch.

3 Method

3.1 System overview

The system is a progressive partial-trajectory search framework. The input is a repository and issue statement; the output is one final patch. Rather than treating candidate generation and patch selection as a single monolithic run, the system separates them into three stages:

(1) generate multiple candidate SWE-agent trajectories for the same issue;
(2) render candidate trajectory evidence into judge prompts and compare candidates pairwise;
(3) aggregate pairwise preferences and submit the top-ranked candidate patch.

The broader design supports pausing, checkpointing, ranking, pruning, and resuming branches. The current experiment uses only full-trajectory checkpoints, but the same artifact format is intended to support earlier temporal checkpoints once those runs are complete.

3.2 Pause and checkpoint mechanism

The planned temporal system pauses only at turn boundaries. A turn consists of one assistant response, the tool calls emitted by that response, and the corresponding tool outputs. The current deterministic pause policy uses structural trigger families: completed repository edits and reproduction or validation commands. A cooldown rule suppresses very early pauses so that the system does not stop after trivial setup actions.

When a branch pauses, the system stores a checkpoint rather than a live session handle. Each checkpoint contains raw and simplified trajectory history, pause metadata, and a recoverable repository-state snapshot. The snapshot is represented as an incremental diff chain plus an overlay of changed files. A selected branch can be resumed by reconstructing the workspace from the base repository, replaying the diff chain, materializing overlay files, and launching a fresh worker session. This design ensures that selection operates over explicit, reproducible search artifacts rather than hidden backend state.

Table 1: Temporal checkpoint definitions. The current preliminary experiment uses only `C4`.

ID	Name	Prefix definition
`C1`	First edit	Through first completed repository edit
`C2`	First test	Through first test command
`C3`	Last test	Through last test command
`C4`	Full trajectory	Complete simplified trajectory plus final patch

3.3 Candidate trajectory artifacts

For each issue, the replay pipeline discovers candidate attempts from attempt directories. Each attempt includes the agent trajectory, tool outputs, and final patch when available. This makes verifier experiments reusable: prompt rendering, pairwise judging, aggregation, and ablations can be rerun without regenerating candidate trajectories.

The intended checkpoint definitions are shown in Table 1. Earlier checkpoints expose partial progress, while `C4` exposes the complete simplified trajectory and final patch. The present 50-instance experiment uses only `C4`. This choice isolates the verification question—whether trajectory-aware weak judging can select a better completed candidate—but does not yet evaluate early pruning.

3.4 Pairwise weak judge

For each issue, the judge receives the issue statement and rendered evidence for two candidates, denoted Candidate A and Candidate B. The judge answers a comparative question: which candidate is more likely to be correct, given the trajectory evidence and final patch? The judge is instructed to emit a single score token from a fixed legal set:

$$\mathcal{S} = \{A, B, C, D, E, F, G, H, I, J\}. \tag{1}$$

The score token encodes both preference direction and confidence. Instead of relying only on the emitted visible token, the system records the probability distribution over the legal A–J alternatives whenever token log probabilities are available. The conceptual output artifact is:

```
{
  "judge_output_token": "H",
  "legal_token_coverage": 0.97,
  "score_distribution": {
    "A": ..., "B": ..., "C": ..., "D": ..., "E": ...,
    "F": ..., "G": ..., "H": ..., "I": ..., "J": ...
  }
}
```

The `legal_token_coverage` diagnostic measures the amount of next-token probability mass captured by the legal score tokens. Low

coverage suggests that the model may prefer to emit explanations, invalid symbols, or other non-score text, making the comparison less reliable.

The current 50-instance run uses `gpt-oss-120b` as the weak judge. The model is used only for evaluation and selection, not for patch generation. This matters because the experiment measures whether a judge can extract selection signal from candidate trajectories produced by SWE agents. OpenAI describes `gpt-oss-120b` as an open-weight reasoning model with 117B total parameters, 5.1B active parameters per token, and a 128k context length [8].

3.5 From score probabilities to pairwise margins

The A–J score distribution is converted into a signed pairwise margin. Probability mass assigned to tokens favoring Candidate A contributes positive evidence; probability mass assigned to tokens favoring Candidate B contributes negative evidence; and neutral or low-confidence tokens contribute smaller absolute margin. This produces a weighted directed comparison rather than a hard win/loss label.

Using probability-weighted margins has two advantages. First, it preserves uncertainty: a near-tie comparison should not count as strongly as a confident preference. Second, it reduces sensitivity to a single sampled output token because the ranking uses the judge's local probability distribution over the legal score alternatives.

