DPIAgent: Divide, Protocol, Isolate for Agentic Reproduction Test Generation

Hao Liu[1*†] **Steven Liu**[1*†] **Xin Zhang**[1*‡] **Jane Luo**[1†]
Yu Kang[1] **Jie Wu**[1,2†] **Fangkai Yang**[1] **Yangyu Huang**[1]
Pengfei Gao[1] **Scarlett Li**[1] **Yan Lu**[1]
[1]Microsoft [2]Tsinghua University
{haoliu.ai, lutgasx}@outlook.com xinzhang3@microsoft.com

Project page: https://neverwinhao.github.io/DPIAgent/

Abstract

Reproduction test generation, producing a failing-then-passing test that captures a reported bug, is a critical step in automated software engineering. Existing agentic methods treat this as a monolithic loop, despite the task inherently comprising two subtasks of distinct nature: diagnosing the root cause and writing a fail-to-pass test. Without explicit separation, the agent faces a compound objective with underspecified intermediate goals, leading to goal drift. We propose DPIAgent, a structured agentic framework built on three principles, Divide, Protocol, Isolate (DPI), that mitigates compound-objective ambiguity and goal drift: it Divides the task into single-objective phases of defect exploration and test generation; enforces a handoff Protocol that records the diagnosis and test plan, preventing context loss; and Isolates each phase's action space by tailoring the toolset to its task, preventing irrelevant tools from misleading execution. On SWT-Bench Verified, DPIAgent outperforms seven baselines across three backbone LLMs. With DPI alone it reaches 81.76% success rate on GPT-5, the highest reported among open-source methods, gaining up to 11.88 points over the strongest baseline on GPT-5-Mini; with test selection, it reached 86.17% in our evaluation and ranked **first** on the official SWT-Bench Verified leaderboard. Our analysis shows that architectural structure and backbone capability are complementary axes rather than substitutes, demonstrating DPI's generalizability across model classes.

1 Introduction

Large Language Models (LLMs) [OpenAI, 2024, Yang et al., 2025, Comanici et al., 2025] have demonstrated strong capabilities across various domains and are increasingly applied to code generation [Chen et al., 2021, Hui et al., 2024, Luo et al., 2026a] and software engineering automation. A particularly important yet underexplored task is reproduction test generation: given a bug report, the goal is to produce a test that fails on the buggy code and passes after the fix. Despite its central role in the software development lifecycle, reproduction tests are rarely available in the test suite when an issue is first reported [Kang et al., 2023, Mündler et al., 2024, Liu et al., 2026]. And developers widely consider writing tests a tedious and time-consuming task [Straubinger and Fraser, 2023].

Existing approaches to reproduction test generation fall into two paradigms. The first follows a **workflow design** [Ahmed et al., 2025, 2026, Nashid et al., 2026], where the task is decomposed into fixed pipeline stages such as localization, planning, and generation, with predefined control

[*]Equal contribution. [‡]Corresponding author.
[†]Work done during internships at Microsoft Research Asia.

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

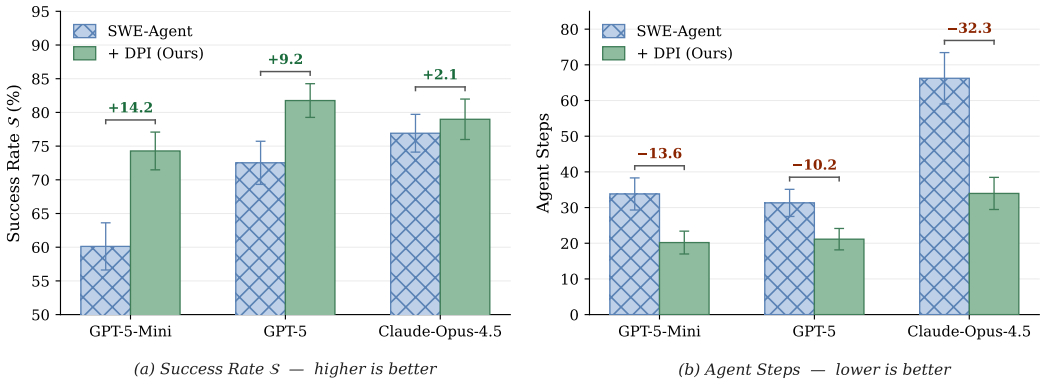


Figure 1: **Isolating DPI's contribution.** Adding DPI to SWE-Agent yields consistent gains in success rate and reduces agent steps across three backbones. GPT-5-Mini+DPI surpasses GPT-5 SWE-Agent, indicating structural intervention can compensate for model scale.

flow orchestrating LLM calls at each stage. The second adopts an **agent paradigm** [Yang et al., 2024, Wang et al., 2025a, Team et al., 2025], where the LLM is equipped with tools and dynamically decides which actions to take and when to terminate, without hardcoded stage transitions. The agent paradigm offers key advantages: it allows the model to pivot based on runtime observations and maintains reasoning continuity throughout the task, achieving strong performance on repository-level benchmarks [Jimenez et al., 2024, Mündler et al., 2024, Zhang et al., 2026]. Recent leading systems for repository-scale code tasks are predominantly agent-based; we therefore build our method.

However, while the agent paradigm offers flexibility, existing approaches treat reproduction test generation as a monolithic task, mixing exploration and test writing in an undifferentiated loop. In reality, this task inherently comprises two subtasks of distinct nature: understanding the root cause through code exploration, and producing a test satisfying the fail-to-pass criterion. Without explicit separation, the agent faces a compound objective with underspecified intermediate goals, which has been shown to systematically induce goal drift [Arike et al., 2025] and is identified as a primary failure mode for software engineering agents [Vijayvargiya et al., 2026]. In practice, developers naturally follow a two-stage process of first localizing the root cause and then writing the test [Cheng et al., 2025], suggesting that explicit decomposition aligns with the inherent structure of the task. Moreover, developers locate relevant code through their understanding of code structure and dependencies, whereas existing agents lack such awareness and rely on basic text-matching tools, demanding extensive back-and-forth in large repositories.

Motivated by these observations, we propose DPIAgent, an agent design that imposes structure *only at phase boundaries* while preserving a full agent loop within each phase. DPIAgent follows three principles—**D**ivide, **P**rotocol, **I**solate (**DPI**). **(D)ivide:** DPIAgent replaces the monolithic loop with two sequential phases, defect exploration and test generation. Each phase has a single, well-defined objective, eliminating the compound-objective ambiguity of monolithic agents. **(P)rotocol:** Agents cannot transition silently: it must explicitly write down its diagnosis and test plan before entering the next phase. This forces an explicit handoff so no context is lost between exploration and test writing. **(I)solate:** Agents often struggle with tool distraction when exposed to an overly broad action space. To maintain focus, we tailor each phase's toolset strictly to its task, preventing irrelevant tools from misleading the agent's execution. In addition, we adopt a test selection step from e-Otter++ [Ahmed et al., 2026] that turns variance across diversified candidate runs into improved final output quality.

We evaluate DPIAgent on SWT-Bench Verified across three backbone LLMs (GPT-5-Mini [Singh et al., 2026], GPT-5 [Singh et al., 2026], and Claude-Opus-4.5[1]), comparing against seven baselines spanning diverse agent architectures. Results show that DPIAgent consistently outperforms all baselines across all backbones, reaching 81.76% with DPI alone on GPT-5 (the highest among open-source methods on this benchmark) and 86.17% with test selection. Isolating DPI's effect from confounds in baseline architectures, we observe consistent gains across all backbones in both pass rate and step efficiency (Fig. 1). These gains diminish but persist as model capability grows, indicating that structural design and model scale are complementary axes rather than substitutes.

[1] `https://www.anthropic.com/news/claude-opus-4-5`

Ablation studies verify the effectiveness of DPI's design components. First-edit quality analysis shows that DPI consistently improves pre-feedback test readiness across all backbones, especially benefiting weaker models that rarely recover from incomplete initial test designs.

Our contributions can be summarized as follows:

- We propose DPIAgent, a structured agentic framework that mitigates compound-objective ambiguity and goal drift in composite tasks via the **DPI** principles. It divides monolithic tasks into single-objective phases, enforces a structured handoff protocol to preserve context, and isolates phase-specific toolsets to prevent action-space distraction.
- On SWT-Bench Verified, DPIAgent outperforms seven baselines across three backbone LLMs. With DPI alone, DPIAgent reaches **81.76%** success rate on GPT-5, the highest reported among open-source methods, and gains up to 11.88 points over the strongest baseline on GPT-5-Mini. Adding test selection further raises the result to 86.17% on GPT-5.
- Our analysis reveals that **architectural structure and backbone capability are complementary axes rather than substitutes**. DPIAgent yields the largest absolute gain on the weakest model, 11.88 points on GPT-5-Mini, demonstrating DPI's generalizability across model classes. It further reaches peak pass rates in 3$\times$ fewer steps, uniquely resolves 18+ issues, and improves function-level localization by 25+ Acc@5 points.

2 Related Work

Workflow-Based Test Generation One line of work decomposes test generation into fixed pipeline stages with predefined control flow. Early methods focus on post-hoc selection or assertion manipulation without explicit localization [Kang et al., 2023, Khatib et al., 2026]. Subsequent work introduces localization as a prerequisite: Issue2Test [Nashid et al., 2026] and Auto-TDD [Ahmed et al., 2024] perform root-cause analysis before generation, SWE-Tester [Soni et al., 2026] trains open-source LLMs with BM25-based retrieval, and the Otter series [Ahmed et al., 2025, 2026] progressively enriches the pipeline with self-reflective planning, execution-augmented repair, and surrogate patch-based selection. While these methods benefit from explicit stage structure, they rely on rigid control flow that cannot adapt to runtime observations.

