SWE-XPLORER: SCALING CODING AGENTS THROUGH ITERATIVE TREE SEARCH

Anonymous authors
Paper under double-blind review

ABSTRACT

Test-time scaling is a promising way to improve LLM-based coding agents. However, long-horizon software repair produces extended trajectories whose partial progress must be preserved, compared, and reused; simply adding steps or branches can yield diminishing returns. We present **SWE-X**plorer, an iterative tree-search framework built around a repeatable explore–reconcile–prune cycle. Within each iteration, path-level best-first selection extends promising trajectories; at the iteration boundary, compatible paths exchange evidence and the frontier is compressed to seed the next iteration. This bounded design lets the search use larger test-time budgets while repeatedly consolidating its most promising progress; end patch selection then compares candidates accumulated across the full search. On SWE-bench Verified (gpt-5-mini), increasing the test-time budget from 30 to 50 improves SWE-Xplorer from 54.6% to 61.4%, while the prior tree-search framework SWE-Search improves from 54.2% to 54.8%. SWE-Xplorer also has lower average cost and runtime than SWE-Search; at budget 50, the reductions are 64.9% and 32.2%, respectively. It achieves a higher resolved rate than strong single-trajectory agents—Claude Code and OpenHands—while costing less than either. Relative to its single-trajectory backbone (mini-SWE-agent), SWE-Xplorer improves resolved rate with all three evaluated models on both SWE-bench Verified and SWE-bench Multilingual. Together, these results show that iteration-based search can turn additional test-time compute into continued gains for long-horizon coding agents.

1 INTRODUCTION

Effective software debugging is an inherently exploratory process. It involves reading relevant parts of the codebase, forming and discarding hypotheses, and sometimes discovering that two lines of investigation explored independently contain complementary clues that, taken together, point directly to the fix. This kind of flexible, multi-threaded investigation is a hallmark of effective debugging. Single-trajectory coding agents instead follow one sequence of actions (Yang et al., 2024; Wang et al., 2025), making it difficult to compare alternative lines of investigation or return to an earlier decision point.

Test-time compute scaling offers a natural remedy: prior work explores multiple candidate solutions through parallel rollouts (Ehrlich et al., 2025), sequential refinement (Kim et al., 2026), and tree search (Zhou et al., 2024; Antoniades et al., 2025), establishing that exploring alternative repair strategies can outperform a single trajectory. We study how to keep tree-search performance improving as the test-time budget grows.

Yet additional branches and steps do not by themselves guarantee useful progress – naive tree search can waste a larger budget as easily as it can exploit one. As the budget grows, the search must rank more paths, preserve more evidence, and compare more candidate patches, and each of these requirements exposes a distinct failure mode:

C1. One bad step can strand a good strategy. Action-level best-first methods prioritize individual frontier nodes or actions by their scores (Dihan et al., 2025; Koh et al., 2025). A locally weak action can therefore leave an otherwise promising trajectory unexpanded even with more budget. In debugging, this is particularly harmful: missing one file inspection can prevent an otherwise correct repair strategy from ever being reached.

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.

C2. Unconstrained search spreads the budget too thinly. Without pruning, new branches can cause the frontier to keep growing. The search can then spend its budget maintaining many shallow, partially explored paths rather than following a promising strategy to completion.

C3. Pruning discards accumulated investigative evidence. Pruning keeps the frontier small, but choosing survivors only by score can lose useful context. Two trajectories that reach the same codebase state through different investigations may hold complementary clues about the bug; keeping just one discards the other's evidence.

C4. More exploration makes patch selection consequential. A larger budget can uncover plausible patches on different branches. Submitting one early can discard candidates found later, while exhausting the budget without an explicit submission can leave promising edited states unused. The final decision should compare patches found across the search, not just the currently active path.

To address these challenges, we introduce **SWE-X**plorer, a tree-search framework for autonomous issue resolution built around three techniques. First, SWE-Xplorer selects which node to expand by *path-level best-first search*: expansion priority is scored over the accumulated trajectory rather than its most recent action alone, so a single locally weak step no longer strands an otherwise strong path (**C1**) – the path can stay competitive until a later action recovers. Second, rather than letting this best-first process run unconstrained, SWE-Xplorer divides the total budget into a sequence of iterations, each exploring the frontier for its share of the budget before compressing it to the top-k trajectories that seed the next iteration; this bounds the frontier at every iteration boundary instead of letting it grow across the whole budget (**C2**). Before that compression discards the rest of the frontier, SWE-Xplorer applies *cross-path reconciliation*: trajectories that reach the same codebase state through different investigations are merged into a single joint representative informed by both histories, so two independently accumulated lines of evidence are combined rather than one being thrown away because its score fell outside the top k (**C3**).

Finally, SWE-Xplorer decouples proposing a fix from committing to it, addressing **C4** through three mechanisms: *deferred termination* records a proposed patch as a candidate without ending the search, so exploration can continue toward better alternatives instead of stopping early; *automated solution augmentation* turns any promising edited codebase state into a candidate even when the agent never explicitly submits it, so a good fix is never lost to an exhausted budget; and *recursive tournament voting* then selects among the resulting pool by comparing candidates head-to-head, rather than trusting a single noisy reward score or whichever path happened to run last.

We evaluate SWE-Xplorer on the full 500-task SWE-bench Verified and 300-task SWE-bench Multilingual benchmarks with three backend model families. It improves resolved rate over its single-trajectory mini-SWE-agent backbone by 3.0–22.3 percentage points across all six model–benchmark settings, reaching 65.4% on Verified and 49.33% on Multilingual with DeepSeek-V4-Flash. In the scaling study on Verified with gpt-5-mini, increasing the test-time budget from 30 to 50 raises SWE-Xplorer from 54.6% to 61.4% (+6.8 points), while mini-SWE-agent and SWE-Search gain only 1.6 and 0.6 points over the corresponding budgets. With gpt-5-mini, a budget of 50, and branching factor 2, SWE-Xplorer resolves 61.4% versus SWE-Search's 54.8%, at \$0.14 versus \$0.39 average cost and 18.3 versus 27.0 minutes per task. It also resolves more tasks than OpenHands and Claude Code at lower average cost: \$0.14 per task versus \$0.21 and \$0.67, respectively.

2 Related Work

Coding agents and test-time scaling. SWE-bench (Jimenez et al., 2024) has motivated systems that produce one patch through a single pipeline or interaction trajectory, including SWE-agent (Yang et al., 2024), OpenHands (Wang et al., 2025), and Agentless (Xia et al., 2024). A complementary line of work scales test-time computation by sampling independent trajectories, refining earlier solutions, or branching retrospectively from completed runs (Ehrlich et al., 2025; Kim et al., 2026; Ding & Zhang, 2026). These methods increase solution diversity, but independent runs repeatedly rediscover the same repository context, while retrospective branching can exploit an alternative only after the original trajectory has unfolded. SWE-Xplorer instead maintains an online tree: alternatives share their common prefix and compete for subsequent expansions as evidence arrives.

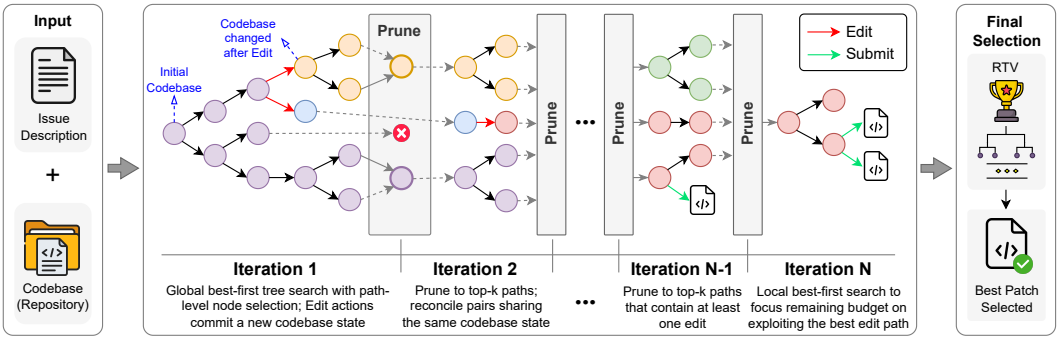


Figure 1: Overview of **SWE-X**plorer. Given an issue description and a codebase, the system divides its search budget across a sequence of tree-search iterations to produce a patch. At each step, the node with the highest path score is selected for expansion. Edit actions (red edges) produce a new codebase state; non-edit actions leave the codebase unchanged. For example, in Iteration 1 the search expands nodes from the root by path score alone; when two expansions reach the same codebase state (highlighted), they are reconciled into a single merged node before the frontier is pruned to its top-k survivors, which seed Iteration 2. In the final iterations, the search narrows to paths that have made at least one edit and focuses on finalizing the fixes found so far rather than exploring new directions. The resulting patch candidates are passed to Recursive Tournament Voting (RTV) for final selection.

Search for language and coding agents. Tree of Thoughts (Yao et al., 2023a) and LATS (Zhou et al., 2024) established tree search as a way to deliberate over language-model decisions. In software engineering, SWE-Search (Antoniades et al., 2025) applies Monte Carlo tree search, while DARS (Aggarwal et al., 2025) revisits consequential action points through adaptive resampling. SWE-Xplorer differs in treating a recent root-to-node path, rather than an isolated node, as the unit of selection. It also periodically prunes the frontier, reconciles evidence from compatible live branches, and postpones the choice of a final patch until exploration stops. Together, these choices target a central difficulty in coding search: preserving coherent lines of investigation while still allowing information discovered on one branch to benefit another. Appendix A provides a detailed conceptual comparison.

3 Method

3.1 Problem Formulation

We formulate issue resolution as search over agent–environment interaction trajectories. A task specifies an issue description $\mathcal{I}$ and an initial codebase $\mathcal{C}_0$; the goal is to produce a patch that resolves the issue when applied to $\mathcal{C}_0$. The environment is a sandboxed checkout of the codebase, which the agent accesses through an action interface. The agent selects actions $a_1, a_2, \ldots$ from an action space $\mathcal{A}$ that includes file reads, searches, code edits, test runs, and patch submission. Executing action a_t produces an observation o_t. An interaction trajectory is therefore an alternating sequence

$$h_t = (a_1, o_1, a_2, o_2, \ldots, a_t, o_t).$$

Actions are typed: $\tau(a) \in \{\texttt{read}, \texttt{search}, \texttt{edit}, \texttt{test}, \texttt{submit}\}$. Each action executes in a fresh shell with no shared shell-session state; for example, a directory change in one action does not carry over to the next. Under our action interface, only `edit` actions can change the codebase state; all other action types leave it unchanged.

3.2 Path-Level Best-First Search

SWE-Xplorer represents agent–environment trajectories as a rooted tree $\mathcal{T}$. The root has an empty observation. Each subsequent node n records the action taken from its parent, the resulting observation (such as file contents or test output), a scalar reward $v(n)$, and metadata including the action type, accessed files, and a reference to the resulting codebase state. An action-type-specific LLM

evaluator scores each action–observation pair, with execution feedback adjusting the reward $v(n)$ (Appendix C.3). Children of non-edit actions inherit their parent's codebase state; an edit child records the resulting state. The root-to-n path records the sequence of actions and observations leading to n; this is the interaction history available to the agent when continuing from that node.

The frontier $\mathcal{F}$ contains nodes available for expansion and starts with the root. To rank frontier nodes, SWE-Xplorer uses a *path score* $V_{\text{path}}(n)$, a discounted average of the most recent W node rewards along the root-to-n path. Limiting the window prevents early rewards from sustaining a path indefinitely, while discounting older rewards gives the latest actions more weight:

$$V_{\text{path}}(n) = \frac{\sum_{i=0}^{\min(W-1,\text{depth}(n))} \gamma^i v(p_i(n))}{\sum_{i=0}^{\min(W-1,\text{depth}(n))} \gamma^i}.$$

Here $p_i(n)$ is the i-th ancestor of n, with $p_0(n) = n$, and γ is the discount factor. At each step, SWE-Xplorer selects the frontier node n^* with the highest path score.

Before expanding n^*, the framework restores its recorded codebase state via Git if necessary (Appendix C.5). It then generates candidate actions (Appendix C.2) and executes each one, reusing a cached observation when the same action was previously executed from the same codebase state. Each resulting child node is scored and, unless produced by a `Submit` action, added to $\mathcal{F}$. A `Submit` action creates a *terminating node*: it ends that trajectory, so the node is retained as a patch candidate rather than expanded. Other frontier nodes remain available for expansion.

Insight (The dead-end problem)**.** *Action-level best-first selection can strand a useful investigation after one low-scoring step: its frontier node may never rank high enough to be expanded again. The path score retains credit for recent productive steps, allowing that investigation to continue and recover from a locally weak action.*

3.3 Iteration-Based Search

Unconstrained best-first search spreads the expansion budget across an ever-growing frontier: each expansion adds b new candidate leaves, so after s expansions the frontier can contain up to $O(bs)$ open trajectories. In practice this means the search spends its budget maintaining a large number of shallow, partially explored paths rather than following any single strategy to completion. SWE-Xplorer addresses this through a repeatable iteration design: the total expansion budget B is divided across N iterations of roughly B/N expansions each. Within each iteration, the search uses path-level best-first exploration. This structure lets larger budgets be organized into bounded rounds of exploration and consolidation rather than spent on an ever-growing frontier. At the end of each iteration, a pruning step narrows the frontier to the top-k nodes to seed the next iteration, discarding the rest. This structure alternates between broad parallel exploration within an iteration and focused exploitation across iterations as the frontier narrows. Figure 1 illustrates this cycle of expansion and pruning. Appendix D gives the complete search procedure.