3.6 Weighted Kemeny aggregation

For each issue, pairwise margins are aggregated with exact weighted Kemeny ranking. A Kemeny ranking is an ordering that maximizes agreement with pairwise preferences, and has a long history in rank aggregation and social choice [6, 18]. Because the number of candidates per issue is small, exact enumeration is feasible in the current setting.

In a fully populated eight-candidate design, one prompt variant requires

$$\binom{8}{2} = 28 \tag{2}$$

pairwise judge calls per issue. With two prompt variants and one checkpoint, this gives

$$28 \times 2 = 56 \tag{3}$$

judge calls per issue. The present 50-instance preliminary table should be read as a four-candidate diagnostic evaluation; the eight-candidate full-pairwise protocol is reserved for the planned scale-up. This distinction is important because an oracle baseline must use the same candidate pool size as the verifier in any final cost-normalized comparison.

3.7 Prompt variants and context budgeting

The current design uses two prompt variants:

- **Root-cause trace:** emphasizes whether the candidate identified the correct failure mechanism, localized the right files, and edited the relevant code.
- **Process monitorability:** emphasizes whether the trajectory shows reliable debugging behavior, validation attempts, and convergence rather than superficial exploration.

The variants are aggregated at the pairwise-margin level so that each prompt contributes complementary evidence without requiring either prompt to be individually perfect.

Full `C4` prompts can be long because they contain the task statement, two simplified trajectories, and two final patches. For local 32k-context models, prompt budgeting must be pair-level rather than only single-trajectory-level. The deterministic truncation policy is label-blind. It prioritizes the task statement, final patches, final answers, validation evidence, test commands, error summaries, edit context, and then compresses repetitive logs before middle truncation. For every truncated prompt, the system should record estimated prompt length, original and rendered trajectory length, patch length, whether either patch was truncated, and the truncation reason.

4 Evaluation Setup

4.1 Benchmark and subset

The evaluation uses SWE-bench Verified, a 500-instance human-filtered subset of SWE-bench [2, 14]. The current result is computed on 50 instances. These 50 instances should be described as a relatively difficult diagnostic subset, not as a random sample of the full benchmark. They are useful for stress-testing candidate selection because even the oracle over four generated attempts solves only 35/50 instances, leaving 15/50 unresolved despite multiple trajectories. This means the subset contains both candidate-generation failures and candidate-selection failures.

This distinction affects interpretation. The 50-instance result can show whether the verifier has signal on a hard selection set, but it cannot estimate full SWE-bench Verified performance. It also cannot establish statistical significance by itself. Any final claim about benchmark-level performance should use the larger selected set or the full 500-instance benchmark and should report paired per-instance outcomes.

4.2 Candidate source and label separation

For each instance, multiple candidate trajectories are generated by the underlying SWE-agent worker. The verifier ranks candidates without access to the official resolved labels. After the selected candidate is written, the patch is evaluated using the official SWE-bench execution harness. Rankings should be produced and cached before labels are joined, so that selection cannot accidentally leak ground-truth test outcomes into the judge or aggregation process.

4.3 Baselines

The preliminary experiment reports three quantities:

- **Pass@1 baseline:** success rate of a single unverified candidate.
- **Verifier-selected top-1:** success rate of the single candidate selected by pairwise weak judging and Kemeny aggregation.
- **Oracle pass@4:** an upper baseline that counts an issue as solved if at least one of four candidate attempts solves it.

Oracle pass@4 is not a deployable single-patch selector because it assumes knowledge of which candidate passes the hidden tests. It is included to measure how much useful solution mass exists in the sampled candidate pool. The verifier-selected top-1 result

Table 2: Preliminary result on a 50-instance relatively difficult SWE-bench Verified diagnostic subset. Wilson intervals are included to emphasize small-sample uncertainty.

Method	Solved / 50	Rate	95% Wilson CI
Pass@1 baseline	31	62.0%	[48.2%, 74.1%]
Verifier-selected top-1	33	66.0%	[52.2%, 77.6%]
Oracle pass@4	35	70.0%	[56.2%, 80.9%]

should be compared primarily to pass@1. The gap between verifier-selected top-1 and oracle pass@4 measures remaining selection error, provided the verifier and oracle use the same candidate pool.