Agent-Based Test Generation A second line applies general-purpose LLM agents that dynamically decide actions based on observations. Representative systems include SWE-Agent+ [Mündler et al., 2024], OpenHands [Wang et al., 2025a], and TraeAgent [Team et al., 2025], which differ in tool design and candidate selection but all treat test generation as a monolithic task, mixing exploration and writing in a single undifferentiated loop. AEGIS [Wang et al., 2025b] introduces a two-agent design where a Searcher retrieves context and a Reproducer generates tests, but relies on multi-agent coordination that risks information loss across boundaries. Our work occupies a distinct point: we retain a full agent loop *within* each phase while enforcing DPI structure *at phase boundaries*, combining agent adaptability with pipeline focus without multi-agent overhead.

3 Methodology

This section details DPIAgent as the instantiation of our DPI principles. §3.1 formulates the problem and overviews the design. Following the **Divide** principle, §3.2 and §3.3 detail the two phases, defect exploration and test generation. **Protocol** governs the handoff at their boundary, and **Isolate** constrains each phase's action space to its specific task. §3.4 introduces the test selection mechanism.

3.1 Overview

Problem setup Given a codebase C and a natural-language-described software issue I, the goal of an issue reproduction approach is to automatically generate a test suite $T_{\text{pred}} = \{t_1, t_2, ..., t_M\}$ that reliably reproduces I. A generated test t_i is considered to reproduce the issue if it fails on the original buggy codebase C but passes on the fixed codebase C' obtained by applying the golden patch. The generated suite T_{pred} is correct if one t_i reproduces the issue and others show pass-to-pass behavior.

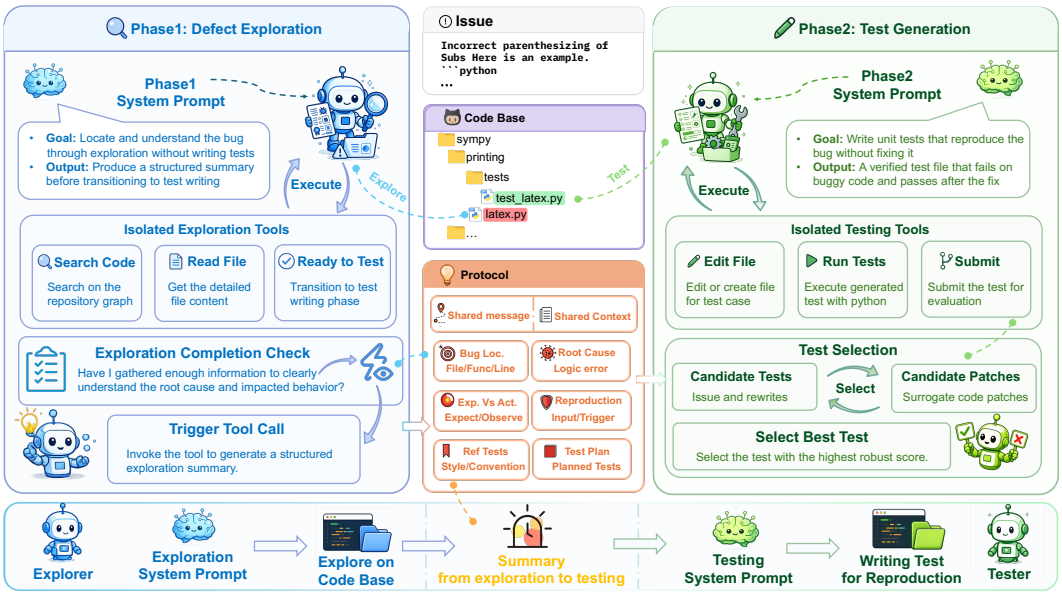


Figure 2: Overview of DPIAgent. Following DPI, the pipeline has three stages: **Divide** splits exploration from test generation; **Protocol** passes diagnostic findings through a structured summary; and **Isolate** switches tools across the boundary while preserving context. A final **Test Selection** stage scores candidates against surrogate patches.

Motivation While agents generally achieve higher performance than traditional pipelines due to their operational flexibility [Yang et al., 2026], this uncontrolled freedom becomes a liability in composite tasks like writing reproduction tests. In such unconstrained environments, agents often lose track of their current stage and overall objective, leading to execution confusion and significant fluctuations in the generated results. To bridge this gap, DPIAgent translates the DPI principles (§1) into a concrete architectural design. By Dividing the monolithic loop, we assign these two subtasks to distinct phases: defect exploration (§3.2) and test generation (§3.3). To ensure critical diagnosis context is not lost during the transition, an explicit handoff Protocol bridges the phase boundary. To maintain focus, we Isolate the phases by tailoring the toolset to each phase's specific task, preventing irrelevant tools from misleading the agent's execution. We also use e-Otter++ test selection [Ahmed et al., 2026] (§3.4) to improve outputs via run variance.

3.2 Defect Exploration

In the Defect Exploration phase, the agent commits to building a complete diagnosis of the bug before any test is written. The agent is equipped with file-reading tools such as `view` and graph-based search tools that operate over a pre-constructed code graph Luo et al. [2026b] capturing both code dependencies and semantic relations. The graph enables the agent to retrieve bug-relevant code through structural relationships rather than text patterns, which is faster and more precise.

After exploration, DPIAgent enforces the Protocol principle at the phase boundary: the agent must invoke `ready_to_write_test` to produce a structured handoff summary before transitioning, preventing context loss. The summary captures the agent's diagnosis and a concrete test plan. The diagnosis spans five fields: (1) bug location, with exact file paths and function names; (2) root cause, identifying the specific logic error; (3) expected versus actual behavior; (4) reproduction strategy, describing the inputs or conditions that trigger the bug; (5) reference tests, noting existing test files examined for conventions. The test plan (6) outlines which test file to create and what assertions to make. Together, the diagnosis and test plan constitute the agent's complete brief for the test writing phase, ensuring it begins from a finalized understanding rather than partial fragments.

3.3 Test Generation

Upon transition, the agent enters the Test Generation phase with the full exploration trajectory preserved in context. The exploration tools are entirely removed from the action space and replaced

with `edit` for modifying test files and `bash` for executing tests; the system prompt is correspondingly reoriented. This realizes the Isolate principle: with the agent's tools aligned to the test-writing task, irrelevant tools cannot mislead execution, keeping the agent's full Phase 2 effort dedicated to test construction. Combined with the exploration summary as a guideline, the agent focuses on crafting and verifying reproduction tests without repeating prior investigation.

3.4 Test Selection

On top of DPI's structural guarantees, individual pipeline runs naturally vary in candidate quality. We adopt a test selection step from e-Otter++ [Ahmed et al., 2026] that deliberately diversifies inputs and selects across the resulting candidate pool, turning this variance into improved final output quality.

Following e-Otter++ [Ahmed et al., 2026], we generate five stylistically distinct rewrites of each issue and run DPIAgent on each, yielding a candidate set $\mathcal{T}$ of six test patches. We use SWE-Agent [Yang et al., 2024] to generate three surrogate patches $\mathcal{B}$, proxying the unavailable golden patch.

Selection proceeds in two steps. First, we **filter** $\mathcal{T}$ to retain only candidates that exhibit fail-to-pass (F2P) behavior on at least one $b_i \in \mathcal{B}$ (if none qualifies, all candidates are kept as fallback). Second, we **rank** the surviving candidates by the number of surrogate patches under which F2P holds, with average coverage as a tiebreak:

$$S_j = \big|\{b_i \in \mathcal{B} : t_j \text{ is F2P under } b_i\}\big|, \qquad \mathrm{cov}_j = \frac{1}{|\mathcal{B}|}\sum_i \mathrm{Coverage}(t_j, C \oplus b_i). \tag{1}$$

The top-ranked candidate is returned. The intuition is that a test that triggers the bug under more diverse fixes is more likely to capture the true buggy behavior than to overfit a specific patch.

4 Experiments Setup

Benchmark We evaluate DPIAgent on SWT-Bench [Mündler et al., 2024], a benchmark constructed from SWE-Bench [Jimenez et al., 2024] that organizes GitHub issues together with their corresponding fixes and test cases. SWT-Bench filters out instances with unreliable golden patches due to flaky tests and offers two subsets: SWT-Bench Lite (276 instances) and SWT-Bench Verified (433 instances). Consistent with prior work, we use SWT-Bench Verified as our primary evaluation dataset, which provides a larger set of validated instances and reliable comparisons across models.

Baselines We compare against representative approaches for test generation: SWE-Agent [Yang et al., 2024] equips the LLM with direct access to a limited shell environment and provides specialized tools for file searching, viewing, and editing; Mini-SWE-Agent [Yang et al., 2024] is a lightweight variant that simplifies the tool interface while maintaining core exploration capabilities; OpenHands [Wang et al., 2025a] is a generalist agent that interacts with the repository via bash commands and Python calls to a predefined skill library within a stateful IPython session; TraeAgent [Team et al., 2025] ensembles multiple candidate patches and selects the final one via pruning and a voting-based selector agent. Aider[2] performs repository indexing to guide file selection, incorporates model-generated summaries to augment context, and validates edits via static analysis and repository test cases before application. Terminus-2 [Merrill et al., 2026] is a minimal scaffold that interacts with the codebase purely through Bash commands in a headless terminal, designed as a neutral testbed without specialized tools. Claude Code[3] is Anthropic's official command-line coding agent that operates through a curated set of file and shell tools. Together, these baselines span a diverse range of agent architectures, from minimal terminal scaffolds to specialized SWE agents and ensemble-based systems, providing a comprehensive comparison across the design space of LLM-based test generation. Implementation details for each baseline are provided in Appendix C.