The closest related pruning strategy is beam search, which also maintains a fixed-width frontier of k candidates, retaining only the top-k nodes by score after every expansion. Both methods bound the frontier size; they differ in when and how pruning occurs. Beam search prunes *after every expansion*, which forces early commitment and can permanently eliminate a promising path after a single weak action. SWE-Xplorer instead prunes only at *iteration boundaries*. Trajectories compete freely within an iteration, so strong follow-up actions can compensate for a locally weak step before any path is discarded. Pruning is informed by the full path score accumulated over B/N expansions rather than a single-action score. Beam search also offers no mechanism to combine information across trajectories, whereas SWE-Xplorer enriches the candidate pool before each pruning step through cross-path reconciliation (Section 3.4).

Convergence-forcing. As the expansion budget runs down, continuing to follow paths that have not edited the codebase can leave the search with too few candidate patches. SWE-Xplorer therefore prioritizes *edit paths*—paths containing at least one `edit` action—during the final two iterations. In the pruning step before iteration $N-1$, edit paths rank first among the k survivors. This gives partial fixes another round of actions. The agent can inspect their effects, run tests, and revise the code during that round. The next pruning step can then choose among more developed repair attempts. Before iteration N, the search retains only the best edit path instead of k paths. The final iteration then shifts from exploring multiple approaches to completing the selected repair through *local best-first search*:

after expanding a node, it selects the best non-terminating child for the next expansion. This keeps the remaining budget on that line of work rather than moving between branches.

When few expansions remain in the budget, the agent receives an explicit prompt instructing it to prioritize completing any pending code fix rather than continuing to explore. This makes the agent's proposed actions responsive to the remaining budget, encouraging it to finish a repair in the final expansions.

3.4 Cross-Path Reconciliation

One naive approach to *pruning* at each iteration boundary is to simply pick the top-k nodes by path score and discard the rest, losing all investigative findings accumulated along the pruned trajectories. To maximize information gain from the full frontier before it is narrowed, SWE-Xplorer applies cross-path reconciliation. Frontier nodes are grouped into reconcilable pairs, and for each pair the agent creates a *merged node* informed by the observations along both trajectories. It replaces the pair in the frontier but has only one structural parent, so the search structure remains a tree. Appendix F visualizes a complete search tree from an actual run, including two such reconciliation events.

Two nodes n_A and n_B are *reconcilable* if they share the same codebase state—meaning all files in the codebase have identical content at both nodes. Among all eligible pairs, SWE-Xplorer greedily forms disjoint pairs by *structural proximity* in the tree, measured as the sum of each node's distance to their lowest common ancestor, so that trajectories that diverged most recently are merged together. The agent receives the shared history up to the common ancestor followed by the distinct action-observation sequences for both n_A and n_B, and is prompted:

> *"From this point, two alternative investigation sequences were explored. Your task is to reason across both and decide the best next step."*

The synthesized action is executed *once* from the codebase state shared by n_A and n_B, producing a single action–observation pair and resulting codebase state for the merged node m. We do not re-execute the action from each trajectory. Instead, we evaluate this same realized outcome twice: once with the history ending at n_A in the reward-model prompt and once with the history ending at n_B. Thus, $r(m \mid n_A)$ and $r(m \mid n_B)$ are two trajectory-conditioned assessments of one execution, and may differ because the histories provide different evidence about the action's relevance and information gain. The merged value is their weighted combination:

$$\tilde{v}(m) \;=\; w_A \cdot r(m \mid n_A) \;+\; w_B \cdot r(m \mid n_B),$$

where $w_X = V_{\text{path}}(n_X)/(V_{\text{path}}(n_A) + V_{\text{path}}(n_B))$. The merged value serves as m's reward in subsequent path scoring. The parent of m is whichever of n_A and n_B gives the higher trajectory-conditioned reward, $r(m \mid n_A)$ or $r(m \mid n_B)$.

3.5 Deferred Termination and Solution Selection

A common failure mode of coding agents is *termination timing*: the agent either submits too early or exhausts its budget without submitting at all. SWE-Xplorer decouples termination *generation* from termination *execution* throughout the search. Whenever the agent emits a `Submit` action, the corresponding terminating node is recorded as a candidate rather than ending the episode, and exploration continues within the current iteration. To guard against submitting too early, exploration cannot stop until the tree has accumulated enough terminating candidates (a threshold, Appendix C.1). This threshold is checked only at iteration boundaries. A stop decision therefore never cuts off exploration already under way in the current iteration. If the threshold is never reached, exploration continues until the full expansion budget is exhausted.

Once exploration ends, SWE-Xplorer applies *automated solution augmentation*: every unique codebase state produced by an edit action, except the initial codebase, is a potential solution. For any such state that does not already have an explicit `Submit` candidate, one is programmatically generated. This ensures that adequate fixes the agent reached but never explicitly submitted are not lost due to budget exhaustion.

Given the resulting pool of candidates, selecting by raw reward score alone is unreliable due to reward model noise and known biases in LLM-based evaluation (Zheng et al., 2023). Instead, SWE-Xplorer

runs *recursive tournament voting* (Kim et al., 2026). Candidates are pre-sorted using a composite score that considers their reward, discovery order, provenance (agent-generated or automatically augmented), and path score. The top-ranked candidates enter a bracket, and an LLM judge compares pairs head-to-head based on a structured trajectory summary (root cause, files changed, fix strategy, regression risk). Winners advance until a single candidate remains.

4 EXPERIMENTS

4.1 EXPERIMENTAL SETUP

Benchmarks. We evaluate SWE-Xplorer on SWE-bench Verified (Jimenez et al., 2024) and SWE-bench Multilingual (Yang et al., 2026). SWE-bench Verified contains 500 real GitHub issues drawn from popular Python repositories, each paired with a test suite that programmatically verifies whether a submitted patch resolves the issue. SWE-bench Multilingual contains 300 issues spanning multiple programming languages; we use it to test whether gains transfer beyond Python. The benchmark measures the *resolved rate*: the fraction of issues for which the submitted patch causes all associated tests to pass.

Implementation. SWE-Xplorer uses mini-SWE-agent (Yang et al., 2024) as its action agent. Within each experimental configuration, the same language model is used for action generation, cross-path reconciliation, and tournament voting. Unless otherwise stated, this model is gpt-5-mini. We also evaluate Qwen-2.5-7B (Team, 2024) and DeepSeek-V4-Flash (**?**) in these roles. For reward scoring, we use Qwen-2.5-7B as a lightweight evaluator, keeping the evaluator fixed across model configurations while limiting scoring cost. We use *test-time budget* as the common term throughout: it counts interaction steps for a single-trajectory agent and tree expansions for a tree-search agent. Unless otherwise stated, SWE-Xplorer uses a branching factor $b = 2$, frontier size $k = 4$, $N = 5$ iterations, and an expansion budget of $B = 50$ per task (10 expansions per iteration). Hyperparameters are detailed in Appendix C. Prompts used for the coding agent and test reproduction agent are provided in Appendix H.

Baselines. We compare SWE-Xplorer against mini-SWE-agent running as a single-trajectory agent with the same model and the same nominal test-time budget of 50. This comparison keeps the underlying agent model and numeric budget fixed; because each tree expansion samples up to b actions and invokes a reward model, it is not a token- or API-call-matched comparison. We also evaluate SWE-Search (Antoniades et al., 2025) as a tree-search baseline on SWE-bench Verified with gpt-5-mini, an expansion budget of 50, and branching factor 2, matching these settings in SWE-Xplorer. We additionally include OpenHands (Wang et al., 2025), OpenCode (Anomaly, 2026), and Claude Code (Anthropic, 2026) in the cost–performance comparison.

4.2 MAIN RESULTS

Table 1's Overall rows report resolved rates on SWE-bench Verified and SWE-bench Multilingual. SWE-Xplorer outperforms the single-trajectory mini-SWE-agent baseline for each evaluated model. With gpt-5-mini, SWE-Xplorer achieves **61.4%** against 53.2% for the baseline (+8.2 points). With Qwen-2.5-7B, SWE-Xplorer achieves **4.2%** versus 1.0% (+3.2 points); because both values are low, we do not draw a strong conclusion from this configuration alone. With DeepSeek-V4-Flash, SWE-Xplorer achieves **65.4%** against 54.0% (+11.4 points), demonstrating that the gains transfer across model families. Across the three Verified difficulty tiers with at least 42 tasks, gpt-5-mini gains 6.2, 10.3, and 4.8 points, respectively. DeepSeek-V4-Flash also gains in each of these tiers; the >4-hour tier has only three tasks.

The gains also extend across languages. On SWE-bench Multilingual, SWE-Xplorer improves gpt-5-mini from 26.0% to 43.33% (+17.33 points) and DeepSeek-V4-Flash from 27.0% to 49.33% (+22.33 points). Both gains are larger than their corresponding gains on SWE-bench Verified, indicating that the search procedure remains useful when repository languages and toolchains vary. The Qwen result also rises from 0.0% to 3.00%. In Table 1's per-language rows, gpt-5-mini improves in all nine Multilingual languages (+2.3 to +37.2 points); DeepSeek-V4-Flash improves in eight, with Ruby the exception (45.5% vs. 50.0%, $n = 44$). Some subgroups are small, so the individual differences are descriptive rather than conclusive.

Table 1: Resolved rate (%, mini-SWE-agent / SWE-Xplorer) across all three backbones, broken down by SWE-bench Verified's official difficulty tier (Python only) and by each of SWE-bench Multilingual's nine implementation languages.

	Language	Difficulty	n	Resolved rate (%): mini-SWE-agent / SWE-Xplorer gpt-5-mini	Qwen-2.5-7B	DeepSeek-V4-Flash
Verified	Python	<15 min fix	194	71.1 / **77.3**	2.1 / **9.8**	74.7 / **81.4**
		15 min–1 hour	261	46.4 / **56.7**	0.4 / **0.8**	46.0 / **61.7**
		1–4 hours	42	16.7 / **21.4**	0.0 / 0.0	11.9 / **16.7**
		>4 hours	3	0.0 / 0.0	0.0 / 0.0	0.0 / **33.3**
		Overall	500	53.2 / **61.4**	1.0 / **4.2**	54.0 / **65.4**
Multilingual	Ruby		44	36.4 / **43.2**	0.0 / **2.3**	**50.0** / 45.5
	Java		43	20.9 / **51.2**	0.0 / **2.3**	18.6 / **41.9**
	PHP		43	16.3 / **53.5**	0.0 / 0.0	16.3 / **58.1**
	Rust		43	44.2 / **46.5**	0.0 / **2.3**	39.5 / **58.1**
	Go		42	14.3 / **26.2**	0.0 / 0.0	14.3 / **35.7**
	JavaScript		33	21.2 / **48.5**	0.0 / 0.0	30.3 / **54.5**
	C		30	33.3 / **43.3**	0.0 / 0.0	26.7 / **56.7**
	C++		12	16.7 / **25.0**	0.0 / **50.0**	16.7 / **50.0**
	TypeScript		10	20.0 / **30.0**	0.0 / 0.0	10.0 / **40.0**
	Overall		300	26.0 / **43.3**	0.0 / **3.0**	27.0 / **49.3**

These results indicate that iterative tree search with path-level selection and cross-path reconciliation can extract more value from a fixed nominal test-time budget than a single-trajectory agent, at the cost of additional branching and scoring calls per expansion.

4.3 Scaling Analysis

The key test of SWE-Xplorer's iteration-based search is whether it continues to turn additional expansions into resolved issues as the test-time budget grows. We compare resolved rates from budgets 10 to 50 on SWE-bench Verified with gpt-5-mini against mini-SWE-agent and SWE-Search (Figure 2).

mini-SWE-agent improves sharply from 19.6% at a budget of 10 to 51.6% at 30, but then reaches only 52.6% at 40 and 53.2% at 50. SWE-Search follows a similar pattern: it rises from 43.0% at 10 to 54.2% at 30, then plateaus at 54.8% for both 40 and 50. Thus, from a budget of 30 to 50, mini-SWE-agent and SWE-Search gain only 1.6 and 0.6 percentage points, respectively.

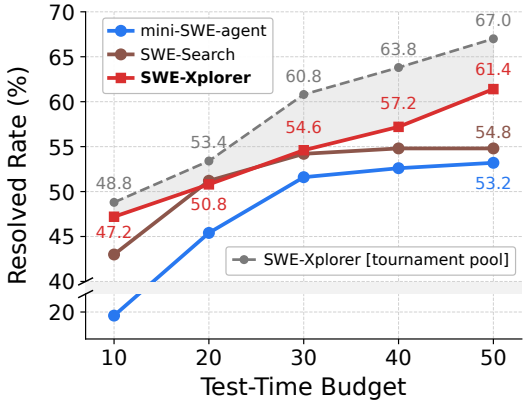


Figure 2: Resolved rate by test-time budget on SWE-bench Verified (gpt-5-mini). The dashed line is the candidate-pool ceiling: the rate if a resolving patch in the tournament pool were always chosen.

SWE-Xplorer resolves 47.2% at a budget of 10 and is level with SWE-Search through 20 (50.8% vs 51.2%). It separates from 30 onward, improving 54.6% to 57.2% to 61.4%: a 6.8-point gain over the range where the two baselines gain 1.6 and 0.6, widening its lead over mini-SWE-agent from 3.0 to 8.2 points. This continued improvement is consistent with later tree expansions pursuing alternative solution paths, while each iteration carries the strongest trajectories and reconciled evidence into the next.