4.4 Metrics and diagnostics

The primary metric is the official resolved rate:

$$\text{resolved rate} = \frac{\text{number of resolved instances}}{\text{number of evaluated instances}}. \tag{4}$$

For a verifier-based system, resolved rate alone is insufficient. The following diagnostics should also be reported:

- invalid judge-output rate;
- legal A–J token coverage;
- number of pairwise comparisons per issue;
- prompt-length and truncation statistics;
- candidate-pool size and missing-candidate rate;
- cache/resume consistency;
- token usage for candidate generation and judging;
- wall-clock time and hardware utilization where applicable.

These diagnostics are necessary because a small resolved-rate gain is only meaningful if the verifier is reliable, reproducible, and cost-effective.

5 Preliminary Results

Table 2 summarizes the current 50-instance result. The pass@1 baseline solves 31/50 instances, or 62.0%. The weak-judge selection pipeline with `gpt-oss-120b` solves 33/50 instances, or 66.0%. The oracle pass@4 baseline solves 35/50 instances, or 70.0%.

The verifier improves over pass@1 by 4 percentage points, corresponding to two additional resolved instances on the 50-instance subset. Oracle pass@4 improves over pass@1 by 8 percentage points, corresponding to four additional resolved instances. Under this preliminary protocol, the verifier therefore recovers

$$\frac{33 - 31}{35 - 31} = 0.5 \tag{5}$$

of the observed pass@4 improvement over pass@1. Equivalently, it recovers half of the available four-rollout oracle gap while still trailing the oracle by two instances.

This is a positive but limited signal. The positive signal is that the weak judge is not merely adding noise: it selects a top-1 patch that solves more instances than the unverified baseline. The limitation is that two additional successes are not enough to make a strong statistical claim. The Wilson intervals overlap substantially, and a proper significance test would require paired per-instance outcomes, such as whether the verifier fixed or broke the baseline candidate on each instance. With paired data, McNemar testing or a paired bootstrap over instances would be more appropriate than treating the three rates as independent binomial estimates.

The result also supports the decision to call the subset relatively difficult. Even with four generated attempts and an oracle selector, 15/50 instances remain unsolved. Thus the subset is not merely testing whether the judge can identify obvious wins; it includes cases where candidate generation fails entirely and cases where the judge must distinguish correct patches from plausible incorrect ones. This is the right stress regime for a verifier, but it also means that the result should not be extrapolated to the full 500-instance benchmark.

6 Discussion

6.1 What the result suggests

The main positive finding is that trajectory evidence appears useful. A judge that does not generate patches can still improve final candidate selection by examining the files inspected, commands run, validation attempts, and final diffs. The improvement from 31/50 to 33/50 suggests that the judge identifies at least some cases where the first or default candidate is not the best candidate among the sampled attempts.

The result also suggests that pairwise ranking is a reasonable interface for weak verification. Pairwise comparison can be easier than assigning calibrated absolute correctness scores, especially when candidates solve the same issue and can be compared relative to the same problem statement. This matches broader evidence that pairwise verification can be a strong primitive for test-time scaling [13].

6.2 What the result does not prove

The current experiment does not prove compute efficiency. If the system still generates four complete trajectories for every issue and then adds many judge calls, it may improve selected-patch quality while increasing total cost. The intended contribution is progressive allocation over partial trajectories, but the current numerical result uses `C4` full trajectories only. Therefore, the current result validates candidate selection signal, not early pruning.

The current result also does not prove that `gpt-oss-120b` is a generally calibrated verifier. The judge may overvalue superficial indicators such as running many tests, producing longer explanations, or editing files that look semantically central. The planned ablations should separate genuine root-cause understanding from these surface cues.

6.3 Likely failure modes

The gap between verifier-selected top-1 and oracle pass@4 indicates remaining selection error. Plausible causes include:

- **Evidence omission:** deterministic truncation may remove decisive evidence from one candidate.
- **Prompt framing bias:** one prompt may overemphasize process quality while underweighting patch correctness.
- **Score-token noise:** the A–J mapping may not be sufficiently calibrated across issues.

- **Aggregation loss:** Kemeny aggregation may be robust to cycles but cannot recover evidence that the pairwise judge failed to express.
- **Benchmark ambiguity:** some patches may pass tests without matching developer intent, or fail tests despite being functionally plausible [9, 15].

7 Limitations

The current report intentionally avoids strong claims for five reasons.

First, the result is based on only 50 instances. The observed improvement is two instances, so the result should be treated as directional evidence. Second, the subset is relatively difficult and diagnostic rather than randomly sampled. This is appropriate for stress-testing a verifier, but not for estimating full-benchmark performance. Third, the current experiment uses only `C4` full-trajectory evidence. It does not yet establish that early checkpoint selection can reduce compute. Fourth, the experiment uses one judge model and one main selection configuration. Robustness across judge backends, prompt variants, aggregation methods, and candidate-generation seeds is unknown. Fifth, compute-normalized measurement is incomplete. The final evaluation should report candidate-generation cost, judge-call cost, token usage, wall-clock time, hardware utilization, and pruning efficiency.