Evaluation Metrics We adopt three standard metrics following SWT-Bench [Mündler et al., 2024]: Success rate ($\mathcal{S}$) measures the fraction of instances for which the generated tests successfully reproduce the target issue; The Mean coverage ($\mathbf{Cov}.$) reports the average fraction of the golden patch's executable modified lines covered by the generated tests. The Change coverage ($\mathbf{\Delta C}$)

[2] `https://github.com/Aider-AI/aider`
[3] `https://claude.ai/code`

Table 1: Performance results on SWT-Bench Verified. **Ours** applies DPI on top of SWE-Agent, **without test selection**.

Model	Method	$\mathcal{S}$. ↑	ΔC ↑	Cov. ↑	TDD ↑
GPT-4o	LIBRO†	17.80	38.00	–	–
	AssertFlip†	45.50	47.40	–	–
	Otter†	31.60	37.60	74.59	25.78
	Otter++†	37.40	42.80	79.47	34.04
Claude-Sonnet-3.7	e-Otter++†	62.10	62.30	85.73	56.80
v20250405-dev	Amazon Q†	51.00	57.40	70.04	42.72
Gemini-2.5-Pro	Echo†	66.30	68.50	–	–
GPT-5-Mini	TraeAgent	60.74	55.95	64.75	47.23
	Mini-SWE-Agent	38.80	41.54	53.11	35.02
	OpenHands†	62.40	60.60	69.92	42.13
	Claude Code	55.65	53.11	67.62	45.69
	Aider	40.42	41.33	47.62	30.62
	Terminus-2	42.73	50.39	60.21	35.04
	SWE-Agent	60.12	64.78	**80.12**	54.29
	Ours	**74.28**	**65.51**	75.97	**62.22**
GPT-5	TraeAgent	69.98	61.77	75.80	60.59
	Mini-SWE-Agent	54.27	44.80	54.69	41.94
	OpenHands†	79.80	66.30	68.85	45.79
	Claude Code	72.51	64.63	78.50	59.94
	Aider	34.87	39.33	46.84	26.59
	Terminus-2	44.11	48.14	58.46	34.23
	SWE-Agent	72.52	69.26	84.47	56.30
	Ours	**81.76**	**70.95**	**84.92**	**68.02**
Claude-Opus-4.5	TraeAgent	77.60	60.25	72.22	59.75
	Mini-SWE-Agent	76.67	67.91	82.37	66.85
	OpenHands	73.90	57.17	71.69	55.34
	Claude Code	73.21	59.48	70.70	55.35
	Aider	48.26	51.29	59.59	35.97
	Terminus-2	68.13	54.22	59.57	41.99
	SWE-Agent	76.91	71.62	86.64	69.77
	Ours	**78.98**	**71.74**	**89.52**	**73.77**

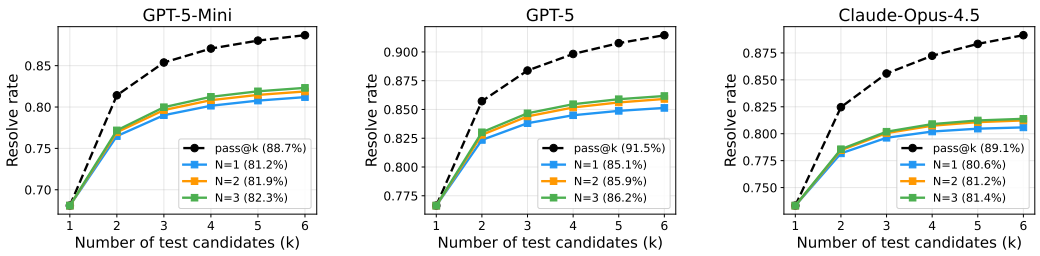
†Results reported from the public leaderboard.

further restricts this measurement to lines that are not already covered by the repository's original test suite, isolating the incremental coverage contributed by the generated tests. In addition, we follow [Ahmed et al., 2025] and include TDD coverage (**TDD**), which credits coverage only from instances satisfying $\mathcal{S}$, formally $\mathbf{TDD} = \frac{1}{N}\sum_{i:\mathcal{S}_i=1} Cov._i$, where N is the total number of instances. This metric discounts tests that touch the modified code without triggering the bug.

Implementation Details We evaluate DPIAgent with three backbone LLMs: GPT-5-Mini [Singh et al., 2026], GPT-5 [Singh et al., 2026], and Claude-Opus-4.5. Baselines follow their original configurations with all other parameters kept at default values. To evaluate the correctness of the generated tests, we adopt the official SWT-Bench Docker environment [Mündler et al., 2024] to ensure consistent and reliable execution. All results are averaged over 3 runs.

5 Main Results

DPIAgent Substantially Improves Issue Reproduction Table 1 demonstrates that our method consistently achieves the highest success rate ($\mathcal{S}$) across all three backbone models. On GPT-5-Mini, DPIAgent reaches 74.28% $\mathcal{S}$, outperforming the strongest baseline OpenHands by 11.88 points.



(a) GPT-5-Mini (b) GPT-5 (c) Claude-Opus-4.5

Figure 3: Resolve rate of test selection using N code patches compared with the pass@k oracle upper bound, as a function of the number of test candidates k. N denotes the number of code patches used to score and rank test candidates. The percentages in the legend denote the resolve rate at $k = 6$.

Although SWE-Agent reports a higher $Cov.$ of 80.12%, its TDD drops to 54.29%, well below our 62.22%, indicating that much of its coverage comes from tests that do not actually reproduce the issue. Furthermore, on GPT-5, DPIAgent elevates $\mathcal{S}$ to 81.76%, while simultaneously achieving the highest $Cov.$ and ΔC among all methods. Since DPIAgent is built on top of SWE-Agent, Figure 1 directly visualizes the gains from DPI: +14.16 $\mathcal{S}$ on GPT-5-Mini, +9.24 on GPT-5, and +2.07 on Claude-Opus-4.5. The TDD improvements (+7.93, +11.72, +4.00) are particularly notable, confirming that DPI helps the agent write tests that genuinely discriminate buggy from fixed behavior rather than merely touching modified lines. The gains are most pronounced on weaker backbones, suggesting that structured decomposition compensates for limited model capability.

DPI Improves Pre-feedback Test Readiness To characterize what phase structure contributes before execution feedback, we evaluate each trajectory's first test edit with an LLM-as-Judge. The judge scores two dimensions: *Target Alignment* (Target), whether the test targets the correct code location, and *Specification Completeness* (Spec), whether the test captures the trigger condition, critical input, call path, oracle, and discriminative behavior. We mark a first edit as *Ready* only when both dimensions receive the maximum score. We instantiate the judge with GPT-5.6-Terra, run it three times independently, apply outcome-aware correction by marking an edit Ready if it already achieves F2P without post-first-edit modification, and report mean case counts. Table 2 shows that DPI primarily improves pre-feedback test design quality, and this improvement is most critical for weaker models. DPI lifts Ready counts across all three backbones; since Ready cases convert to F2P at over 90% across all configurations, increasing Ready directly expands the solved set. Weaker models benefit more because they rarely recover from Not-Ready first edits, whereas stronger models can self-correct more often, so DPI's new Ready cases overlap with cases they could solve.

Table 2: Pre-feedback test design quality on SWT-Bench Verified. Entries are case counts; Δ rows are count differences.

Model	Config	Target	Spec	Ready	Ready F2P	Total F2P
GPT-5-Mini	SWE-Agent	319.7±0.6	269.7±1.2	269.7±1.2	243.7±1.5	273.7±1.5
	DPI	370.0±1.7	332.7±4.2	332.7±4.2	311.0±2.0	331.0±2.0
	Δ	**+50.7±1.2**	**+63.0±3.5**	**+63.0±3.5**	**+67.3±1.5**	**+57.7±1.2**
GPT-5	SWE-Agent	369.3±1.5	310.0±4.0	310.0±4.0	284.7±3.5	316.7±2.5
	DPI	405.0±1.7	377.7±1.5	377.7±1.5	352.7±0.6	355.7±0.6
	Δ	**+35.3±1.2**	**+67.7±4.7**	**+67.7±4.7**	**+68.0±3.6**	**+38.7±2.9**
Claude-Opus-4.5	SWE-Agent	365.0±0.0	294.7±0.6	294.7±0.6	269.7±1.5	332.7±0.6
	DPI	392.3±0.6	365.0±2.6	365.0±2.6	338.3±1.2	344.3±1.2
	Δ	**+27.3±0.6**	**+70.3±2.1**	**+70.3±2.1**	**+68.7±0.6**	**+11.7±1.5**

Test Selection Improves Reliability Figure 3 reports the success rate as a function of the number of candidate runs, comparing the oracle upper bound (Pass@k) with our surrogate patch-based selection. Selection consistently improves as N increases, confirming that additional surrogate patches provide

complementary signal for ranking candidates. Compared to our pipeline without selection (Table 1), adding test selection with k=6 and N=3 further improves success rate by 8.04 points on GPT-5-Mini (from 74.28% to 82.32%), 4.41 points on GPT-5 (from 81.76% to 86.17%), and 2.40 points on Claude-Opus-4.5 (from 78.98% to 81.38%). We find that weaker models benefit significantly more from selection than stronger ones. Although Claude-Opus-4.5 has a high Pass@k ceiling (89.15%), its candidates exhibit subtler behavioral differences that make surrogate patch-based scoring less effective at identifying the best one. We provide a detailed ablation of selection signals in Appendix D.

6 Ablation Study

Experimental Setup To isolate the contribution of each design choice, we run component ablations on SWT-Bench Verified with GPT-5-Mini and Claude-Opus-4.5. Each ablation removes one or more DPI principles. For framework ablation (Table 3), we remove the structured handoff summary (*w/o Prot.*, removing Protocol), expose all tools across phases (*w/o Isol.*, removing Isolate), and collapse the two-stage design into a single reactive loop (*w/o Divd.*, removing Divide) to assess structured planning, phase-specific tool gating, and stage decoupling.