Is the search re-ranking a fixed pool, or finding new solutions? To answer this, we compute the candidate-pool ceiling shown in Figure 2. At each budget, a task counts as solvable if any candidate entering the first tournament round resolves it. The ceiling rises from 48.8% at budget 10 to 67.0% at 50, including a rise from 60.8% to 67.0% between budgets 30 and 50. The continued increase

shows that additional exploration brings resolving patches into the candidate pool as the budget grows. Figure 8 in Appendix G makes the same point on a single issue, showing one instance's complete search tree at all five budgets: it fails at 10 through 40 and resolves only at 50, as each added iteration buys further distinct edit attempts to compare.

4.4 Cost and Runtime Analysis

Cost–accuracy tradeoff. Figure 3 places SWE-Xplorer at 61.4% resolved rate and \$0.14 per task. This is a fivefold cost increase over its mini-SWE-agent backbone (\$0.028), but the additional \$0.112 raises performance from 53.2% to 61.4% (+8.2 points). Among the stronger single-trajectory agents, SWE-Xplorer achieves a higher resolved rate than OpenHands (\$0.21) and Claude Code (\$0.67) while costing less than either; Claude Code reaches 57.0% at approximately 4.8× the cost. mini-SWE-agent and OpenCode (52.0% at \$0.033) remain cheaper, lower-performance alternatives. Thus, the search overhead yields a favorable tradeoff relative to agents with comparable performance. Table 2 tracks all three quantities as the budget grows. Cost and runtime scale linearly in the budget, at roughly \$0.0029 and 0.37 minutes per expansion ($R^2 = 0.97$ and 0.98 for a fit through the origin), rather than super-linearly: every stage except expansion itself is bounded by a hyperparameter rather than by the size of the tree (Table 5). At the $N - 1 = 4$ boundaries between iterations, reconciliation runs and the frontier is pruned to at most $k = 4$ nodes; it can grow between boundaries. The tournament compares at most $2k = 8$ candidates however many the search produced. The 4.9× rise in cost and runtime from a budget of 10 to 50 therefore buys 14.2 points of resolved rate at a stable marginal price.

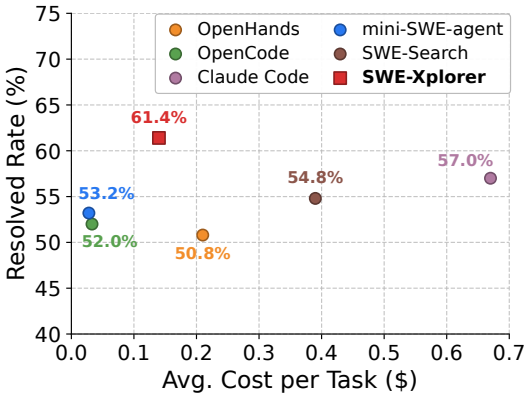


Figure 3: Resolved rate vs average cost per task on SWE-bench Verified (gpt-5-mini, budget 50). Both tree-search methods use branching factor 2.

Runtime tradeoff. Against SWE-Search with the same model, expansion limit, and branching factor, SWE-Xplorer improves resolved rate from 54.8% to 61.4% while reducing average cost from \$0.390 to \$0.137 (64.9%) and runtime from 27.03 to 18.32 minutes (32.2%). The shared limit does not mean equal realized search effort: SWE-Search records 40.9 expansions per task, versus 30.1 for SWE-Xplorer (Table 2). SWE-Xplorer builds its search on the simple mini-SWE-agent action agent. It checks whether enough terminating candidates have accumulated at iteration boundaries, allowing the search to stop before exhausting its expansion budget (Section 3.5). SWE-Search uses a custom layer to track and reverse file changes; SWE-Xplorer uses lightweight version control for state restoration instead (Appendix C.5). It also reuses cached observations for repeated (state, action) pairs (Appendix E.3). The boundary-based stopping rule helps explain the lower average expansion count; the simple action agent, lightweight state restoration, and caching may also limit overhead. These results do not measure the cost or runtime savings from each design choice separately.

Table 2: Resolved rate, average cost, and average runtime per task as the test-time budget grows on SWE-bench Verified (gpt-5-mini). Budget is the expansion limit; SWE-Search is shown at 50. Avg. exp. is the mean number of recorded expansions over all 500 tasks.

Agent	Budget	Avg. exp.	Resolved (%)	Cost (\$/task)	Time (min/task)
SWE-S	50	40.9	54.8	0.390	27.03
SWE-X	10	9.0	47.2	0.028	3.72
	20	14.3	50.8	0.046	5.96
	30	24.0	54.6	0.095	12.36
	40	28.0	57.2	0.117	15.28
	50	**30.1**	**61.4**	**0.137**	**18.32**

4.5 Ablation Study

Table 3 reports one-at-a-time ablations on the full SWE-bench Verified benchmark using gpt-5-mini. Each row disables one mechanism: *w/o cross-path reconciliation* takes the top-k frontier nodes directly instead of merging same-state branches; *w/o deferred termination* applies the stopping check

as soon as the threshold is met instead of deferring it to the next iteration boundary;*w/o recursive tournament voting* submits the candidate ranked highest by reward alone; *w/o path-level selection* ranks the frontier by each node's own reward rather than its path score $V_{\text{path}}(n)$. Path-level selection has the largest measured effect (-6.2 points), while reconciliation and deferred termination each account for 3.2 points. The remaining components also improve the full configuration, with measured differences between 1.2 and 3.0 points. Because each variant removes only one component, these values are marginal effects within the full system and should not be interpreted as additive contributions. The larger drop without path-level selection suggests that ranking complete trajectories by path score is more important than choosing nodes from their immediate rewards alone. The drops from reconciliation and deferred termination further indicate that both information sharing across branches and delaying commitment contribute beyond branching itself.

Table 3: Ablation study on SWE-bench Verified with gpt-5-mini. Δ is relative to SWE-Xplorer (full).

Configuration	**SR (%)**	Δ
mini-SWE-agent	53.2	-8.2
SWE-Xplorer (full)	**61.4**	—
w/o cross-path reconciliation	58.2	-3.2
w/o deferred termination	58.2	-3.2
w/o automated solution augmentation	60.2	-1.2
w/o recursive tournament voting	59.6	-1.8
w/o convergence-forcing	58.4	-3.0
w/o path-level selection	55.2	-6.2

4.6 Error Analysis

Figure 2 also exposes a selection gap: the candidate-pool ceiling exceeds the reported resolved rate by 1.6 points at budget 10 and 5.6 points at 50. These are tasks for which the search finds a resolving patch but the final selection chooses another candidate. The widening gap points to candidate selection as a source of remaining error.

Recursive tournament voting selects the final patch through pairwise comparisons. At a budget of 50, we assess this selection step by comparing each judge decision with the held-out harness verdicts for pairs where exactly one candidate resolves the issue; pairs where both or neither resolve it do not have a correct choice. Across 359 discriminative comparisons, the judge advances the resolving candidate 290 times (80.8%; Table 4). Accuracy is similar across rounds, but a wrong choice can eliminate a correct patch before the final vote. Such errors help explain the gap in Figure 2 and point to better candidate selection as an avenue for improvement. Appendix E.2 further classifies unresolved instances across all evaluated agents by empty submissions and incorrect patches.

Table 4: Tournament judge accuracy on discriminative pairs, where exactly one candidate passes the harness (SWE-bench Verified, gpt-5-mini, budget 50).

	Discriminative pairs	**Judge accuracy**
Round 1	204	79.9%
Round 2	100	83.0%
Round 3	55	80.0%
All rounds	359	**80.8%**

5 Conclusion

We presented SWE-Xplorer, which organizes path-level tree search into repeated exploration, reconciliation, and frontier-compression iterations for autonomous issue resolution. It consistently improves over its single-trajectory backbone across three model families on SWE-bench Verified and SWE-bench Multilingual, showing that the benefits extend across models and beyond Python repositories. On SWE-bench Verified with gpt-5-mini, increasing the test-time budget from 30 to 50 improves performance from 54.6% to 61.4%, while mini-SWE-agent and SWE-Search largely plateau over the same range. The comparison with SWE-Search further demonstrates improved accuracy and efficiency over prior tree search in the evaluated setting, while retaining the compute overhead of search relative to a single trajectory. More broadly, the iteration design provides a structured way to scale search while repeatedly carrying forward coherent trajectories and reconciled evidence; this principle may transfer to other agentic tasks with state-preserving and state-changing actions.

Reproducibility Statement

To support reproducibility, we plan to release the full implementation of SWE-Xplorer upon publication. The artifact will include source code, configuration files, prompts, and scripts for reproducing the experiments reported in this paper.

In particular, the release will provide the implementation of the tree-search procedure and its mini-SWE-agent integration; benchmark configurations for SWE-bench Verified and SWE-bench Multilingual; the hyperparameters governing the branching factor, frontier size, iteration schedule, expansion budget, reward model, reconciliation, and final-solution selection; and per-instance outputs and evaluation logs needed to verify the reported aggregate results. We will also document the software environment and commands required to run the experiments, including the SWE-Search baseline configuration and the cost and runtime aggregation procedure. The planned public repository will include the evaluated implementations and their version history, along with the analysis scripts used to produce Appendix E.

AI use statement

Generative AI tools were used to assist with language editing and formatting. The authors reviewed all AI-assisted changes and take full responsibility for the paper's text, claims, and artifacts.

References

Vaibhav Aggarwal, Ojasv Kamal, Abhinav Japesh, Zhijing Jin, and Bernhard Schölkopf. Dars: Dynamic action re-sampling to enhance coding agent performance by adaptive tree traversal. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 19808–19855, 2025.

Anomaly. OpenCode: The open source ai coding agent. https://github.com/anomalyco/opencode, 2026. Software, accessed September 2, 2026.

Anthropic. Claude Code. https://github.com/anthropics/claude-code, 2026. Software, accessed September 2, 2026.

Antonis Antoniades, Albert Örwall, Kexun Zhang, Yuxi Xie, Anirudh Goyal, and William Wang. Swe-search: Enhancing software agents with monte carlo tree search and iterative refinement. In *International Conference on Learning Representations*, volume 2025, pp. 64485–64515, 2025.

Hyungjoo Chae, Seonghwan Kim, Junhee Cho, Seungone Kim, Seungjun Moon, Gyeom Hwangbo, Dongha Lim, Minjin Kim, Yeonjun Hwang, Minju Gwak, et al. Web-shepherd: Advancing prms for reinforcing web agents. *Advances in Neural Information Processing Systems*, 38:63314–63356, 2026.

Mahir Labib Dihan, Tanzima Hashem, Mohammed Eunus Ali, and Md Rizwan Parvez. Weboperator: Action-aware tree search for autonomous agents in web environment. *arXiv preprint arXiv:2512.12692*, 2025.

Yifeng Ding and Lingming Zhang. Swe-replay: Efficient test-time scaling for software engineering agents. *arXiv preprint arXiv:2601.22129*, 2026.

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

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? In *International Conference on Learning Representations*, volume 2024, pp. 54107–54157, 2024.

Joongwon Kim, Wannan Yang, Kelvin Niu, Hongming Zhang, Yun Zhu, Eryk Helenowski, Ruan Silva, Zhengxing Chen, Srini Iyer, Manzil Zaheer, Daniel Fried, Hannaneh Hajishirzi, Sanjeev Arora, Ruslan Salakhutdinov, Gabriel Synnaeve, and Anirudh Goyal. Scaling test-time compute for agentic coding. In *Third Conference on Language Modeling*, 2026. URL https://openreview.net/forum?id=J9kWWkRzUM.

Jing Yu Koh, Stephen Marcus McAleer, Daniel Fried, and Ruslan Salakhutdinov. Tree search for language model agents. *Transactions on Machine Learning Research*, 2025. ISSN 2835-8856. URL https://openreview.net/forum?id=QF0N3x2XVm.

Jacky Kwok, Shulu Li, Pranav Atreya, Yuejiang Liu, Yixing Jiang, Chelsea Finn, Marco Pavone, Ion Stoica, and Azalia Mirhoseini. LLM-as-a-verifier: A general-purpose verification framework. In *COLM 2026 The 2nd Workshop on Lifelong Agents: Learning, Aligning, and Evolving*, 2026. URL https://openreview.net/forum?id=cMipVV8wEo.

Stephen Robertson and Hugo Zaragoza. The probabilistic relevance framework: Bm25 and beyond. *Foundations and trends® in information retrieval*, 4(1-2):1–174, 2009.

Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling llm test-time compute optimally can be more effective than scaling parameters for reasoning. In *International Conference on Learning Representations*, volume 2025, pp. 10131–10165, 2025.

Aditya Bharat Soni, Rajat Ghosh, Vaishnavi Bhargava, Valerie Chen, and Debojyoti Dutta. Swe-tester: Training open-source llms for issue reproduction in real-world repositories. *arXiv preprint arXiv:2601.13713*, 2026.

Qwen Team. Qwen2.5: A party of foundation models, September 2024. URL https://qwenlm.github.io/blog/qwen2.5/.

Xinchen Wang, Pengfei Gao, Xiangxin Meng, Chao Peng, Ruida Hu, Yun Lin, and Cuiyun Gao. Aegis: An agent-based framework for general bug reproduction from issue descriptions. *arXiv preprint arXiv:2411.18015*, 2024.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. Openhands: An open platform for ai software developers as generalist agents. In *International Conference on Learning Representations*, volume 2025, pp. 65882–65919, 2025.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying llm-based software engineering agents. *arXiv preprint arXiv:2407.01489*, 2024.