There are also benchmark-level limitations. SWE-bench Verified was designed to improve reliability over the original SWE-bench, but recent work argues that benchmark contamination and model memory may affect interpretation [11]. Other work shows that test-passing patches can still diverge behaviorally from developer-intended fixes [15]. OpenAI has also argued that SWE-bench Verified no longer measures frontier coding capabilities because of saturation, contamination, and residual test-quality problems [9]. For this project, SWE-bench Verified is still useful as a controlled diagnostic benchmark for trajectory selection, but it should not be presented as a complete measure of real-world software engineering reliability.

8 Planned Large-Scale Experiments

8.1 Scale-up protocol

The next stage is to run the larger evaluation with consistent candidate-pool definitions. A recommended sequence is:

(1) one-instance dry run with prompt emission and no judge API calls;
(2) one-instance live smoke test with full pairwise comparison;
(3) prompt-length audit over all selected instances;
(4) small live batch, for example five instances, with conservative worker count;
(5) full selected-set run after invalid-output rate, token coverage, and cache behavior are acceptable.

All judge-call artifacts should be cacheable. Ranking outputs should be written before resolved labels are joined. The artifact store should include the issue id, candidate ids, checkpoint id, prompt variant, truncation diagnostics, judge backend, score-token distribution, pairwise margin, aggregate ranking, selected candidate, and final resolved label.

8.2 Local SWE-Star 14B variant

A local-GPU version will evaluate whether a smaller open model can serve as the weak judge. The planned backend is SWE-Star 14B, part of the SWE-Star family of agentic coding models [7]. The experimental design should remain unchanged: only the judge inference backend and score-probability extraction method should differ. The local backend must return a single A–J score token and a normalized distribution over legal A–J alternatives. If an OpenAI-compatible endpoint cannot provide reliable top log probabilities, the fallback is a native next-token-logits backend that computes legal-token probabilities directly from the model output head.

The local run should be reported as a separate variant from the `gpt-oss-120b` result. Its artifacts must record served model name, quantization mode, context length, score-probability source, truncation policy, and cache-key fields. These details matter because local serving choices can change the score distribution even when the prompt text is unchanged.

8.3 Ablations

The final report should include ablations that isolate which components matter:

- **Prompt format:** root-cause trace, process monitorability, and their combination.
- **Checkpoints:** `C1`, `C2`, `C3`, and `C4` when partial-checkpoint runs are available.
- **Aggregation:** weighted Kemeny versus majority vote, Borda-style averaging, and average pairwise margin.
- **Probability use:** full score-token probability margins versus hard visible-token outcomes.
- **Context truncation:** no truncation where possible versus deterministic pair-level truncation.
- **Judge backend:** `gpt-oss-120b` versus local SWE-Star 14B and any additional open judge models.
- **Candidate pool size:** pass@4-compatible four-candidate ranking versus eight-candidate ranking with a matching pass@8 oracle.

8.4 Expected final claims

The final claims should be framed in terms of measured tradeoffs, not only raw resolved rate. The strongest supported claim would require showing that weak-to-strong temporal verification selects a single candidate that improves over pass@1, approaches pass@k, and does so with judge overhead small enough to justify the selection step. The current preliminary result supports only the first part: selected top-1 improves from 62.0% to 66.0% on a 50-instance relatively difficult diagnostic subset, while remaining below the 70.0% oracle pass@4 baseline.

9 Conclusion

This report studies weak-to-strong verification for SWE agents: using a judge model to compare candidate trajectories and select one final patch. The current 50-instance result is encouraging but preliminary. A `gpt-oss-120b` judge improves selected top-1 performance from 31/50 to 33/50 resolved instances and recovers half of the observed oracle pass@4 improvement. However, the experiment is small, uses a relatively difficult diagnostic subset, evaluates

only full trajectories, and lacks complete compute accounting. The next step is a larger, artifact-complete evaluation that tests partial checkpoints, reports cost-normalized performance, and compares judge backends and aggregation choices.

References

[1] Islem Bouzenia and Michael Pradel. 2025. Understanding Software Engineering Agents: A Study of Thought-Action-Result Trajectories. https://arxiv.org/abs/2506.18824 arXiv preprint arXiv:2506.18824.