Table 3: Ablation study of DPI principles on SWT-Bench. w/o Prot. removes the structured handoff summary, w/o Isol. exposes all tools across phases, and w/o Divd. collapses the two-stage design into a single reactive loop.

Metric	GPT-5-Mini				Claude-Opus-4.5			
	Full	**w/o Prot.**	**w/o Isol.**	**w/o Divd.**	**Full**	**w/o Prot.**	**w/o Isol.**	**w/o Divd.**
$\mathcal{S}$. ↑	**74.28**	70.90	72.70	67.66	**78.98**	75.51	76.21	69.05
ΔC ↑	**65.51**	63.71	59.40	54.59	**71.74**	65.98	64.03	61.03
Cov. ↑	**75.97**	72.88	67.80	64.28	**89.52**	79.27	79.50	75.96

DPI Components Play Complementary Roles Table 3 reveals the distinct contributions of each framework component. The Protocol component (structured handoff summary) transforms the agent's multi-turn exploration into a detailed plan that guides downstream test generation. Removing it consistently degrades performance, with $\mathcal{S}$ dropping from 78.98% to 75.51% on Claude-Opus-4.5. Isolate primarily affects test quality: exposing all tools across phases reduces ΔC by 6.11 points and $Cov.$ by 8.17 points on GPT-5-Mini, and by 7.71 and 10.02 points on Claude-Opus-4.5. Divide separates repository understanding from test generation, allowing each phase to focus on a single objective. Collapsing it into a single reactive loop causes substantial drops across all metrics, with $\mathcal{S}$ falling by 9.93 points and $Cov.$ dropping by 13.56 points on Claude-Opus-4.5. The full pipeline integrates all three principles to consistently outperform all ablated variants across both backbones.

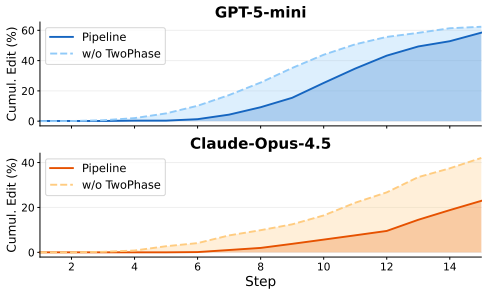


Figure 4: Edit tool usage rate across steps for Pipeline and w/o Divd. variant.

Table 4: Attribution study on GPT-5-Mini. Tool Contribution rows add individual mechanisms to monolithic SWE-Agent; + DPI applies DPI phase structure with standard SWE-Agent tools and no Search Code.

Type	**Config.**	$\mathcal{S}$. ↑	**TDD** ↑	$\Delta\mathcal{S}$.	**ΔTDD**
Baseline	SWE-Agent	60.12	54.29	–	–
Tool Contribution	+ Search Code	67.66	50.40	+7.54	-3.89
	+ Summary Prompt	54.96	42.44	-5.16	-11.85
	+ Transition Tool	61.20	48.68	+1.08	-5.61
DPI Contribution	+ DPI	**72.97**	**56.41**	**+12.85**	**+2.12**

DPI Gains Come from Phase Structure Rather Than Tool Changes To disentangle architectural structure from implementation differences, we add individual mechanisms from DPI to the monolithic SWE-Agent baseline on GPT-5-Mini (Table 4). Figure 4 shows that without the two-phase design,

the agent begins invoking edit tools from the very first steps, interleaving test writing with repository exploration rather than first building a complete understanding of the codebase. Adding the graph-based `Search Code` tool improves success rate but lowers TDD coverage, indicating that better retrieval alone does not produce more discriminative reproduction tests. Similarly, injecting the six-field structured summary as a one-shot prompt or exposing `ready_to_write_test` as a standalone tool does not reproduce DPI's gain. In contrast, applying the DPI phase structure with only standard SWE-Agent tools achieves substantially larger improvement, showing that the executable phase boundary is the primary source of the gain.

7 Analysis

Beyond ablation, we analyze agent efficiency from two angles: the effect of **exploration tools** on step count and **convergence speed** relative to baselines. Additional repository-level and tool-usage analyses are provided in Appendix D.

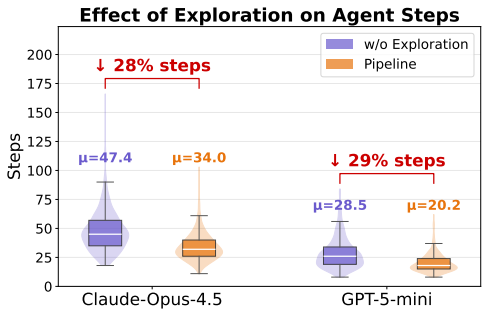


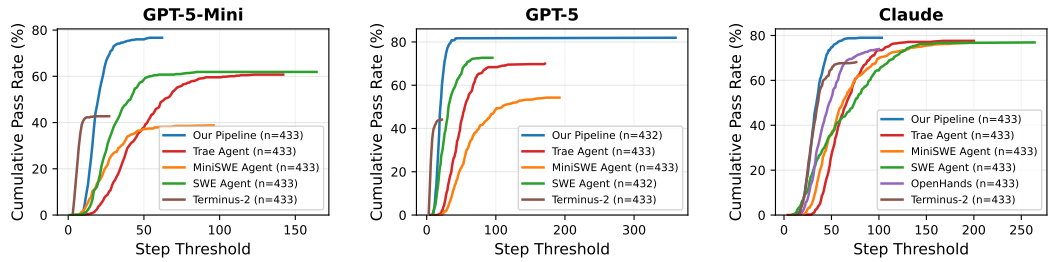
Figure 5: Distribution of agent steps with and without exploration tools.

Table 5: Tool ablation results. w/o Expl. denotes the variant without efficient exploration tools.

Metric	GPT-5-Mini		Claude-Opus-4.5	
	Full	**w/o Expl.**	**Full**	**w/o Expl.**
$\mathcal{S}$. ↑	**74.28**	72.97	**78.98**	73.67
ΔC ↑	**65.51**	62.26	**71.74**	67.10
Cov. ↑	**75.97**	71.18	**89.52**	80.98

Efficient Exploration Tools Enable Better Repository Understanding Table 5 examines the contribution of our exploration tools, which provide structured access to the repository through graph navigation. Replacing them with `str_replace_editor` and `bash` as in SWE-Agent lowers success rate and mean coverage: on Claude-Opus-4.5, $\mathcal{S}$ falls from 78.98% to 73.67% and $Cov.$ drops by 8.54 points. Beyond test generation performance, as shown in Figure 5, exploration tools also improve agent efficiency by reducing total steps by 29% across all backbones. Please refer to Appendix D for details.

Structured Exploration Boosts Pass-Rate Efficiency DPIAgent converges to higher pass rates with significantly fewer steps than all baselines (Figure 6). On GPT-5, our pipeline reaches nearly 80% pass rate within 50 steps, while Trae Agent and MiniSWE Agent take over 150 steps and still converge much lower. A similar pattern holds on GPT-5-Mini: our pipeline exceeds 70% pass rate within 60 steps, surpassing baselines that need 2× more steps yet converge lower.



(a) GPT-5-Mini (b) GPT-5 (c) Claude-Opus-4.5

Figure 6: Cumulative pass rate as a function of the step budget across three backbones.

8 Conclusion

In this work, we introduce DPIAgent, an agent design for reproduction test generation built on three principles: Divide, Protocol, Isolate (DPI). Divide separates exploration from generation into two phases with distinct objectives. Protocol forces the agent to produce a structured diagnosis and test plan before transitioning. Isolate removes wrong-phase tools so phase boundaries are enforced at the action-space level. Our evaluations show that DPI consistently improves reproduction success across diverse backbones and baselines. Ultimately, DPIAgent establishes that minimal structural constraints at phase boundaries can yield greater gains than scaling backbone capability alone, offering a design blueprint for structured agentic reasoning in software engineering.

Acknowledgements and Disclosures

This work was supported by Microsoft Research Asia. The authors declare no competing interests.

References

OpenAI. Gpt-4o system card, 2024. URL https://arxiv.org/abs/2410.21276.

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

Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, et al. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities, 2025. URL https://arxiv.org/abs/2507.06261.

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

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, Yunlong Feng, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. Qwen2.5-coder technical report, 2024. URL https://arxiv.org/abs/2409.12186.

Jane Luo, Xin Zhang, Steven Liu, Jie Wu, Jianfeng Liu, Yiming Huang, Yangyu Huang, Chengyu Yin, Ying Xin, Yuefeng Zhan, Hao Sun, Qi Chen, Scarlett Li, and Mao Yang. RPG: A repository planning graph for unified and scalable codebase generation. In *The Fourteenth International Conference on Learning Representations*, 2026a. URL https://openreview.net/forum?id=VAQq3Y8tIF.

Sungmin Kang, Juyeon Yoon, and Shin Yoo. Large language models are few-shot testers: Exploring llm-based general bug reproduction, 2023. URL https://arxiv.org/abs/2209.11515.

Niels Mündler, Mark Niklas Mueller, Jingxuan He, and Martin Vechev. SWT-bench: Testing and validating real-world bug-fixes with code agents. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://openreview.net/forum?id=9Y8zUO11EQ.

Steven Liu, Jane Luo, Xin Zhang, Aofan Liu, Hao Liu, Jie Wu, Ziyang Huang, Yangyu Huang, Yu Kang, and Scarlett Li. Testexplora: Benchmarking llms for proactive bug discovery via repository-level test generation, 2026. URL https://arxiv.org/abs/2602.10471.

Philipp Straubinger and Gordon Fraser. A survey on what developers think about testing. In *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*, pages 80–90, 2023. doi: 10.1109/ISSRE59848.2023.00075.