Guowei Xu, Zhenting Qi, Huangyuan Su, Weirui Ye, Himabindu Lakkaraju, Sham M Kakade, and Yilun Du. Self-improving language models with bidirectional evolutionary search. *arXiv preprint arXiv:2605.28814*, 2026.

John Yang, Carlos Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652, 2024.

John Yang, Kilian Lieret, Carlos Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. Swe-smith: Scaling data for software engineering agents. *Advances in Neural Information Processing Systems*, 38, 2026.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *Advances in neural information processing systems*, 36:11809–11822, 2023a.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*, 2023b. URL https://openreview.net/forum?id=WE_vluYUL-X.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in neural information processing systems*, 36:46595–46623, 2023.

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. Language agent tree search unifies reasoning, acting, and planning in language models. In *Forty-first International Conference on Machine Learning*, 2024. URL https://openreview.net/forum?id=njwv9BsGHF.

A Extended Related Work

SWE-bench and coding agent frameworks. SWE-bench (Jimenez et al., 2024) defines the task of resolving real GitHub issues as a patch-generation problem evaluated against hidden test suites, and has become the standard benchmark for autonomous software engineering. Early agents such as SWE-agent (Yang et al., 2024) adopt a ReAct-style loop (Yao et al., 2023b) over a specialized shell interface. OpenHands (Wang et al., 2025) provides a general platform in which software agents edit code and interact with a sandboxed command line. Agentless (Xia et al., 2024) decouples file localization from patch generation, using retrieval as a preprocessing step. mini-SWE-agent, the single-step agent underlying SWE-Xplorer, is a lightweight ReAct agent that serves as the backbone for our tree search. These frameworks all operate as single-trajectory systems; SWE-Xplorer wraps such an agent in a search procedure that enables genuine multi-strategy exploration without modifying the agent itself.

Test-time compute scaling for agentic coding. Scaling inference compute rather than model parameters has emerged as a powerful strategy for both reasoning (Snell et al., 2025) and agentic tasks. CodeMonkeys (Ehrlich et al., 2025) demonstrates both serial and parallel test-time scaling on SWE-bench by sampling multiple iterative editing trajectories and selecting among their candidate patches. SWE-Replay (Ding & Zhang, 2026) reduces the redundancy of independent rollouts by recycling earlier trajectories and selectively branching from critical intermediate steps. This is closely related to our use of shared prefixes, but SWE-Replay branches retrospectively from archived trials, whereas SWE-Xplorer constructs and scores an online search tree and can reconcile evidence across concurrent frontier paths. Kim et al. (2026) systematically study test-time scaling for coding agents, identifying two complementary paradigms: *parallel scaling*, which runs multiple independent rollouts and selects the best result, and *sequential scaling*, which conditions successive rounds on information from earlier attempts. They show that combining both—iterating on top-k solutions per round with recursive tournament voting—yields the largest gains. SWE-Xplorer builds on this iterative structure but replaces independent rollouts with a shared-prefix tree search, allowing common trajectory prefixes to be reused rather than re-executed across independent rollouts. Cross-path reconciliation, path-level selection, and deferred termination are introduced as principled additions that have no counterpart in the parallel or sequential scaling paradigms.

Tree search for coding and language agents. The closest coding-agent search systems are SWE-Search (Antoniades et al., 2025), which applies Monte Carlo Tree Search to SWE-bench, and DARS (Aggarwal et al., 2025), which adaptively branches completed trajectories at consequential action points using their execution feedback. SWE-Search manages codebase state through a custom framework-internal layer that tracks and reverses file modifications programmatically. SWE-Xplorer delegates state management to Git instead, making state restoration a single atomic `git checkout` and eliminating an entire class of edge cases around new files, deleted files, and binary assets. Beyond state management, SWE-Xplorer differs from these systems in its path-level selection criterion, iteration-based pruning structure, and cross-path reconciliation. In particular, DARS traverses alternatives depth-first after observing a completed trajectory, while SWE-Xplorer maintains a scored frontier and explicitly synthesizes compatible live branches before pruning. Our evaluation also compares these complete systems empirically: with gpt-5-mini, an expansion budget of 50, and branching factor 2 on SWE-bench Verified, SWE-Xplorer achieves higher resolved rate with lower average cost and runtime than SWE-Search (Section 4.4). The comparison does not attribute these differences to state management alone.

More broadly, Tree of Thoughts (Yao et al., 2023a) applies deliberate search to multi-step reasoning, and LATS (Zhou et al., 2024) adapts Monte Carlo Tree Search to language-model agents using self-reflection as the value signal. Concurrent work on Bidirectional Evolutionary Search (BES) (Xu et al., 2026) recombines partial reasoning trajectories through evolutionary operators such as combination, translocation, and crossover. This is conceptually related to cross-path reconciliation, but BES edits abstract trajectory segments in a general search framework, whereas SWE-Xplorer reconciles compatible live coding-agent branches grounded in a shared repository state and uses each merged node as a joint representative during frontier pruning. In web navigation, WebOperator (Dihan et al., 2025) applies best-first search and introduces context variation, action deduplication, and format-transparent retry—three techniques we adopt and extend (Section 3.2). Our key distinctions

are path-level selection, iteration-based frontier compression, and state-grounded reconciliation of executable coding-agent branches.

B LIMITATIONS

Compute overhead and comparison scope. Despite achieving higher accuracy at lower cost and runtime than the evaluated SWE-Search configuration, SWE-Xplorer costs 5× as much and takes approximately 7.6× as long as mini-SWE-agent on SWE-bench Verified with gpt-5-mini. Matching the expansion budget and branching factor does not match token usage or auxiliary model calls. The SWE-Search comparison is limited to this backend and benchmark, and does not establish the same advantage across other models or languages. The budget-scaling analysis is likewise limited to gpt-5-mini on SWE-bench Verified, so whether the same scaling pattern holds for other models and benchmarks remains open. Runtime depends on execution conditions, including model serving, tool execution, and concurrency; aggregate averages alone cannot identify the source of the observed speedup or quantify run-to-run variability.

Destructive actions in real-world deployment. SWE-bench provides a sandboxed repository environment: actions are confined to the codebase and executing code does not affect any external state. In real-world software development, however, an agent may encounter *destructive actions*—operations whose effects propagate outside the codebase, such as database writes, network requests, or filesystem changes beyond the repository. Such actions cannot be undone by a `git checkout`, and their consequences may be difficult or impossible to reverse. SWE-Xplorer's state management design assumes that all environment effects are captured by version control, an assumption that breaks in the presence of destructive actions. Extending the framework to high-stakes real-world settings would require detecting and sandboxing destructive actions before they are executed, a direction we leave for future work.

Files excluded from version control. SWE-Xplorer relies on Git to track codebase state, which means it cannot detect modifications to files that are excluded from version control via `.gitignore`. If the agent modifies a gitignored file, the change is invisible to the framework: the state hash computed by `git write-tree` will not reflect it, state restoration via `git checkout` will not revert it, and the cache may return stale observations for subsequent actions from the same apparent state. In practice this limitation is benign for SWE-bench tasks, where the files relevant to fixing an issue are almost always tracked, but it is a principled gap in the framework's correctness guarantee for repositories with significant gitignored state.

C IMPLEMENTATION DETAILS

C.1 ITERATION SCHEDULE AND HYPERPARAMETERS

Table 5: SWE-Xplorer's iteration schedule and search hyperparameters.

Hyperparameter	Value
Total expansion budget B	50
Number of iterations N	5
Expansions per iteration	10
Branching factor b	2
Top-k at pruning	4
Path score discount γ	0.85
Path score window W	3
Tournament pool size	$2k = 8$
Deadline reminder threshold	5 expansions remaining
Early-stop submission threshold	2 terminating candidates
Early-stop uniqueness threshold	1 distinct edited state

C.2 Action Generation Mechanisms

At each expansion (Section 3.2), the search samples b candidate next actions from the selected node and applies three mechanisms—adapted from WebOperator (Dihan et al., 2025)—to ensure the resulting children are diverse, valid, and non-redundant.

Context variation. Rather than sampling the same prompt b times, each sample receives a distinct prompt context—for example, appending different subsets of top-ranked relevant files (Appendix C.6) to the issue description. This encourages branches to approach each step from different angles, increasing the diversity of candidate actions.

Duplicate action merging. Even with context variation, the b parallel samples may produce identical or near-identical actions. Such duplicates are collapsed into a single candidate before the corresponding branches are added to the search tree, ensuring that the resulting branches represent distinct actions rather than sampling redundancy. Consequently, the effective branching factor may be smaller than the configured value b, which avoids spending the expansion budget on redundant branches.

Invalid-action recovery. When a sampled response fails to parse into a valid action, the agent retries with an error message appended, and only the final valid action is inserted into the tree. Retry attempts are hidden from the search context: the tree records only successful, intentional actions, keeping the recorded trajectory clean and free of formatting-error noise.

C.3 Reward Model Implementation

After each action, an action-type-specific LLM evaluator scores the resulting action-observation pair. Raw LLM scores are prone to clustered ties that limit their ability to distinguish candidates (Zheng et al., 2023); rather than relying on scoring-token logits or repeated evaluator sampling (Kwok et al., 2026; Koh et al., 2025; Chae et al., 2026), which respectively require API-level logit access and add inference cost, we instead adjust a single LLM score using directly observable execution evidence, described next.

The evaluator scores reads and searches by information gain, edits by fix effectiveness, tests by diagnostic value, and submits by termination readiness. Each type-specific scoring call receives the issue description, a truncated action history (up to a configurable number of prior steps, with the oldest dropped first if context is exceeded), the current action and its reasoning, and the observation. The score is returned as an integer in $[0, 100]$ inside a structured `<score>` XML tag and passed through the shaping pipeline below to obtain the node reward $v(n)$. Prompt templates for each action type are provided in Section I.

After the type-specific LLM score $K_0 \in [0, 100]$ is computed, the following adjustments are applied in order. All retrieval and similarity signals are rescaled to $[0, 100]$ before blending so that their units match the LLM score. Let K denote the running adjusted score.

1. **Format/execution penalties.** Actions with null commands are multiplied by 0.90. Non-zero return codes (excluding test-expected failures) are multiplied by 0.95. Timed-out actions are multiplied by 0.90.
2. **Edit-specific adjustments.** For code edits, the reward is blended with BM25 file relevance: $0.7 \cdot K + 0.3 \cdot (100\rho_{\text{BM25}})$, where $\rho_{\text{BM25}} \in [0, 1]$ is the normalized BM25 score of the edited file against the issue tokens. Test regressions apply a penalty of $\max(0.9,\ 1.0 + 0.1 \cdot \Delta_{\text{test}})$ where $\Delta_{\text{test}} < 0$ is the number of newly failing tests. Test improvements apply a bonus of $\min(1.1,\ 1.0 + 0.1 \cdot \Delta_{\text{test}})$. Large diffs ($> 50$ changed lines) apply an exponential penalty $e^{-(|\text{diff}|-50)/100}$.
3. **Read-specific adjustments.** Empty observations are multiplied by 0.70. Observations exceeding 5,000 characters are penalized as $\max(0.8,\ e^{-(|o|-5000)/3000})$. If read files have a pre-computed BM25 relevance score, the reward is blended: $0.9 \cdot K + 0.1 \cdot (100\rho_{\text{BM25}})$.
4. **Global semantic similarity.** A sentence-transformer similarity score $s \in [0, 1]$ between the action-observation pair and the issue text is blended into the final reward: $v(n) = 0.7 \cdot K + 0.3 \cdot (100s)$.

The test signal comes from a lightweight reproduction agent that constructs a script before search and applies it transiently after edits; successful fixes receive a bonus and regressions a penalty.

Appendix C.4 describes this procedure. This follows work on agent-based bug reproduction and test generation (Wang et al., 2024; Soni et al., 2026).

C.4 Test Reproduction Agent

Before tree search begins on each instance, a dedicated reproduction agent attempts to reproduce the failing behavior described in the issue. The agent runs a simple ReAct loop with an interaction-step budget of 30, instructed to locate and execute the test described in the issue and produce a self-contained script that records structured test outcomes.

The reproduction script is not committed to the repository. During tree search, it is applied transiently to each edited codebase state at every `edit` action: the test outcomes are parsed to determine whether previously failing tests now pass, and the script is removed without affecting the recorded codebase state. This gives the reward model an execution-based signal from the agent-generated reproduction script at every edit, rather than relying on a single end-of-episode evaluation.

If the reproduction agent fails to produce a valid script—for example, because the issue does not include a reproducible test case—the test feedback signal is omitted and the reward model falls back to LLM scoring and retrieval-based shaping alone.

C.5 Lightweight State Management via Version Control

Tree search must restore the repository state associated with any selected branch. Rather than implementing its own mechanism for tracking and restoring changed files, as in prior coding-agent search systems (Antoniades et al., 2025), SWE-Xplorer uses Git directly in the task repository.

Every `edit` action is recorded immediately as a Git commit. Non-edit nodes inherit the parent's commit because they do not change tracked content. Before expanding a selected node, the framework compares its recorded commit with the active commit and, when necessary, restores it with a single `git checkout`; no action replay or file-level reversal is required.

Each node also stores the content-addressed tree hash returned by `git write-tree`. Commits with the same tree hash are treated as the same codebase state, independent of their histories. Action results are cached by $(\text{state hash},\ \text{action})$, so repeating an action from an identical state reuses its observation without re-execution.

Because agent-issued version-control commands could corrupt this history, they are rejected as malformed actions and passed to the retry mechanism (Section 3.2). For tracked content, Git handles additions, deletions, and binary files and provides atomic restoration. Limitations for external side effects and gitignored files appear in Appendix B.