[2] Neil Chowdhury, James Aung, Chan Jun Shern, Oliver Jaffe, Dane Sherburn, Giulio Starace, Evan Mays, Rachel Dias, Marwan Aljubeh, Mia Glaese, Carlos E. Jimenez, John Yang, Leyton Ho, Tejal Patwardhan, Kevin Liu, and Aleksander Madry. 2024. Introducing SWE-bench Verified. OpenAI blog post. https://openai.com/index/introducing-swe-bench-verified/

[3] Yifeng Ding and Lingming Zhang. 2026. SWE-Replay: Efficient Test-Time Scaling for Software Engineering Agents. https://arxiv.org/abs/2601.22129 arXiv preprint arXiv:2601.22129.

[4] Naman Jain, Jaskirat Singh, Manish Shetty, Liang Zheng, Koushik Sen, and Ion Stoica. 2025. R2E-Gym: Procedural Environments and Hybrid Verifiers for Scaling Open-Weights SWE Agents. https://arxiv.org/abs/2504.07164 arXiv preprint arXiv:2504.07164.

[5] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-World GitHub Issues?. In *The Twelfth International Conference on Learning Representations*. https://openreview.net/forum?id=VTF8yNQM66

[6] John G. Kemeny. 1959. Mathematics Without Numbers. *Daedalus* 88, 4 (1959), 577–591.

[7] LogicStar AI. 2026. SWE-Star: Best-in-Class Agentic Coding Models. Technical blog post. https://logicstar.ai/blog/swe-star

[8] OpenAI. 2025. gpt-oss-120b and gpt-oss-20b Model Card. https://arxiv.org/abs/2508.10925 arXiv preprint arXiv:2508.10925.

[9] OpenAI. 2026. Why SWE-bench Verified no longer measures frontier coding capabilities. OpenAI blog post. https://openai.com/index/why-we-no-longer-evaluate-swe-bench-verified/

[10] 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 Machine Learning Research*, Vol. 267. 47717–47737. https://arxiv.org/abs/2412.21139

[11] Thanosan Prathifkumar, Noble Saji Mathews, and Meiyappan Nagappan. 2025. Does SWE-Bench-Verified Test Agent Ability or Model Memory? https://arxiv.org/abs/2512.10218 arXiv preprint arXiv:2512.10218.

[12] Mohit Raghavendra, Anisha Gunjal, Bing Liu, and Yunzhong He. 2026. Agentic Rubrics as Contextual Verifiers for SWE Agents. https://arxiv.org/abs/2601.04171 arXiv preprint arXiv:2601.04171.

[13] Harman Singh, Xiuyu Li, Kusha Sareen, Monishwaran Maheswaran, Sijun Tan, Xiaoxia Wu, Junxiong Wang, Alpay Ariyak, Qingyang Wu, Samir Khaki, Rishabh Tiwari, Long Lian, Yucheng Lu, Boyi Li, Alane Suhr, Ben Athiwaratkun, and Kurt Keutzer. 2026. V_1: Unifying Generation and Self-Verification for Parallel Reasoners. https://arxiv.org/abs/2603.04304 arXiv preprint arXiv:2603.04304.

[14] SWE-bench Team. 2026. SWE-bench Verified. Benchmark documentation. https://www.swebench.com/verified.html

[15] You Wang, Michael Pradel, and Zhongxin Liu. 2025. Are "Solved Issues" in SWE-bench Really Solved Correctly? An Empirical Study. https://arxiv.org/abs/2503.15223 arXiv preprint arXiv:2503.15223.

[16] 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 (2025), 801–824. doi:10.1145/3715754

[17] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. In *Advances in Neural Information Processing Systems*. https://arxiv.org/abs/2405.15793

[18] H. Peyton Young and Arthur Levenglick. 1978. A Consistent Extension of Condorcet's Election Principle. *SIAM J. Appl. Math.* 35, 2 (1978), 285–300. doi:10.1137/0135023

[19] Guangtao Zeng, Maohao Shen, Delin Chen, Zhenting Qi, Subhro Das, Dan Gutfreund, David Cox, Gregory Wornell, Wei Lu, Zhang-Wei Hong, and Chuang Gan. 2025. Satori-SWE: Evolutionary Test-Time Scaling for Sample-Efficient Software Engineering. https://arxiv.org/abs/2505.23604 arXiv preprint arXiv:2505.23604.

[20] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. AutoCodeRover: Autonomous Program Improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. doi:10.1145/3650212.3680384