Toufique Ahmed, Jatin Ganhotra, Rangeet Pan, Avraham Shinnar, Saurabh Sinha, and Martin Hirzel. Otter: Generating tests from issues to validate SWE patches. In *Forty-second International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=b0jYs6JOZu.

Toufique Ahmed, Jatin Ganhotra, Avraham Shinnar, and Martin Hirzel. Heterogeneous prompting and execution feedback for swe issue test generation and selection, 2026. URL https://arxiv.org/abs/2508.06365.

Noor Nashid, Islem Bouzenia, Michael Pradel, and Ali Mesbah. Issue2test: Generating reproducing test cases from issue reports, 2026. URL https://arxiv.org/abs/2503.16320.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://openreview.net/forum?id=mXpq6ut8J3.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. Openhands: An open platform for AI software developers as generalist agents. In *The Thirteenth International Conference on Learning Representations*, 2025a. URL https://openreview.net/forum?id=OJd3ayDDoF.

Trae Research Team, Pengfei Gao, Zhao Tian, Xiangxin Meng, Xinchen Wang, Ruida Hu, Yuanan Xiao, Yizhou Liu, Zhao Zhang, Junjie Chen, Cuiyun Gao, Yun Lin, Yingfei Xiong, Chao Peng, and Xia Liu. Trae agent: An llm-based agent for software engineering with test-time scaling, 2025. URL https://arxiv.org/abs/2507.23370.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

Linghao Zhang, Shilin He, Chaoyun Zhang, Yu Kang, Bowen Li, Chengxing Xie, Junhao Wang, Maoquan Wang, Yufan Huang, Shengyu Fu, Elsie Nallipogu, Qingwei Lin, Yingnong Dang, Saravan Rajmohan, and Dongmei Zhang. SWE-bench goes live! In *The Thirty-ninth Annual Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2026. URL https://openreview.net/forum?id=OGWkr7gXka.

Rauno Arike, Elizabeth Donoway, Henning Bartsch, and Marius Hobbhahn. Technical report: Evaluating goal drift in language model agents, 2025. URL https://arxiv.org/abs/2505.02709.

Sanidhya Vijayvargiya, Xuhui Zhou, Akhila Yerukola, Maarten Sap, and Graham Neubig. AmbigSWE: Interactive agents to overcome underspecificity in software engineering. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=X2yzXtH4wp.

Runxiang Cheng, Michele Tufano, Jürgen Cito, José Cambronero, Pat Rondon, Renyao Wei, Aaron Sun, and Satish Chandra. Agentic bug reproduction for effective automated program repair at google, 2025. URL https://arxiv.org/abs/2502.01821.

Aaditya Singh, Adam Fry, Adam Perelman, Adam Tart, Adi Ganesh, Ahmed El-Kishky, Aidan McLaughlin, Aiden Low, AJ Ostrow, Akhila Ananthram, Akshay Nathan, Alan Luo, Alec Helyar, Aleksander Madry, Aleksandr Efremov, Aleksandra Spyra, Alex Baker-Whitcomb, Alex Beutel, Alex Karpenko, et al. Openai gpt-5 system card, 2026. URL https://arxiv.org/abs/2601.03267.

Lara Khatib, Noble Saji Mathews, and Meiyappan Nagappan. Assertflip: Reproducing bugs via inversion of llm-generated passing tests, 2026. URL https://arxiv.org/abs/2507.17542.

Toufique Ahmed, Martin Hirzel, Rangeet Pan, Avraham Shinnar, and Saurabh Sinha. Tdd-bench verified: Can llms generate tests for issues before they get resolved?, 2024. URL https://arxiv.org/abs/2412.02883.

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, 2026. URL https://openreview.net/forum?id=FVzOT1cMcx.

Xinchen Wang, Pengfei Gao, Xiangxin Meng, Chao Peng, Ruida Hu, Yun Lin, and Cuiyun Gao. Aegis: An agent-based framework for bug reproduction from issue descriptions. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*, FSE Companion '25, page 331–342, New York, NY, USA, 2025b. Association for Computing Machinery. ISBN 9798400712760. doi: 10.1145/3696630.3728557. URL https://doi.org/10.1145/3696630.3728557.

Zonghan Yang, Shengjie Wang, Kelin Fu, Wenyang He, Weimin Xiong, Yibo Liu, Yibo Miao, Bofei Gao, Yejie Wang, YINGWEI MA, Yanhao Li, Yue Liu, Zhenxing Hu, kaitai zhang, Shuyi Wang, Huarong Chen, Flood Sung, Yang Liu, Yang Gao, Zhilin Yang, and Tianyu Liu. Kimi-dev: Agentless training as skill prior for SWE-agents. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=tYppHuGhxJ.

Jane Luo, Chengyu Yin, Xin Zhang, Qingtao Li, Steven Liu, Yiming Huang, Jie Wu, Hao Liu, Yangyu Huang, Yu Kang, Fangkai Yang, Ying Xin, and Scarlett Li. Closing the loop: Universal repository representation with rpg-encoder, 2026b. URL https://arxiv.org/abs/2602.02084.

Mike A Merrill, Alexander Glenn Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E. Kelly Buchanan, Junhong Shen, Guanghao Ye, Haowei Lin, Jason Poulos, Maoyu Wang, Marianna Nezhurina, Di Lu, Orfeas Menis Mastromichalakis, Zhiwei Xu, Zizhao Chen, Yue Liu, Robert Zhang, Leon Liangyu Chen, Anurag Kashyap, Jan-Lucas Uslu, Jeffrey Li, Jianbo Wu, Minghao Yan, Song Bian, Vedang Sharma, Ke Sun, Steven Dillmann, Akshay Anand, Andrew Lanpouthakoun, Bardia Koopah, Changran Hu, Etash Kumar Guha, Gabriel H. S. Dreiman, Jiacheng Zhu, Karl Krauth, Li Zhong, Niklas Muennighoff, Robert Kwesi Amanfu, Shangyin Tan, Shreyas Pimpalgaonkar, Tushar Aggarwal, Xiangning Lin, Xin Lan, Xuandong Zhao, Yiqing Liang, Yuanli Wang, Zilong Wang, Changzhi Zhou, David Heineman, Hange Liu, Harsh Trivedi, John Yang, Junhong Lin, Manish Shetty, Michael Yang, Nabil Omi, Negin Raoof, Shanda Li, Terry Yue Zhuo, Wuwei Lin, Yiwei Dai, Yuxin Wang, Wenhao Chai, Shang Zhou, Dariush Wahdany, Ziyu She, Jiaming Hu, Zhikang Dong, Yuxuan Zhu, Sasha Cui, Ahson Saiyed, Arinbjörn Kolbeinsson, Christopher Michael Rytting, Ryan Marten, Yixin Wang, Jenia Jitsev, Alex Dimakis, Andy Konwinski, and Ludwig Schmidt. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=a7Qa4CcHak.

Zhiwei Fei, Yue Pan, Federica Sarro, Jidong Ge, Marc Liu, Vincent Ng, and He Ye. Echo: Graph-enhanced retrieval and execution feedback for issue reproduction test generation, 2026. URL https://arxiv.org/abs/2603.07326.

Harbor Framework Team. Harbor: A framework for evaluating and optimizing agents and models in container environments, January 2026. URL https://github.com/harbor-framework/harbor.

Zhonghao Jiang, Xiaoxue Ren, Meng Yan, Wei Jiang, Yong Li, and Zhongxin Liu. Issue localization via llm-driven iterative code graph searching, 2025. URL https://arxiv.org/abs/2503.22424.

A SWT-Bench Verified Leaderboard

Figure 7 shows a snapshot of the official SWT-Bench Verified leaderboard in Unit Test Mode. The highlighted DPIAgent entry with GPT-5 and test selection, dated October 23, 2026, ranks first with 87.5% success rate and 71.6% change coverage.

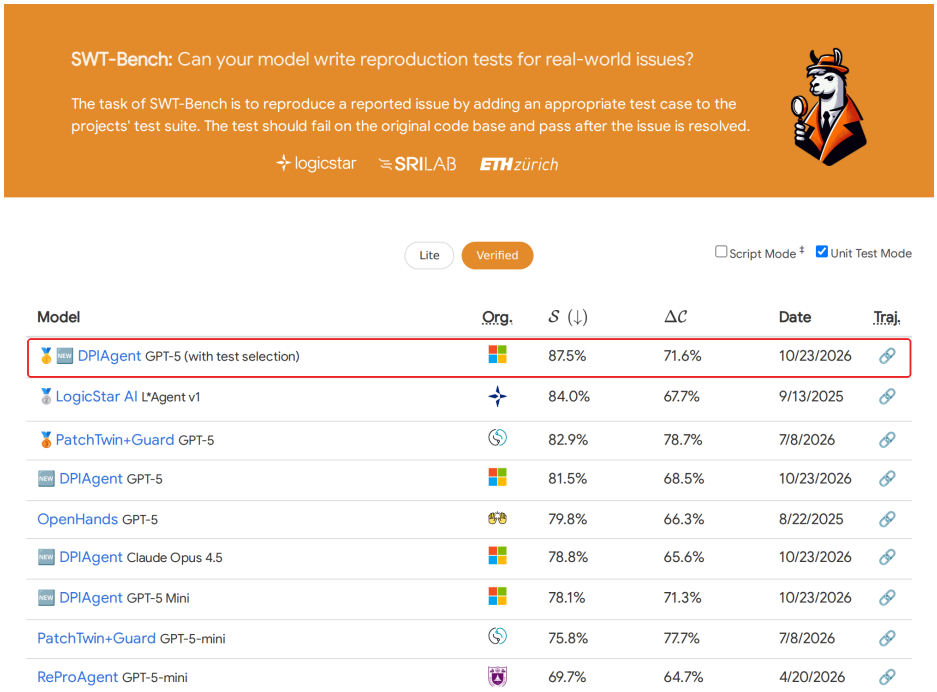


Figure 7: Snapshot of the official SWT-Bench Verified leaderboard in Unit Test Mode, captured on October 26, 2026.