C.6 BM25-Based File Ranking

At the start of each episode, all source files in the repository are extracted, tokenized, and indexed with BM25Okapi (Robertson & Zaragoza, 2009). The issue description is scored against this index, producing a ranked list of files with normalized relevance scores in $[0, 1]$.

This ranking serves two purposes. First, the relevance scores are used for the read- and edit-specific reward adjustments described in Appendix C.3. Second, the ranking provides context variation during action generation. In the primary configuration with $b = 2$, the base prompt is used for one sample, while the top-ranked files are appended to the other using the retrieval template in Appendix H.

C.7 Solution Selection Details

Solution summaries for tournament voting are generated by an LLM given the full action-observation trajectory and the resulting code diff. The summary prompt asks for: root cause identified, files and modules modified, main fix strategy, edge cases handled, regression risk, and final outcome. Summaries are cached per action to avoid redundant LLM calls across tournament rounds.

The composite pre-sort score for tournament seeding is:

$$s(n) = 0.8 \cdot \bar{\tilde{v}}(n) + 0.1 \cdot \left(1 - \tfrac{\text{order}(n)}{B}\right) + 0.1 \cdot \mathbf{1}[\text{LLM-generated}] + \epsilon \cdot \overline{V_{\text{path}}}(n),$$

where $\mathrm{order}(n)$ is the expansion index at which the action's parent was expanded, B is the total expansion budget, bars denote scores normalized to $[0, 1]$ over the candidate pool, and ϵ is a small tiebreaker weight.

D SEARCH ALGORITHM

Algorithm 1 gives the main search loop described in Sections 3.2–3.5; Algorithm 2 details the cross-path reconciliation procedure it invokes at each iteration boundary.

Initialization. The reproduction agent runs once on the issue $\mathcal{I}$ to produce a fail-to-pass test script $\mathcal{R}$ (Appendix C.4), used throughout the search to score edit actions. The tree $\mathcal{T}$ starts from a single root node n_0 with an empty observation and the initial codebase state $\mathcal{C}_0$. The frontier $\mathcal{F}$ initially contains only n_0, and the deferred submit pool $\mathcal{S}$ starts empty. The expansion budget B is divided into N iterations of B/N expansions each.

Node selection and expansion. At each expansion, the search selects $n^* = \arg\max_{n \in \mathcal{F}} V_{\mathrm{path}}(n)$—restricted to the current path once in the final iteration (Section 3.3)—and restores the environment to n^*'s recorded codebase state via Git if it differs from the active commit (Appendix C.5). It generates b candidate actions from n^* (Appendix C.2), deduplicating and retrying failed generations. Each action is executed, or served from the cache if an identical action was already executed from the same state, and, if it is an edit, evaluated against $\mathcal{R}$ to obtain a test-outcome delta. The reward model scores the resulting observation together with this delta, and the child node it produces is added to $\mathcal{F}$. A `submit` action instead moves its node directly into $\mathcal{S}$, deferring termination rather than ending the search (Section 3.5).

Iteration boundary. Once the current iteration has completed B/N expansions, $\mathcal{S}$ is augmented with a programmatic submit for every distinct edited codebase state not already represented (automated solution augmentation, Section 3.5). Next, cross-path reconciliation enriches $\mathcal{F}$ (Algorithm 2). The frontier is then pruned to its top-k nodes by path score. Before iteration $N - 1$, edited paths take priority among the k survivors; any remaining slots go to the highest-scoring unedited paths. Before iteration N, pruning retains only the highest-scoring edited path, or the highest-scoring path overall if none is edited (Section 3.3). Thus, one path is carried into the final iteration for local refinement. Reconciliation still runs at that boundary. It can merge two paths sharing a codebase state into the one node carried forward. The search also terminates early if pruning empties $\mathcal{F}$, since only submitted candidates would remain.

Termination. When the expansion budget is exhausted, $\mathcal{S}$ receives one final round of automated solution augmentation, and the accumulated candidates are passed to recursive tournament voting (Section 3.5), which returns the single selected patch.

Cross-path reconciliation. Algorithm 2 greedily pairs frontier nodes that share an identical codebase state, considering the closest pairs by tree distance first. For each pair, it synthesizes one merged node by proposing and executing a single action from the shared state. The resulting action–observation pair is scored twice, once against each trajectory's own history. The two scores are combined by a path-score-weighted average (Section 3.4). The merged node is attached to whichever original node gives its action–observation pair the higher reward, as its sole structural parent. Thus, the tree retains a single lineage even though two histories informed the merge.

E ADDITIONAL QUANTITATIVE ANALYSIS

This appendix reports several analyses computed directly from the released per-instance evaluation logs, referenced from the main text, so they can be independently regenerated from the released artifact.

Algorithm 1 SWE-Xplorer Search

Require: Issue $\mathcal{I}$, codebase $\mathcal{C}_0$, expansion budget B, iterations N, branching factor b, frontier size k
Ensure: Final patch

Run reproduction agent on $\mathcal{I}$; obtain test script $\mathcal{R}$
Initialise tree $\mathcal{T}$ with root node n_0 (empty observation)
$\mathcal{F} \leftarrow \{n_0\}$ ▷ frontier
$\mathcal{S} \leftarrow \emptyset$ ▷ deferred submit candidates
expansions $\leftarrow 0$, iter $\leftarrow 1$, budget_per_iter $\leftarrow B/N$
while expansions $< B$ **do**
  // Select next node to expand
  if iter $= N$ **then**
    $n^* \leftarrow \arg\max_{n \in \mathcal{F}_{\text{local}}} V_{\text{path}}(n)$ ▷ local best-first in final iteration
  else
    $n^* \leftarrow \arg\max_{n \in \mathcal{F}} V_{\text{path}}(n)$
  end if
  $\mathcal{F} \leftarrow \mathcal{F} \setminus \{n^*\}$
  // State restoration if needed
  if current active commit $\neq n^*$'s recorded commit **then**
    Restore environment to n^*'s recorded commit
  end if
  // Generate b candidate actions with context variation
  $A \leftarrow$ *GenerateActions*$(n^*, b, \mathcal{I})$
  $A \leftarrow$ *DeduplicateAndRetry*(A)
  // Execute, score, and insert children
  for each action $a \in A$ **do**
    if $(h(n^*), a)$ in cache **then**
      $(o, s') \leftarrow$ cache$[(h(n^*), a)]$
    else
      Execute a; obtain observation o and new state s'
      cache$[(h(n^*), a)] \leftarrow (o, s')$
    end if
    if $\tau(a) =$ `edit` **then**
      Apply $\mathcal{R}$ transiently; compute test outcome delta Δ
    end if
    $v(n) \leftarrow$ *RewardModel*$(a, o, \Delta, \mathcal{I})$
    Create child node n; append to $\mathcal{T}$ and $\mathcal{F}$
    if $\tau(a) =$ `submit` **then**
      $\mathcal{S} \leftarrow \mathcal{S} \cup \{n\}$; remove n from $\mathcal{F}$
    end if
  end for
  expansions $\leftarrow$ expansions $+ 1$
  // Iteration boundary
  if expansions $\bmod$ budget_per_iter $= 0$ **and** iter $< N$ **then**
    Augment $\mathcal{S}$ with programmatic submits for all edited states
    $\mathcal{F} \leftarrow$ *Reconcile*$(\mathcal{F}, \mathcal{T}, \mathcal{S})$ ▷ enrich frontier before pruning; Algorithm 2
    if iter $= N - 1$ **then**
      $\mathcal{F} \leftarrow$ top-1 node in $\mathcal{F}$ by V_{path}, restricted to paths with ≥ 1 edit
    else
      $\mathcal{F} \leftarrow$ top-k nodes in $\mathcal{F}$ by V_{path}
    end if
    iter $\leftarrow$ iter $+ 1$
  end if
  if $\mathcal{F} = \emptyset$ **then**
    break ▷ all remaining candidates are submits
  end if
end while
Augment $\mathcal{S}$ with programmatic submits for all remaining edited states
return *TournamentVoting*$(\mathcal{S})$

Algorithm 2 Cross-Path Reconciliation

Require: Frontier $\mathcal{F}$, tree $\mathcal{T}$, submit candidates $\mathcal{S}$
Ensure: Frontier $\mathcal{F}'$ of merged nodes replacing reconciled pairs

$\mathcal{F}' \leftarrow \mathcal{F}$
$\mathcal{P} \leftarrow$ *GreedyDisjointPairs*$(\mathcal{F})$ ▷ equal state hash; shortest tree distance first
for each reconcilable pair $(n_A, n_B) \in \mathcal{P}$ **do**
    Construct prompt: shared history to n_c, then distinct sequences for n_A and n_B
    $a_m \leftarrow$ *LLM*("reason across both and decide the best next step")
    Execute a_m once from the shared codebase state of n_A, n_B; obtain observation o_m and resulting state s_m
    $r_A \leftarrow$ *RewardModel*$(a_m, o_m \mid \text{history}(n_A))$
    $r_B \leftarrow$ *RewardModel*$(a_m, o_m \mid \text{history}(n_B))$ ▷ same realised outcome; no second execution
    $w_A \leftarrow V_{\text{path}}(n_A) / \left(V_{\text{path}}(n_A) + V_{\text{path}}(n_B)\right)$; $w_B \leftarrow 1 - w_A$
    $\tilde{v}(m) \leftarrow w_A \cdot r_A + w_B \cdot r_B$
    $n_{\text{par}} \leftarrow \arg\max(r_A, r_B)$ ▷ single structural parent
    Create node m as child of n_{par}; set $\tilde{v}(m)$
    $\mathcal{F}' \leftarrow (\mathcal{F}' \setminus \{n_A, n_B\}) \cup \{m\}$
end for
return $\mathcal{F}'$

E.1 Reward Shaping

Appendix C.3 motivates the shaping pipeline applied to each type-specific LLM score by noting that raw LLM judgments are prone to clustered ties (Zheng et al., 2023). We check this directly against the search-tree logs: Figure 4 compares the raw LLM score against the final shaped reward actually used for node selection, across all 22,337 scored actions in the main gpt-5-mini run.

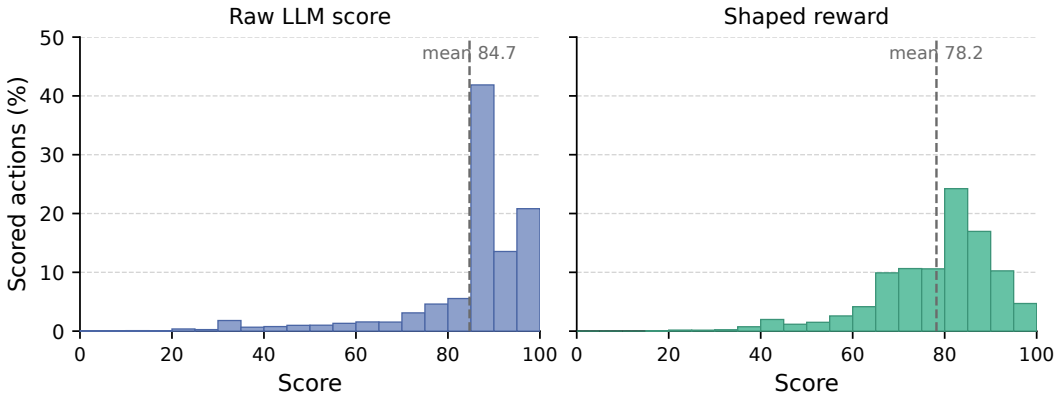


Figure 4: Distribution of the raw LLM judge score and the final shaped reward, across every scored action in the main gpt-5-mini run (22,337 actions, 500 instances). Dashed lines mark each distribution's mean.

The raw score is heavily clustered: 41.9% of all actions receive a score in the single 85–90 bin, with a mean of 84.7 and a median of 87.3, consistent with LLM judges defaulting to a narrow band of high, round-numbered scores rather than discriminating between candidates. Shaping breaks up this clustering—the largest bin's share drops to 24.2% and the mean falls to 78.2—by folding in the format, execution, and semantic-similarity adjustments of Appendix C.3, each of which depends on the specific action and observation rather than the judge's raw impression. The result is a reward signal with more usable spread between candidates that would otherwise receive identical or near-identical raw scores, which is what path-level and tournament-based selection rely on to discriminate between them.

Does shaping improve path guidance? A less-clustered reward is not automatically a more *useful* one: we also check whether shaping makes the reward better at pointing toward the branch that

actually resolves the issue. For every terminating node (agent-issued `Submit` or system-generated, Section 3.5) with a recorded pass/fail outcome, we take three increasingly path-aware scores—the node's own value, an unweighted mean over its full root-to-node path, and the discounted, windowed path score $V_{\text{path}}(n)$ of Section 3.2—computed once from the raw score and once from the shaped reward, and ask how often the passing candidate in a pair outscores the failing one. Comparisons are restricted to candidate pairs from the *same* tree (194 of 500 instances have both a pass and a fail terminating candidate to compare; 2,671 such pairs in total), since pooling across trees would let a task's own difficulty, which shifts both scores and the pass rate together, masquerade as a discrimination signal.

Figure 5 shows shaping improves this discrimination at all three aggregation levels, most on the terminating node's own score (0.683 → 0.746 AUC) and by a smaller, consistent margin once averaged over the path (0.677 → 0.727; 0.663 → 0.714 for V_{path}). A sign test across the 133 instances where the two scores disagree confirms this is not driven by a handful of large trees: shaping wins on 85 instances versus 48 for the raw score ($p \approx 0.002$, two-sided binomial test). All AUCs stay well below the tournament judge's 80.8% pairwise accuracy (Section 4.6), so a single scalar reward, shaped or not, remains a substantially weaker discriminator than a pairwise LLM comparison over full trajectory summaries—consistent with Section 3.5's motivation for using tournament voting rather than reward score alone to pick the final candidate.