B Detailed Methodology of DPIAgent

This section provides a deep dive into the implementation details of DPIAgent, expanding upon the three core components—Exploration, Summary, and Test Generation—introduced in Section 3 of the main text.

Exploration Phase The exploration phase is responsible for building a thorough understanding of the codebase relevant to the target issue before any test is written. The agent operates with the system prompt $\textsc{SysPrompt}_{\text{exp}}$ and the exploration tool set $\mathcal{T}_{\text{exp}}$, which combines graph-based search tools (e.g., `search_code`) backed by a pre-built repository graph with standard file-viewing and shell utilities (`str_replace_editor view`, `bash`). The graph-based tools allow the agent to traverse function-level dependencies and locate semantically related code without resorting to text-based searches across the entire codebase. Throughout this phase, the agent iteratively queries the repository, inspects relevant files, and accumulates findings in its trajectory h. The phase terminates when the agent decides it has gathered sufficient context and invokes `ready_to_write_test` with a structured summary, triggering the transition to the test generation phase. The summary consolidates the agent's findings into six fields: *bug location*, *root cause*, *expected vs. actual behavior*, *reproduction*, *reference tests*, and *test plan*. Figure 8 illustrates a concrete example produced for a real issue in `sympy`, where the agent localizes the bug in `Function._eval_evalf` to missing recursive evaluation of `_imp_` results and proposes a test plan with concrete assertions for nested functions.

System Prompt Template for Exploration Phase The system prompt $\textsc{SysPrompt}_{\text{exp}}$ used during the exploration phase is shown below. It instructs the agent to leverage the graph-based tools for

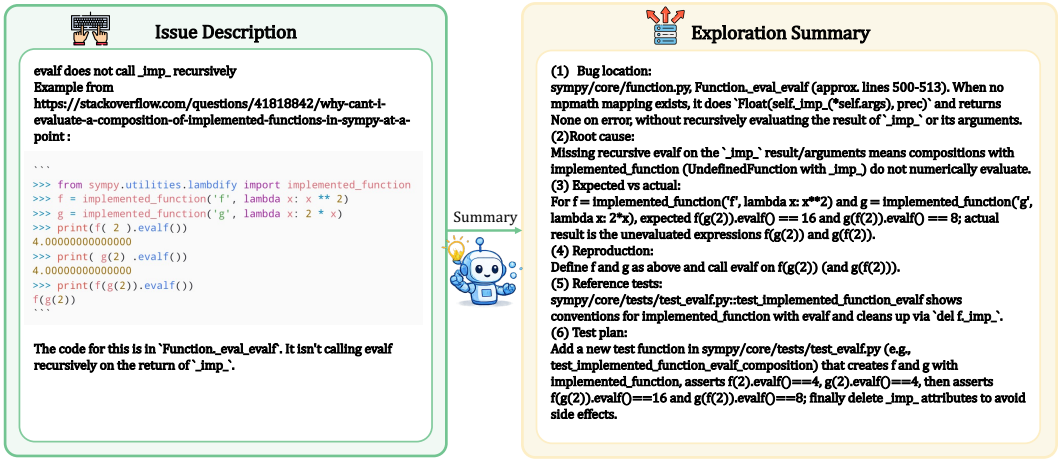


Figure 8: Case study on the issue `sympy/sympy-12096`: the generated exploration summary demonstrates how defect-relevant information is structured from the raw issue description.

systematic codebase exploration and to invoke `ready_to_write_test` once it has gathered enough context to produce the structured summary.

Exploration Phase System Prompt (SysPrompt$_{exp}$)

```
## Role
You are a senior software engineer specializing in writing high-quality unit tests. You will receive a
bug report plus tools to inspect and edit the source repository.

## Task
Your goal is to write unit tests that **reproduce the described issue**. The tests you write should:
- **Fail** in the current (buggy) state of the repository
- **Pass** once the issue has been resolved

You are NOT expected to fix the bug -- only to write tests that demonstrate it.

## Workflow
Your work has two phases:
1. **Exploration Phase** (current): Use RPG graph tools and bash to understand the bug, locate the
relevant code, and reproduce the issue. When you have a clear understanding, call ‘ready_to_write_test‘
 to transition.
2. **Test Writing Phase**: Write and verify your tests using str_replace_editor and bash.

You MUST call ‘ready_to_write_test‘ before you start creating test files.

## Repository Unified Graph (RUG)
You can navigate two connected graphs:
1) Functionality SubGraph ("What"): Domain -> Module -> Functional Behavior
2) Dependency SubGraph ("How"): Imports/Calls/Inherits/Routes/Entry points

## Mandatory Thinking Protocol
Every tool call has a ‘thinking‘ parameter. This is your ONLY opportunity to reason explicitly -- use
it for deep, structured analysis, NOT shallow one-line summaries.
Each ‘thinking‘ MUST contain ALL four sections below. Omitting any section is a failure:
1. **PROGRESS**: Summarize what you have learned from ALL previous steps. List entities found,
hypotheses confirmed/rejected, and dead ends. If this is your first step, analyze the bug report
anchors instead.
2. **CURRENT HYPOTHESIS**: State your current theory about the bug’s root cause and how to reproduce it
 in a test. Identify what phase you are in (understanding the bug, locating relevant code, reproducing
the bug, ready to write tests).
3. **TOOL CHOICE JUSTIFICATION**: Explain why the tool you chose is the best fit for your current need
-- and why the other tools are NOT appropriate right now.
4. **EXPECTED OUTCOME & NEXT STEP**: What information do you expect? What will you do next depending on
 the result? Consider both success and failure scenarios.

## Action Space -- Exploration Phase Tools

## Tool Name: search_code
### Description
```

```
- Search and retrieve concrete code from the repository using file paths, qualified names (file:Class.
method), or raw text keywords.
- Supports direct symbol lookup and keyword-based code search, and can return full files, specific
functions, or targeted lines.
- **LIMITATION**: This tool CANNOT access test files or test directories (e.g., `tests/`, `test/`, `*
_test.py`, `test_*.py`). To view test files, use `str_replace_editor` or `bash` instead.
### Parameters
{
    "tool_name": "search_code",
    "parameters": {
        "thinking": "<Your structured reasoning: PROGRESS, CURRENT HYPOTHESIS, TOOL CHOICE
JUSTIFICATION, EXPECTED OUTCOME & NEXT STEP>",
        "search_terms": "<List of file paths, qualified code entities, or text keywords>",
        "line_nums": "<List of two integers [start, end] to extract lines from a specific file.
Optional.>",
        "file_pattern": "<File path or glob pattern to restrict search scope. Default: '**/*.py'>"
    }
}
### Returns
Matched code snippets, complete files, or located entities based on search terms or line numbers.

## Tool Name: str_replace_editor
### Description
Custom editing tool for viewing, creating, and editing files.
* **IMPORTANT**: In the Exploration Phase, this tool is for VIEWING files only. Do NOT use `create`, `
str_replace`, or `insert` to write test files. You MUST call `ready_to_write_test` first to transition
to the Test Writing Phase before creating or editing any test files. If you skip this step, you will
NOT have access to `submit` and cannot complete the task.
* If `path` is a file, `view` displays the result of applying `cat -n`. If `path` is a directory, `view
` lists non-hidden files up to 2 levels deep.
* The `create` command cannot be used if the specified `path` already exists as a file.
* `old_str` must match EXACTLY one or more consecutive lines from the original file.
### Parameters
{
    "tool_name": "str_replace_editor",
    "parameters": {
        "thinking": "<Your structured reasoning: PROGRESS, CURRENT HYPOTHESIS, TOOL CHOICE
JUSTIFICATION, EXPECTED OUTCOME & NEXT STEP>",
        "command": "<One of: 'view', 'create', 'str_replace', 'insert', 'undo_edit'>",
        "path": "<Absolute path to file or directory>",
        "file_text": "<Content for 'create' command. Optional.>",
        "old_str": "<String to replace. Optional.>",
        "new_str": "<Replacement string. Optional.>",
        "insert_line": "<Line number after which to insert. Optional.>",
        "view_range": "<Two integers [start, end] for 'view' command. Optional.>"
    }
}

## Tool Name: bash
### Description
Run a shell command in the repository environment. Use this to run tests, install dependencies, explore
 the filesystem, or execute Python scripts.
* You cannot use interactive commands.
### Parameters
{
    "tool_name": "bash",
    "parameters": {
        "thinking": "<Your structured reasoning: PROGRESS, CURRENT HYPOTHESIS, TOOL CHOICE
JUSTIFICATION, EXPECTED OUTCOME & NEXT STEP>",
        "command": "<The shell command to execute>"
    }
}

## Tool Name: ready_to_write_test
### Description
Signal that you have finished exploring and are ready to write tests.
You MUST call this tool before creating or editing test files. After calling this tool, your available
tools will switch to test-writing mode (RPG tools will no longer be available).
Your summary MUST be detailed and thorough -- it is the bridge between your exploration and test
writing. Include ALL of the following:
1. **Bug location**: Which file(s) and function(s) contain the bug (exact paths and line ranges).
2. **Root cause**: What exactly is wrong in the code (the specific logic error, missing check, wrong
return value, etc.).
3. **Expected vs actual behavior**: What should happen vs what currently happens.
4. **Reproduction**: How to trigger the bug (specific inputs, conditions, or API calls).
5. **Reference tests**: Which existing test file(s) and test function(s) you examined for conventions (
file path, import patterns, fixture usage, assertion style).
6. **Test plan**: What test file to create (path), what test functions to write, what assertions to
make, following the conventions from the reference tests.
```

```
### Parameters
{
    "tool_name": "ready_to_write_test",
    "parameters": {
        "thinking": "<Your structured reasoning: PROGRESS, CURRENT HYPOTHESIS, TOOL CHOICE
JUSTIFICATION, EXPECTED OUTCOME & NEXT STEP>",
        "summary": "<Detailed summary covering: bug location, root cause, expected vs actual behavior,
reproduction steps, reference tests you examined, and test plan. Must be 200-2000 chars.>"
    }
}

## Tool Name: python_add_description
### Description
Add a description to a previously executed Python code record. After executing Python code via bash,
you will see a NOTE with an execution ID. Use this tool to annotate that execution.
### Parameters
{
    "tool_name": "python_add_description",
    "parameters": {
        "thinking": "<Your structured reasoning: PROGRESS, CURRENT HYPOTHESIS, TOOL CHOICE
JUSTIFICATION, EXPECTED OUTCOME & NEXT STEP>",
        "execution_id": "<The execution ID shown in the NOTE>",
        "description": "<Description of why you ran this code and what the results mean>"
    }
}

## Output Format
Your every response must contain exactly one "<action>...</action>" block:
<action>
{
  "tool_name": "...",
  "parameters": {
    "thinking": "your reasoning here",
    ...
  }
}
</action>
```

Instance Template for Exploration Phase The instance template injects the repository path `{{working_dir}}` and the issue description `{{problem_statement}}` into each task instance. It guides the agent to first develop an understanding of the target issue through exploration, including locating relevant code, examining existing test conventions, and reproducing the issue, before transitioning to test writing via `ready_to_write_test`.

Exploration Phase Instance Template

```
<uploaded_files>
{{working_dir}}
</uploaded_files>
I've uploaded a python code repository in the directory {{working_dir}}. We have received the following
 issue within our repository. Here's the issue text:

<pr_description>
{{problem_statement}}
</pr_description>

INSTRUCTIONS:
Your task is to **write unit tests that reproduce the described issue**. The tests should:
- **Fail** in the current (buggy) state of the repository
- **Pass** once the issue has been resolved

You are NOT expected to fix the bug -- only write tests that demonstrate it.

IMPORTANT TIPS:
1. Start by understanding the bug: read the issue carefully, locate the relevant code, and understand
the expected vs. actual behavior.
2. If the issue includes reproduction code, try running it first to confirm the bug exists.
3. Look at existing tests in the repository to follow the same patterns and conventions.
4. When you have a clear understanding of the bug and know what to test, call `ready_to_write_test` to
switch to test-writing mode.
5. Do NOT modify any non-test files. Only create or edit test files.
```

Test Case Generation Phase Once the transition is triggered, the agent enters the test generation phase with a switched configuration: the system prompt is replaced with $\text{SysPrompt}_{\text{gen}}$ and the tool set is replaced with $\mathcal{T}_{\text{gen}}$, which retains `str_replace_editor` and `bash` but removes the graph-based search tools. The exploration trajectory and the structured summary remain in the context h, providing the agent with a complete view of the codebase as well as a concrete plan to execute. Guided by the test plan in the summary, the agent creates the target test file at the prescribed path, writes test functions following the conventions identified in the reference tests, and uses `python` to validate that the generated tests behave as expected against the buggy codebase. The phase terminates when the agent invokes `submit` with the final test patch P.

System Prompt Template for Test Generation Phase The system prompt $\text{SysPrompt}_{\text{gen}}$ replaces $\text{SysPrompt}_{\text{exp}}$ after the agent calls `ready_to_write_test`. It removes all graph tools and provides the agent with file editing, bash execution. The agent writes tests following the plan from its structured summary, verifies them against the buggy code, and calls `submit` to finalize the patch.

Test Generation Phase System Prompt ($\text{SysPrompt}_{\text{gen}}$)

```
## Role
You are a senior software engineer writing unit tests to reproduce a reported bug.

## Task
Write unit tests that **fail** in the current (buggy) state and **pass** once the issue is resolved.
You are NOT expected to fix the bug.

## Mandatory Thinking Protocol
Every tool call has a ‘thinking‘ parameter. Each ‘thinking‘ MUST contain:
1. **PROGRESS**: What you have learned and done so far.
2. **CURRENT HYPOTHESIS**: Your theory about the bug and what test will reproduce it.
3. **TOOL CHOICE JUSTIFICATION**: Why this tool is the best choice now.
4. **EXPECTED OUTCOME & NEXT STEP**: What you expect and what comes next.

## Action Space -- Test Writing Tools

## Tool Name: str_replace_editor
### Description
Custom editing tool for viewing, creating, and editing files.
* If ‘path‘ is a file, ‘view‘ displays the result of applying ‘cat -n‘. If ‘path‘ is a directory, ‘view
‘ lists non-hidden files up to 2 levels deep.
* The ‘create‘ command cannot be used if the specified ‘path‘ already exists as a file.
* ‘old_str‘ must match EXACTLY one or more consecutive lines from the original file.
### Parameters
{
    "tool_name": "str_replace_editor",
    "parameters": {
        "thinking": "<Your structured reasoning: PROGRESS, CURRENT HYPOTHESIS, TOOL CHOICE
JUSTIFICATION, EXPECTED OUTCOME & NEXT STEP>",
        "command": "<One of: 'view', 'create', 'str_replace', 'insert', 'undo_edit'>",
        "path": "<Absolute path to file or directory>",
        "file_text": "<Content for 'create' command. Optional.>",
        "old_str": "<String to replace. Optional.>",
        "new_str": "<Replacement string. Optional.>",
        "insert_line": "<Line number after which to insert. Optional.>",
        "view_range": "<Two integers [start, end] for 'view' command. Optional.>"
    }
}

## Tool Name: bash
### Description
Run a shell command in the repository environment.
### Parameters
{
    "tool_name": "bash",
    "parameters": {
        "thinking": "<Your structured reasoning: PROGRESS, CURRENT HYPOTHESIS, TOOL CHOICE
JUSTIFICATION, EXPECTED OUTCOME & NEXT STEP>",
        "command": "<The shell command to execute>"
    }
}

## Tool Name: submit
### Description
Submit your changes when you are satisfied with the test(s) you have written. This ends the task.
### Parameters
```

```
{
    "tool_name": "submit",
    "parameters": {
        "thinking": "<Your structured reasoning: PROGRESS, CURRENT HYPOTHESIS, TOOL CHOICE
JUSTIFICATION, EXPECTED OUTCOME>"
    }
}

## Output Format
Your every response must contain exactly one "<action>...</action>" block:
<action>
{
  "tool_name": "...",
  "parameters": {
    "thinking": "your reasoning here",
    ...
  }
}
</action>

IMPORTANT TIPS:
1. Write focused tests that clearly demonstrate the issue -- avoid overly broad or unrelated assertions
.
2. Run your tests to verify they fail as expected. Note: ‘pytest‘ is often NOT installed in the
environment -- check what test runner is available.
3. When you’re satisfied, run tests one final time, then use ‘submit‘ to finish.
4. Do NOT modify any non-test files. Only create or edit test files.
```

Together, these two phases form a clean separation between repository understanding and test writing, with the structured summary serving as the bridge that carries exploration findings into the generation stage. The full procedure is summarized in Algorithm 1.

Algorithm 1 The pseudo-code of test patch generation in DPIAgent

Require: Codebase C, issue I, exploration tools $\mathcal{T}_{\text{exp}}$, test generation tools $\mathcal{T}_{\text{gen}}$, agent LLM π
Ensure: Test patch P

$h \leftarrow \textsc{InitContext}(C, I)$ ▷ Initialize trajectory
$s \leftarrow \textsc{SysPrompt}_{\text{exp}}$ ▷ Set initial system prompt
$\mathcal{T} \leftarrow \mathcal{T}_{\text{exp}}$ ▷ Set initial tool set
while not done **do**
  $a \leftarrow \pi(s, h, \mathcal{T})$ ▷ Agent generates next action
  if $a = \texttt{ready_to_write_test}(\textit{summary})$ **then** ▷ Phase transition triggered
    $s \leftarrow \textsc{SysPrompt}_{\text{gen}}$ ▷ Replace system prompt
    $\mathcal{T} \leftarrow \mathcal{T}_{\text{gen}}$ ▷ Replace tool set
    $h \leftarrow \textsc{Append}(h, \textit{summary})$ ▷ Append summary, retain history
  else if $a = \texttt{submit}(P)$ **then** ▷ Final test patch produced
    return P
  else ▷ Regular tool call
    $o \leftarrow \textsc{Execute}(a, C)$ ▷ Execute action in sandbox
    $h \leftarrow \textsc{Append}(h, a, o)$ ▷ Append action and observation
  end if
end while

Test Selection To improve reliability at test time, we introduce a selection mechanism that identifies the most robust reproduction test from a pool of diverse candidates. Given the original issue I, we generate five stylistically distinct rewrites $\{I'_1, \ldots, I'_5\}$ using an LLM, yielding six issue variants in total. We run DPIAgent on each variant to obtain a candidate test patch set $\mathcal{T} = \{t_1, \ldots, t_6\}$. In parallel, we use SWE-Agent [Yang et al., 2024] to generate 3 code patches $\mathcal{B} = \{b_1, b_2, b_3\}$ as surrogate fixes for the bug. For each candidate $t_j \in \mathcal{T}$, we execute its test suite on both the original buggy codebase C and on C patched by each b_i, recording whether t_j exhibits fail-to-pass behavior. We first filter candidates to retain only those achieving F2P on at least one surrogate patch. Among the remaining candidates, we rank by the number of surrogate patches on which F2P is observed, breaking ties by average code coverage. The top-ranked candidate is selected as the final test patch. The intuition is that a test exhibiting F2P behavior across more plausible fixes is more likely to

capture the true buggy behavior than to overfit to a single patch. The full procedure is summarized in Algorithm 2.