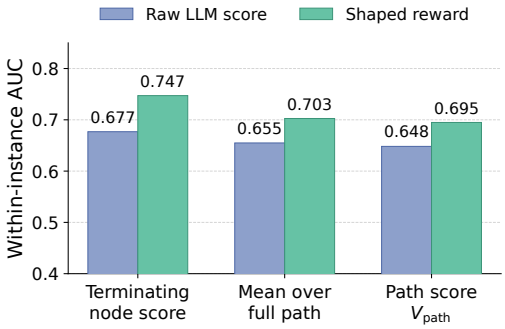


Figure 5: Within-instance AUC—how often the passing candidate in a same-tree pass/fail pair outscores the failing one—for the raw LLM score versus the shaped reward, at three levels of path aggregation.

E.2 Failure-Mode Taxonomy

Table 6 classifies unresolved instances into empty submissions (no patch generated) and wrong-but-non-empty patches (a patch was submitted and applied but did not pass the test suite), for every evaluated agent on SWE-bench Verified with gpt-5-mini: mini-SWE-agent, OpenHands, SWE-Search, OpenCode, Claude-Code, and SWE-Xplorer. SWE-Xplorer and SWE-Search are the only two of the six that never submit an empty patch: every instance reaches at least one syntactically valid submission, via automated solution augmentation and deferred termination for SWE-Xplorer, and via tree search's own fallback candidates for SWE-Search. The four single-trajectory baselines each leave a nontrivial fraction of instances with no patch at all: mini-SWE-agent 22/500 (4.4%), OpenHands 23/500 (4.6%), OpenCode 3/500 (0.6%), and Claude-Code 9/500 (1.8%). For all six agents, the dominant failure mode among unresolved instances is a plausible-looking patch that is submitted, applies cleanly, but fails the held-out test suite.

Table 6: Classification of unresolved instances into empty submissions, wrong-but-non-empty patches, and errors/unscored instances, for every evaluated agent on SWE-bench Verified with gpt-5-mini.

Configuration	Resolved	Empty patch	Wrong patch	Error
mini-SWE-agent	266	22	208	4
OpenHands	254	23	223	0
SWE-Search	274	0	224	2
OpenCode	260	3	237	0
Claude-Code	285	9	205	1
SWE-Xplorer	**307**	**0**	**193**	**0**

E.3 Cache Efficiency

The preceding analyses use the flattened per-instance harness reports; this one instead walks every instance's raw search-tree log directly, to check the caching mechanism of Section 3.2 against the logs rather than the single aggregate rate reported in the main text.

Table 7: Paired significance tests for SWE-Xplorer against each baseline. "Ours" and "Base" count instances resolved only by SWE-Xplorer or only by the baseline, respectively; Δ is the resolved-rate difference in percentage points with a paired Wald 95% confidence interval. The p values are from two-sided exact McNemar tests; all nine comparisons remain significant after Holm correction.

	Backbone	Baseline	Ours	Base	Δ [95% CI]	p
Verified	gpt-5-mini	mini-SWE-agent	63	22	+8.2 [+4.7, +11.7]	9.8×10^{-6}
		SWE-Search	65	32	+6.6 [+2.8, +10.4]	1.0×10^{-3}
		Claude-Code	51	29	+4.4 [+0.9, +7.9]	0.0183
		OpenCode	76	29	+9.4 [+5.5, +13.3]	5.1×10^{-6}
	Qwen-2.5-7B	mini-SWE-agent	17	1	+3.2 [+1.6, +4.8]	1.4×10^{-4}
	DeepSeek-V4-Flash	mini-SWE-agent	74	17	+11.4 [+7.8, +15.0]	1.2×10^{-9}
Multilingual	gpt-5-mini	mini-SWE-agent	68	16	+17.3 [+11.7, +23.0]	8.5×10^{-9}
	Qwen-2.5-7B	mini-SWE-agent	9	0	+3.0 [+1.1, +4.9]	0.0039
	DeepSeek-V4-Flash	mini-SWE-agent	83	16	+22.3 [+16.3, +28.3]	4.4×10^{-12}

An action whose (state, action) pair was previously executed elsewhere in the tree retrieves its observation from cache instead of executing the action again in the environment (Section 3.2). Aggregated over the main gpt-5-mini run, 2,378 of 23,337 total actions (10.2%) were cache hits. Figure 6 breaks this single aggregate number down per instance and shows the reuse is spread broadly across nearly every instance rather than concentrated in a handful of unusually repetitive trees: most instances fall in the 5-15% range, and very few see no reuse at all. This makes caching a direct, tree-structural source of SWE-Xplorer's cost and runtime advantage over SWE-Search (Figure 3), on top of any gain from reward-guided candidate selection itself, and confirms the benefit is a broad property of the search rather than an artifact of a few degenerate, highly-repetitive instances.

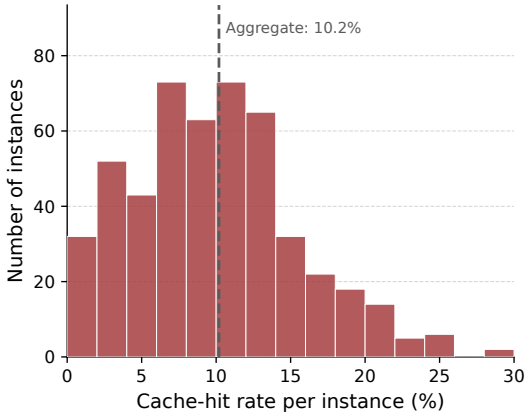


Figure 6: Per-instance cache-hit rate, all 500 SWE-bench Verified instances (main gpt-5-mini run). Dashed line: aggregate rate (10.2%).

E.4 Paired Significance Tests

Every agent here is scored on the same benchmark instances, so each comparison is paired: an instance is resolved by both agents, by neither, or by exactly one. Only that last group says anything about which agent is better; the concordant instances are uninformative and drop out. Table 7 reports an exact two-sided McNemar test over the discordant instances for each comparison. For the paired confidence interval, let D_i be 1 if only SWE-Xplorer resolves instance i, -1 if only the baseline resolves it, and 0 otherwise. The reported interval, in percentage points, is $100(\bar{D} \pm 1.96\, s_D/\sqrt{n})$, where s_D is the sample standard deviation over all n instances. The narrowest margin is against Claude-Code, at $+4.4$ points $[+0.9, +7.9]$: positive throughout, but the one comparison the data constrains only loosely.

An instance counts as resolved for an agent exactly when it appears in that agent's resolved list, over the full benchmark rather than the intersection of the agents' unresolved lists. Those lists are not interchangeable—some runs separate empty-patch and error cases from ordinary failures and others fold them together—so intersecting them would drop instances asymmetrically, discarding the empty-patch and error cases that the single-trajectory baselines produce and SWE-Xplorer does not, which would bias the comparison in SWE-Xplorer's favor.

E.5 Actions per Outcome

Table 8: Action-model calls per instance on resolved versus unresolved outcomes, on SWE-bench Verified with gpt-5-mini.

Agent	Mean actions (resolved)	Median actions (resolved)	Mean actions (unresolved)	Median actions (unresolved)
mini-SWE-agent	13.0	11.0	17.2	15.0
SWE-Xplorer	73.4	66.0	88.5	85.0

Following the steps-per-outcome analysis of single-trajectory agents in Kim et al. (2026), Table 8 compares action-model calls on resolved versus unresolved instances (excluding empty-patch and error/unscored instances, which carry no informative action count) for mini-SWE-agent and SWE-Xplorer; for SWE-Xplorer this counts every action across the whole search tree, not just the submitted path. Both agents take more actions on failures than on resolved instances: 32% more for mini-SWE-agent (17.2 vs. 13.0) and 21% more for SWE-Xplorer (88.5 vs. 73.4). This is the intended behavior of the iteration-based budget (Section C.1): easy instances resolve early with fewer actions (median 66 vs. 85), and the saved budget goes to harder instances instead.

F Qualitative Example

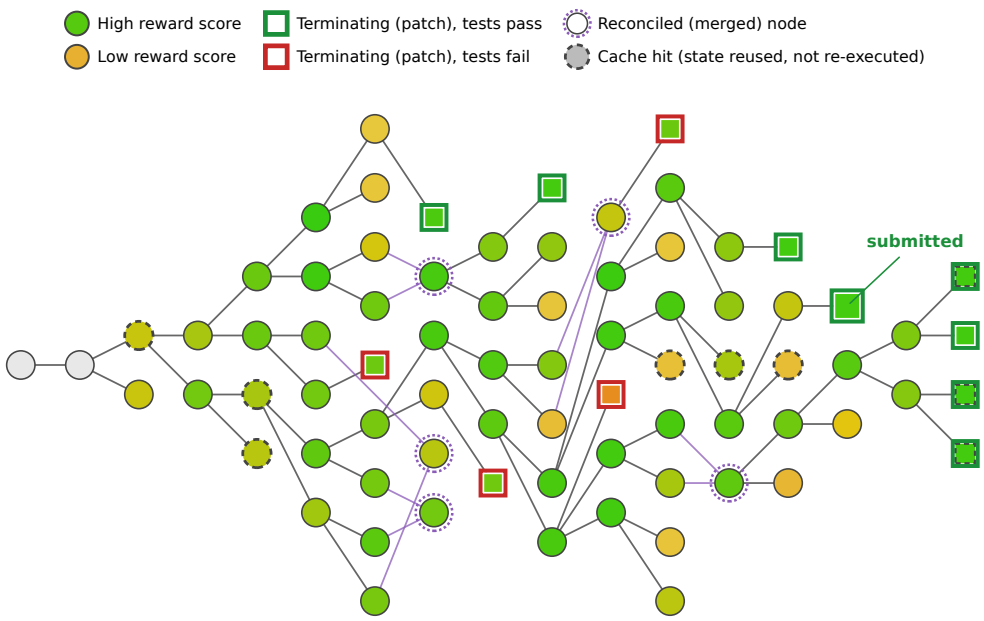
To make the search procedure in Section 3.2 and Algorithms 1–2 concrete, and in particular to show why continuing to search past an early complete-but-wrong attempt pays off, Figure 7 visualizes the complete, unpruned search tree SWE-Xplorer explored for `django__django-13569` (gpt-5-mini, budget 50). The underlying issue is a real SWE-bench Verified bug: calling `.annotate(Count(...)).order_by('?')` on a queryset unexpectedly folds the random ordering expression into the `GROUP BY` clause, breaking the aggregation, because the query compiler's `get_group_by` does not exclude the `Random` expression contributed by `order_by('?')`. The eventual fix updates `django/db/models/sql/compiler.py` to skip `Random` expressions when building `GROUP BY`.

Panel (a) shows that four terminating candidates fail the internal reproduction check, spread across depths 6, 8, 10, and 11 — complete-but-wrong attempts reached at several distinct points well before the end of a search of depth 16 overall. Had the search stopped at any of these (as a single-trajectory agent submitting its first complete attempt would), the issue would not be resolved. Instead, exploration continues for several more expansions past the last of these failures, through further cross-path reconciliation (5 events, purple), before reaching the candidate tournament voting ultimately submits at depth 14, two expansions short of the deepest branch explored, which does resolve the held-out test suite. One representative reconciliation event merges a branch that was inspecting how `Random` expressions are represented (checking for a `get_source_expressions` method) with a branch that was independently locating the compiler's `get_group_by` method to prepare a targeted edit — two investigations converging on the same fix from different code-reading angles, combined into one merged node rather than expanded separately.

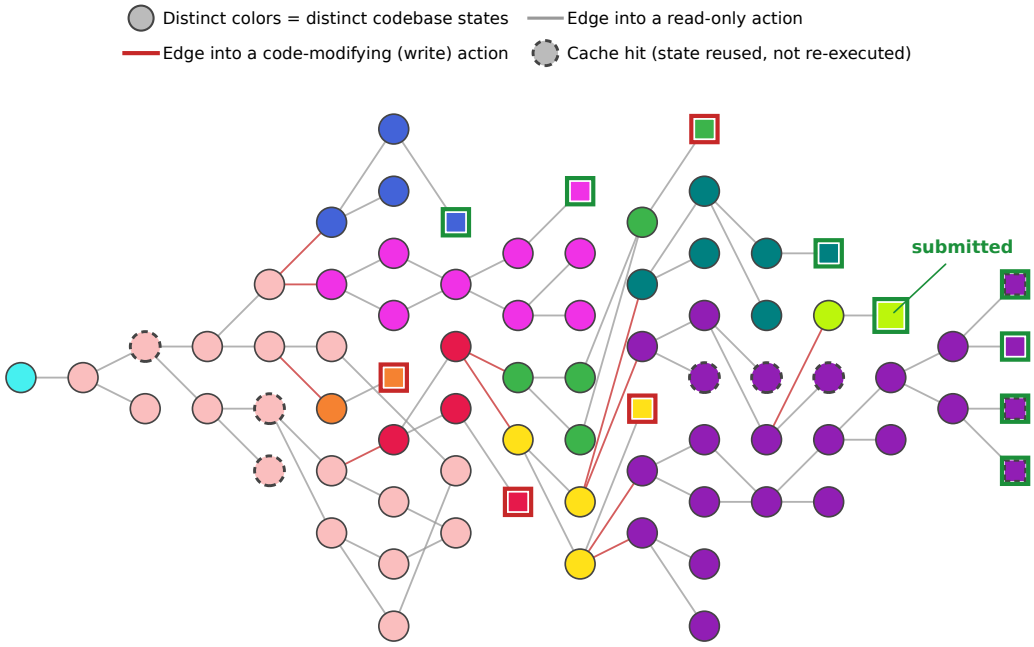
Panel (b) makes the git-based state management of Section C.5 concrete: the large teal cluster on the left is the shared pre-edit state that many read-only exploration actions branch from without changing it, while the large purple cluster on the right (23 of 77 nodes) is the state reached once a correct-shaped edit is applied, from which many further read-only verification and test actions branch without creating new states. Red edges — comparatively rare against the mostly gray, read-only edges — mark exactly where an edit changes the codebase and therefore the state color; four separate branches independently converge on the same intermediate state before three separate edit actions from that shared state all happen to produce the identical purple post-fix state, one concrete example of the redundant work that caching by (state, action) pair is designed to avoid on any later action repeated from an already-visited state.

F.1 The Submitted Trajectory, Condensed

Listing 1 distills the same instance's actually-submitted trajectory (30 agent actions in full) down to the handful of moments that make the underlying git-based state management (Appendix C.5) directly



(a) Node fill is the reward-model score (green = high, orange/red = low); dotted purple rings mark nodes produced by cross-path reconciliation, drawn with edges from both merged paths; solid green/red rings mark terminating nodes whose internal reproduction check passed/failed; the enlarged ring marks the patch tournament voting submitted.



(b) Node fill is the codebase state (content-addressed tree hash): nodes with identical fill share exactly the same codebase state regardless of which action path reached them, and edges are red where the action taken modifies code (an "edit") rather than only reading or searching.

Figure 7: The same complete search tree for `django__django-13569` (77 nodes, depth 16), colored two ways.

visible: the initial baseline commit before any action; a backtrack that needs no `git checkout` because the target state is already active; two edits that each advance the commit; and a second backtrack that *does* need an explicit `git checkout`, this time to a genuinely different, earlier commit, since the search abandons the second edit and reverts to the state left by the first; and a final `git checkout` plus `git restore` to the tournament-voting winner's state immediately before the submitted diff is produced. Reads and searches never appear next to a commit here because they never advance one: a read/search node's recorded commit is always identical to its parent's, verified directly against the released search-tree log. Most of the elided steps are repeated edit-then-relocate cycles across sibling branches independently converging on variants of the same fix and fixing the indentation it initially broke, consistent with the multiple same-purpose edits visible in Figure 7(b).

Listing 1: Condensed trajectory of the submitted patch for `django__django-13569`. `[git]` lines are framework bookkeeping, not agent actions.

```
<pr_description>
order_by('?') unexpectedly breaking queryset aggregation. Steps to reproduce:
class Thing(models.Model): pass
class Related(models.Model): thing = models.ForeignKey(Thing)
Thing.objects.annotate(rc=Count('related')).order_by('?').values('id','rc')
silently drops the GROUP BY grouping because order_by('?') injects a
Random() expression that get_group_by() folds into the GROUP BY clause.
</pr_description>

[git] Initial commit (repository baseline, before any agent action)
-> 69e87b473819fadef15cb4438b1314f51fd43460

[Step 1] READ
THOUGHT: I need to inspect the repository to find the Django compiler code
that handles group_by and order_by logic. I'll list files ...
$ ls -la
-> total 132
   drwxrwxrwx  ...  .git
   -rw-r--r--  ...  setup.py
   drwxrwxrwx  ...  django
   drwxrwxrwx  ...  tests
   ... (28 entries total)

[Step 2] SEARCH
THOUGHT: I will locate the SQL compiler file that contains get_group_by ...
$ grep -R "def get_group_by" -n django || true
-> django/db/models/sql/compiler.py:63:    def get_group_by(self, ...)

[Steps 3-7, elided] Five more read/search actions, across two sibling
branches, open and re-open get_group_by's implementation from different
angles. The active commit stays at the initial baseline throughout, since
none of these actions touch a tracked file.

[git] Backtracking to node ab8484f4-...
(no checkout needed: this branch's recorded commit already matches the one
currently active -- both are still the initial baseline above)

[Steps 8-12, elided] A first attempt edits compiler.py to skip
Random() expressions with a simpler check than what is eventually
submitted, then spends four steps re-verifying and relocating the edit
across sibling branches before the search abandons this branch.

[Step 13] EDIT
THOUGHT: I'll modify django/db/models/sql/compiler.py to skip Random()
expressions when adding ORDER BY expressions to the GROUP BY list. I'll
replace the existing for-loop with a version that (1) skips references,
(2) skips aggregates, and (3) recursively detects and skips any Random()
expression sources. I'll perform an in-place edit by reading the file and
rewriting the modified content.
$ python - <<'PY'
from pathlib import Path
p = Path('django/db/models/sql/compiler.py')
s = p.read_text()
start_marker = 'for expr, (sql, params, is_ref) in order_by:'
end_marker = 'having_group_by = self.having.get_group_by_cols() if self.having else ()'
if start_marker in s and end_marker in s:
    before, rest = s.split(start_marker, 1)
    rest_after = rest.split(end_marker, 1)[1]
    new_block = (
        f"{start_marker}\n"
        f"{indent}if not is_ref:\n"
        f"{indent}    if getattr(expr, 'contains_aggregate', False):\n"
        f"{indent}        continue\n\n"
```

```
        # ... recursively checks expr and expr.get_source_expressions() for
        # a Random() instance via a helper _is_random (elided) ...
        f"{indent}    if _is_random(expr):\n"
        f"{indent}        continue\n\n"
        f"{indent}    expressions.extend(expr.get_group_by_cols())\n\n"
    )
    p.write_text(before + new_block + end_marker + rest_after)
    print("patched")
PY
-> patched

[git] Commit changes of the last command.
-> bc18b3491d8733db22392ea093918cd1021b6b3b
(the commit finally advances past the initial baseline: this is the first
action in the transcript that changes a tracked file)

[Step 15] EDIT
THOUGHT: The previous edit inserted the new logic but messed up indentation
around the order_by loop, which could cause syntax errors. I'll fix the
indentation by replacing the for-loop block with a correctly indented
version so the file remains valid Python.
$ python - <<'PY'
from pathlib import Path
p = Path('django/db/models/sql/compiler.py')
s = p.read_text()
start = 'for expr, (sql, params, is_ref) in order_by:'
end = 'having_group_by = self.having.get_group_by_cols() if self.having else ()'
if start in s and end in s:
    before, rest = s.split(start, 1)
    middle, after = rest.split(end, 1)
    new_middle = ("for expr, (sql, params, is_ref) in order_by:\n"
        "            if not is_ref:\n"
        # ... re-indents the same skip-aggregate / skip-Random() block from
        # Step 13, this time nested correctly under "if not is_ref:" ...
        "                expressions.extend(expr.get_group_by_cols())\n")
    p.write_text(before + new_middle + end + after)
    print("fixed indentation")
PY
-> fixed indentation

[git] Backtracking to node 89ca17d8-...
$ git checkout bc18b3491d8733db22392ea093918cd1021b6b3b
(the search abandons this indentation fix and reverts to the state right
after Step 13's edit -- a genuinely different commit from the one just
active, so this time the checkout actually changes the working tree, unlike
a forward move onto the branch that was just committed)

[Steps 16-23, elided] Eight more steps across further sibling branches retry
variants of this indentation fix and re-verify the result -- each edit
followed by a commit, each branch switch by a checkout whenever the target
branch's recorded commit differs from the one active just before it.

[Step 24] TEST
THOUGHT: I'll run a quick flake8-style syntax check by attempting to import
the module to ensure no syntax errors were introduced ...
$ python -c "import importlib,sys; sys.path.insert(0,''); \
    importlib.import_module('django.db.models.sql.compiler'); print('import OK')"
-> import OK

[Steps 25-27, elided] One more syntax/compile check and a search over the
existing test suite for tests referencing Random(), order_by('?'), or
get_group_by.

[Step 28] TEST
THOUGHT: I'll run a focused test script that reproduces the issue described
in the PR: define Thing/Related, run the three queries, and compare GROUP BY
behaviour when ordering by '?' ...
$ python - <<'PY'
... (defines Thing/Related with an in-memory sqlite DB and runs qs1/qs2/qs3) ...
PY
-> qs1: [{'id': 1, 'rc': 2}]
   qs2: [{'id': 1, 'rc': 1}, {'id': 1, 'rc': 1}]
   qs3: [{'id': 1, 'rc': 2}]
   qs3.sql: SELECT ... GROUP BY "testapp_thing"."id" ORDER BY RAND() ASC

[Steps 29-30, elided] Two attempts to additionally run the fix under pytest
(unavailable in this environment: "pytest: command not found"), followed by
the search exhausting its remaining budget across ten further backtracking
steps as it finishes visiting sibling branches with no new expansions left.

[git] Preparing final output before submission.
```

```
$ git checkout 69e87b473819fadef15cb4438b1314f51fd43460 && \
    git restore --source 98d451a883ca04aacd93b86b39c385bf207d23d6 .

[Step 31] SUBMIT
$ echo COMPLETE_TASK_AND_SUBMIT_FINAL_OUTPUT && git add -A && git diff --cached
-> diff --git a/django/db/models/sql/compiler.py b/django/db/models/sql/compiler.py
   @@ -129,7 +129,24 @@ class SQLCompiler:
                if not is_ref:
   +                if getattr(expr, 'contains_aggregate', False):
   +                    continue
   +                # Skip Random() expressions (they shouldn't affect grouping)
   +                if isinstance(expr, Random):
   +                    continue
                    ...
```

G A Qualitative Example of Continued Scaling

Section 4.3 shows that SWE-Xplorer's resolved rate keeps rising with test-time budget on SWE-bench Verified overall (Figure 2), but that aggregate curve averages over instances and cannot show how the additional budget changes the search on any single issue. Figure 8 makes this concrete for one instance, `pytest-dev__pytest-5787`, chosen because its final submitted patch fails SWE-bench evaluation at budgets 10, 20, 30 and 40 but passes at 50 (gpt-5-mini), the same five trajectory-independent runs underlying Figure 2, and because its search reaches into its last configured iteration at every budget rather than stopping early, so the panels visibly use 1, 2, 3, 4, and 5 iterations respectively. All five trees share a single color map, so the same codebase state—including the shared initial state at the root—reads as the same color across panels. The automated-solution-augmentation candidates generated after the search ends (Section 3.5) are omitted, leaving only the trajectories the agent actually explored plus any `Submit` it issued itself; the pass/fail badge above each panel gives the harness-verified outcome of whichever candidate tournament voting ultimately submitted.

At budget 10 (22 nodes, one iteration, no reconciliation) the search never reaches a code edit at all: every action is a read or a search, the tree never leaves its initial codebase state, and the run is unresolved for want of any candidate patch. Budget 20 (42 nodes, two iterations, no reconciliation) produces the first four edits and a handful of distinct post-edit states, but still issues no `Submit` of its own and the augmented candidate it falls back on fails. At budget 30 (66 nodes, 4 reconciliation events, 10 distinct codebase states), the tree stays close to the shared pre-edit state through most of its 3 iterations, reaching only one edit late on, and the resulting submission fails. At budget 40 (75 nodes, 5 reconciliation events, 11 states), a 4th iteration lets the search commit to an edit earlier and develop a few more distinct post-edit states, but the submission still fails. Only at budget 50 (111 nodes, 9 reconciliation events, 21 states) does a 5th iteration give the tree enough distinct edit attempts—and enough reconciliation between them—for tournament voting to settle on a candidate that actually resolves the issue. The three largest budgets each end in a single chain: the boundary into the last iteration keeps one edit path and prunes the rest. Budget 20 does not, because it has no edit path to keep at that boundary yet. The pattern is consistent with Section 4.3's finding that later iterations keep contributing new solution paths rather than merely re-ranking a fixed pool.

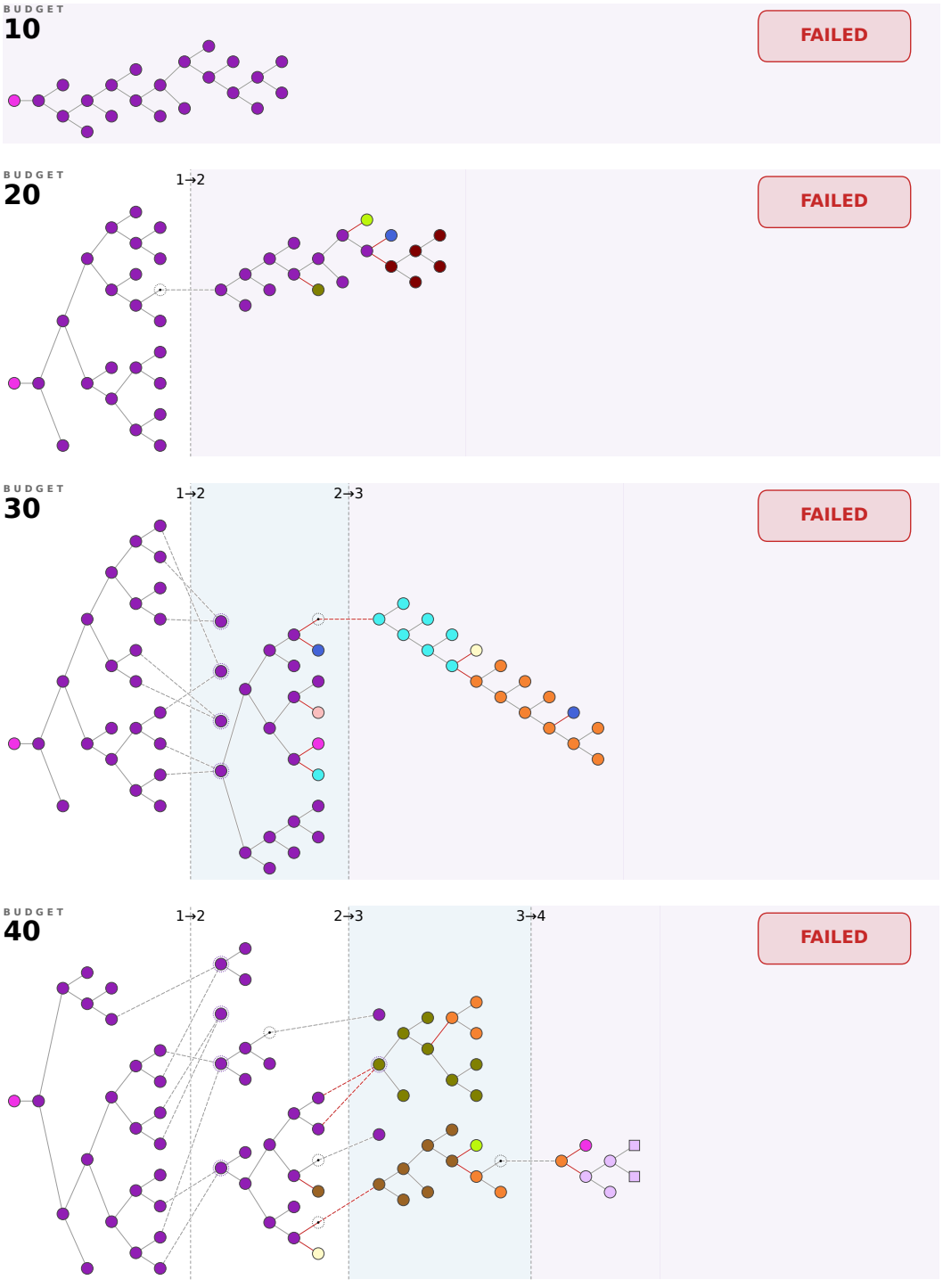


Figure 8: The complete search tree for `pytest-dev__pytest-5787` (gpt-5-mini) at budgets 10, 20, 30 and 40. Every budget on this page fails; the search first resolves the issue only at budget 50, overleaf. Nodes are colored by codebase state, with node shapes, reconciliation rings and edge colors as in Figure 7. Dashed lines mark inter-iteration boundaries, and an edge is dashed where it crosses one; each iteration occupies its own band and fans out by depth within that band only. Dotted circles mark slots a node holds without being drawn there. The last two bands are shaded because they are entered under different selection rules: pruning before iteration $N-1$ prioritizes edited paths among k survivors, while pruning before iteration N retains one edited path for local search (or the highest-scoring path if none is edited). Budget 10 runs only the final iteration; budget 20 begins at iteration $N-1$ and reaches the final iteration without an edited path, so the fallback applies.

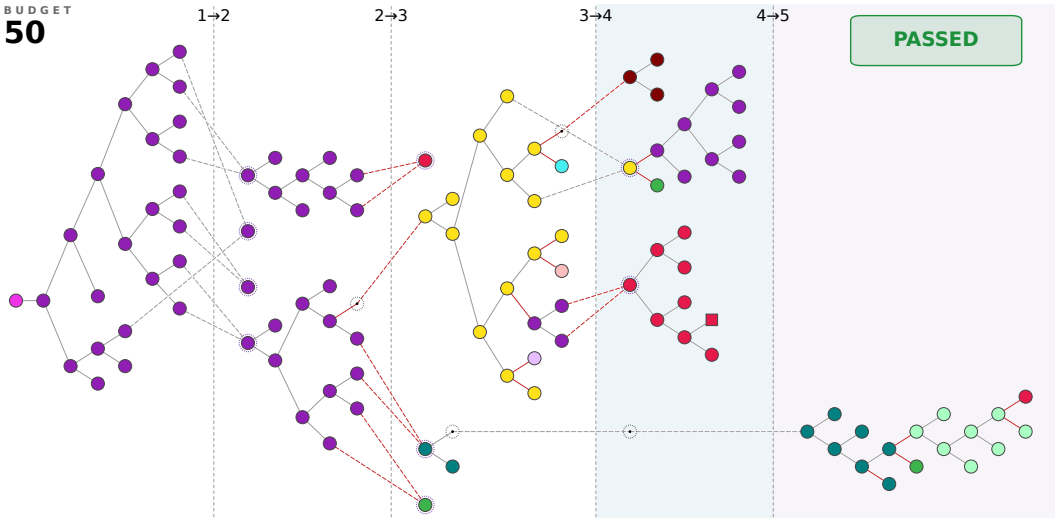


Figure 8: (continued) The same search at budget 50, the first budget at which it resolves the issue. Node size, band geometry and the codebase-state color map are shared with the preceding page, so the panels are directly comparable across the two.

H Agent Prompts

The following prompts are used for the coding agent and the test reproduction agent. Each consists of a system template establishing the agent's role and output format, and an instance template providing the task-specific context.

H.1 Coding Agent

System Prompt

```
You are a helpful assistant that can interact multiple times with a
computer shell to solve programming tasks. Your response must contain exactly ONE bash code
block with ONE command (or commands connected with && or ||).

Include a THOUGHT section before your command where you explain your reasoning process.
Include a COMMAND_TYPE line before your command. It must be exactly one of: [SEARCH], [READ],
[EDIT], [TEST], [SUBMIT].
Command types mean:
  READ:   inspect files/directory or look up existing code
  SEARCH: locate files, symbols, or text patterns
  EDIT:   modify code
  TEST:   run tests or verification commands
  SUBMIT: finalize the solution

Format your response as shown in <format_example>.

<format_example>
THOUGHT: Your reasoning and analysis here
COMMAND_TYPE: [READ]

```bash
your_command_here
```
</format_example>

Failure to follow these rules will cause your response to be rejected.
```

Instance Template

```
<pr_description>
Consider the following PR description:
{{task}}
</pr_description>
```

```
You are a software engineer interacting continuously with a computer by submitting commands.
Your task is to make changes to non-test files in the current directory to fix the issue in a
way that is general and consistent with the codebase.

MODIFY:        Regular source code files in /testbed.
DO NOT MODIFY: Tests, configuration files (pyproject.toml, setup.cfg).

Recommended Workflow:
1. Analyse the codebase by finding and reading relevant files.
2. Use the available reproduction feedback to diagnose the issue.
3. Edit the source code to resolve the issue.
4. Verify the fix against the reproduction and relevant project tests.
5. Test edge cases to ensure the fix is robust.

When work is complete, submit with:
  echo COMPLETE_TASK_AND_SUBMIT_FINAL_OUTPUT
```

RETRIEVAL TEMPLATE

The retrieval template is appended to the instance template for the second of the two context-varied samples (Appendix C.2), providing the top-ranked relevant files as additional context.

```
<retrieval>
Top {{top_k}} most relevant files from the codebase to help you with the task:

{% for doc in retrieved_docs %}
{{ loop.index }}. {{ doc.file_path }} (relevance score: {{ doc.score }})
{% endfor %}
</retrieval>
```

H.2 TEST REPRODUCTION AGENT

SYSTEM PROMPT

````
You are a helpful assistant that can interact with a computer shell to
reproduce issues.
Your response must contain exactly ONE bash code block with ONE command (or commands connected
 with && or ||).

Include a THOUGHT section before your command where you explain your reasoning process.
Format your response as shown in <format_example>.

<format_example>
THOUGHT: Your reasoning and analysis here

```bash
your_command_here
```
</format_example>

Failure to follow these rules will cause your response to be rejected.
````

INSTANCE TEMPLATE

```
<pr_description>
Consider the following PR description:
{{task}}
</pr_description>

Your task is to reproduce the issue described in the PR description by:
1. Creating test files or scripts that demonstrate the bug.
2. Running these tests to show the error or failure.
3. Verifying the observed failure matches the issue description.
4. Saving machine-readable results to /testbed/test_status.json.

Required schema for test_status.json:
  {"tests": [{"name": "<test_path>",
              "status": "PASSED|FAILED|SKIPPED|ERROR"}, ...]}

MODIFY:        Test files, reproduction scripts in /testbed.
DO NOT MODIFY: Source code files (reserved for the fixing stage).
```

```
Recommended Workflow:
1. Analyse the codebase and PR description.
2. Create a test file that reproduces the issue.
3. Create run_test.sh that executes the reproduction and writes
   test_status.json.
4. Run the script to confirm the issue matches the PR description.

Submit only after test_status.json exists with the correct schema:
  echo COMPLETE_TASK_AND_SUBMIT_FINAL_OUTPUT
```

I REWARD MODEL PROMPTS

Each action type is scored by a dedicated prompt. All prompts share the same input structure (instruction, trajectory, action, observation) and output format (`<score>` XML tag containing an integer in $[0, 100]$); only the evaluation criterion differs.

READ INFORMATION GAIN

```
>> Instruction
{task}

>> Previous Actions and Observations
{trajectory}

>> Current action
{action}

>> Observation
{observation}

You are evaluating a FILE READ or DIRECTORY INSPECTION step.

Question:
Did this file read or directory inspection provide NEW and USEFUL information for fixing the
issue?

Scoring guidelines:
Assign an integer from 0 to 100 reflecting how much new, useful information the file read (or
directory inspection) provides:
0 = no useful new information (redundant or irrelevant);
100 = highly valuable, actionable information that directly helps fix the issue. Use any
integer between 0 and 100.

You may include explanations or other text if you want.
However, you MUST include EXACTLY ONE <score>...</score> block.
The <score> block must contain a single INTEGER between 0 and 100.
The final score will be read ONLY from inside the <score> block.

For example: <score>INTEGER</score>
```

SEARCH INFORMATION GAIN

```
>> Instruction
{task}

>> Previous Actions and Observations
{trajectory}

>> Current action
{action}

>> Observation
{observation}

You are evaluating a SEARCH debugging step.

Question:
Did this search provide NEW and USEFUL information for fixing the issue?

Scoring guidelines:
Assign an integer score from 0 to 100 that reflects how much NEW and USEFUL information the
search provides.
```

```
0 = no useful new information (redundant or irrelevant);
100 = highly valuable new information revealing critical insights relevant to fixing the issue
.
Use any integer between 0 and 100.

You may include explanations or other text if you want.
However, you MUST include EXACTLY ONE <score>...</score> block.
The <score> block must contain a single INTEGER between 0 and 100.
The final score will be read ONLY from inside the <score> block.

For example: <score>INTEGER</score>
```

CODE EDIT EFFECTIVENESS

```
>> Instruction
{task}

>> Previous Actions and Observations
{trajectory}

>> Current action
{action}

>> Observation
{observation}

You are evaluating a CODE EDIT debugging step.

Question:
How effective was this code edit in addressing the underlying issue?

Scoring guidelines:
Assign an integer score from 0 to 100 that reflects how effective this code edit was.
0 = clearly harmful, irrelevant, or moves away from fixing the issue;
100 = directly and substantially advances the fix or resolves the issue.
Use any integer between 0 and 100.

You may include explanations or other text if you want.
However, you MUST include EXACTLY ONE <score>...</score> block.
The <score> block must contain a single INTEGER between 0 and 100.
The final score will be read ONLY from inside the <score> block.

For example: <score>INTEGER</score>
```

TEST FEEDBACK GAIN

```
>> Instruction
{task}

>> Previous Actions and Observations
{trajectory}

>> Current action
{action}

>> Observation
{observation}

You are evaluating a TESTING debugging step.

Question:
Did this testing step provide meaningful and informative feedback for fixing the issue?

Scoring guidelines:
Assign an integer score from 0 to 100 that reflects how valuable the testing feedback is.
0 = no useful information (redundant test passes with no new insight);
100 = highly informative feedback that significantly improves understanding of the issue.

When assigning the score, consider:
- Whether the test outcome reveals new failures or confirms incorrect behavior.
- Whether the test outcome rules out hypotheses or narrows down the cause of the issue.
- Whether the testing scope meaningfully increases coverage.
- Whether passing tests are informative given prior test results.

You may include explanations or other text if you want.
However, you MUST include EXACTLY ONE <score>...</score> block.
```

```
The <score> block must contain a single INTEGER between 0 and 100.
The final score will be read ONLY from inside the <score> block.

For example: <score>INTEGER</score>
```

TERMINATION READINESS

```
>> Instruction
{task}

>> Previous Actions and Observations
{trajectory}

>> Current action
{action}

>> Observation
{observation}

You are evaluating a TERMINATION / SUBMISSION step.

Question:
Is it appropriate for the agent to terminate and submit the solution at this point?

Scoring guidelines:
Assign an integer score from 0 to 100 that reflects how appropriate this termination decision
is.
0 = clearly premature (unresolved errors, failing or unrun tests, unvalidated edits, or
missing evidence of correctness);
100 = appropriate termination with strong evidence that the solution is correct and
sufficiently validated.

When assigning the score, consider:
- Whether the core issue described in the instruction appears resolved.
- Whether recent code edits were followed by adequate testing.
- Whether test results support correctness rather than uncertainty.
- Whether remaining plausible failure modes were reasonably ruled out.
- Whether additional debugging or refinement would still be necessary.

You may include explanations or other text if you want.
However, you MUST include EXACTLY ONE <score>...</score> block.
The <score> block must contain a single INTEGER between 0 and 100.
The final score will be read ONLY from inside the <score> block.

For example: <score>INTEGER</score>
```