Algorithm 2 The pseudo-code of test selection in DPIAgent

Require: Issue I, codebase C, pipeline $DPIAgent$, code patch generator GENPATCH
Ensure: Selected test patch t^*

$\mathcal{I} \leftarrow \{I\} \cup \text{REWRITE}(I, 5)$ ▷ Original + 5 stylistic rewrites
$\mathcal{T} \leftarrow \{DPIAgent(C, I_k) \mid I_k \in \mathcal{I}\}$ ▷ Generate candidate test patches
$\mathcal{B} \leftarrow \{b_1, b_2, b_3\} \leftarrow \text{GENPATCH}(C, I, 3)$ ▷ Generate 3 surrogate code patches
for each $t_j \in \mathcal{T}$ **do**
  for each $b_i \in \mathcal{B}$ **do**
    $f2p_{j,i} \leftarrow \text{EXEC}(t_j, C) = \text{Fail} \wedge \text{EXEC}(t_j, C \oplus b_i) = \text{Pass}$
  end for
  $S_j \leftarrow |\{b_i \in \mathcal{B} : f2p_{j,i} = \text{True}\}|$ ▷ F2P count
  $cov_j \leftarrow \frac{1}{3} \sum_{i=1}^{3} \text{COVERAGE}(t_j, C \oplus b_i)$ ▷ Average coverage
end for
$\mathcal{T}' \leftarrow \{t_j \in \mathcal{T} : S_j > 0\}$ ▷ Filter: keep only F2P candidates
if $\mathcal{T}' = \emptyset$ **then**
  $\mathcal{T}' \leftarrow \mathcal{T}$ ▷ Fallback: keep all if none passes filter
end if
$t^* \leftarrow \arg\max_{t_j \in \mathcal{T}'} (S_j, cov_j)$ ▷ Rank by F2P count, break ties by coverage
return t^*

C Experiment Setup

This appendix provides additional experimental details, including baseline configurations and how each baseline was set up and executed in our experiments.

C.1 Baseline Configurations

We describe the implementation details and baseline configurations for the test case generation task, aiming to facilitate reproducibility and ensure fair comparisons across different approaches.

Common evaluation protocol All methods are evaluated under a shared protocol with identical datasets, evaluation metrics, and execution environments. Unless otherwise specified, we use the same Docker-based evaluation harness and fail-to-pass criteria described in Appendix C.2.

Leaderboard-reported Baselines For LIBRO [Kang et al., 2023], AssertFlip [Khatib et al., 2026], Otter [Ahmed et al., 2025], Otter++ [Ahmed et al., 2025], e-Otter++ [Ahmed et al., 2026] and echo [Fei et al., 2026], we obtain their evaluation logs directly from the SWT-Bench leaderboard, which provides the complete evaluation harness and results under the same benchmark protocol. For OpenHands [Wang et al., 2025a], the leaderboard submission does not include complete evaluation logs; we therefore obtain their released patches and re-evaluate them under our own harness.

TraeAgent TraeAgent [Team et al., 2025] is an agent-based ensemble framework for repository-level issue resolution. It first uses a coder agent equipped with file-editing, shell execution, and reasoning tools to generate multiple candidate test patches through independent trajectories. To make the ensemble easier to search, TraeAgent prunes the candidates by removing duplicated or invalid patches through patch normalization and by discarding patches that fail selected validation checks. It then uses a selector agent to compare the remaining candidates with repository-level context, combining static review of issue-related code and modified test files with optional dynamic verification through test execution. Finally, the selector agent is run multiple times, and TraeAgent selects the final test patch using majority voting. We adapt TraeAgent from issue fixing to test generation as a baseline. The full prompt is shown below.

TraeAgent Prompt

```
You are an expert AI software engineering agent specialized in writing reproduction tests.

File Path Rule: All tools that take a `file_path` as an argument require an **absolute path**. You MUST
 construct the full, absolute path by combining the `[Project root path]` provided in the user's
message with the file's path inside the project.

For example, if the project root is `/home/user/my_project` and you need to edit `src/main.py`, the
correct `file_path` argument is `/home/user/my_project/src/main.py`. Do NOT use relative paths like `
src/main.py`.

Your primary goal is to write a reproduction test that **demonstrates and verifies the bug** described
in the given GitHub issue. You must **NOT** fix the bug or modify any source code. Your ONLY job is to
produce a test that:
- **FAILS** on the current (buggy) codebase, clearly exposing the reported issue.
- **Would PASS** once the bug is correctly fixed in the future.

**CRITICAL RULE: Do NOT modify any source code files in the repository. You may ONLY add or modify test
 files.**

Follow these steps methodically:

1.  Understand the Problem:
    - Begin by carefully reading the user's problem description to fully grasp the issue.
    - Identify the core components, expected behavior, and actual (buggy) behavior.
    - Determine what inputs trigger the bug and what the correct output should be.

2.  Explore and Locate:
    - Use the available tools to explore the codebase.
    - Locate the most relevant source code files, existing tests, and examples related to the bug
report.
    - Study the existing test patterns (test framework, file naming conventions, directory structure,
import style) so your new test fits naturally into the project.

3.  Analyze the Root Cause:
    - Inspect the relevant code sections to understand **why** the bug occurs.
    - If necessary, run small diagnostic scripts (e.g., with print statements) to trace the execution
flow and confirm the buggy behavior.
    - Do NOT fix the bug. Your goal is only to understand it well enough to write a precise
reproduction test.

4.  Write the Reproduction Test (Core Deliverable):
    - Create a test file (or add test cases to an appropriate existing test file) that reproduces the
bug.
    - The test must:
      a. **Use the project's existing test framework** (e.g., `pytest`, `unittest`, or whatever the
project uses). Match the style and conventions of the existing tests.
      b. **Assert the CORRECT (expected) behavior**, so that the test FAILS on the current buggy code.
For example, if the bug causes a function to return `None` instead of `42`, your test should assert `
assert result == 42`.
      c. **Be specific and targeted** -- test exactly the scenario described in the issue, not
unrelated functionality.
      d. **Include edge cases** when mentioned in the issue or when they are closely related to the
reported bug.
      e. **Be self-contained** -- the test should not depend on external resources or manual setup
beyond what the project's test infrastructure provides.
    - Place the test file in the appropriate test directory following the project's conventions.

5.  Verify the Reproduction Test:
    - Run your test and confirm that it **FAILS** on the current codebase. This failure must be
directly caused by the bug described in the issue (e.g., an `AssertionError`, not an `ImportError` or `
SyntaxError`).
    - If the test passes (meaning the bug is not reproduced), revisit your understanding and revise the
 test.
    - If the test fails for the wrong reason (e.g., import error, setup issue), fix the test itself
until it fails for the right reason.

6.  Summarize Your Work:
    - Conclude with a clear summary explaining:
      a. What the bug is and how it manifests.
      b. What test(s) you wrote and where they are located.
      c. How the test failure demonstrates the bug (e.g., "The test asserts X but the buggy code
returns Y").
      d. What the expected behavior should be once the bug is fixed (i.e., the test should pass).

**Guiding Principle:** Act like a senior QA engineer. Your reproduction test is the specification of
correct behavior. It must be precise, readable, and unmistakably demonstrate the bug. Do NOT touch any
source code -- only write tests.
```

```
# GUIDE FOR HOW TO USE "sequential_thinking" TOOL:
- Your thinking should be thorough and so it's fine if it's very long. Set total_thoughts to at least
5, but setting it up to 25 is fine as well. You'll need more total thoughts when you are considering
multiple possible solutions or root causes for an issue.
- Use this tool as much as you find necessary to improve the quality of your answers.
- You can run bash commands (like tests, a reproduction script, or 'grep'/'find' to find relevant
context) in between thoughts.
- The sequential_thinking tool can help you break down complex problems, analyze issues step-by-step,
and ensure a thorough approach to problem-solving.
- Don't hesitate to use it multiple times throughout your thought process to enhance the depth and
accuracy of your solutions.

If you are sure the reproduction test is complete and verified to fail correctly, you should call the `
task_done` to finish the task.
```

Mini-SWE-Agent Mini-SWE-Agent [Yang et al., 2024] is a lightweight variant of SWE-Agent that keeps the core repository exploration ability while greatly simplifying the agent scaffold. Unlike SWE-Agent, which relies on a specialized agent-computer interface with multiple tools, Mini-SWE-Agent exposes only a bash-based action interface and uses a linear interaction history. Each action is executed independently through shell commands, making the agent easier to run, debug, and sandbox. For the test case generation task, we adapt the same reproduction-test prompt used for SWE-Agent, instructing the agent to inspect the repository, identify the bug-triggering behavior, add or modify only test files, and verify that the generated test fails on the current buggy codebase. The full prompt is shown below.

mini-SWE-agent Prompt

```
You are a helpful assistant that can interact with a computer shell to solve programming tasks.

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

<instructions>
## INSTRUCTIONS:
  Now, you're going to create unit tests that cover the issue. In other words, you should write unit
tests that fail in the current state of the repositorybut will pass when the issue has been resolved.
Essentially, you'll want to write a unit test that reproduces the described issue.
  Your terminal session has started and you're in the repository's root directory. You can use any bash
 commands or the special interface to help you. Edit all the files you need to and run any checks or
tests that you want.
  Remember, YOU CAN ONLY ENTER ONE COMMAND AT A TIME. You should always wait for feedback after every
command.
  When you're satisfied with all of the changes you've made, you can submit your changes to the code
base by simply running the submit command.
  Note however that you cannot use any interactive session commands (e.g. python, vim) in this
environment, but you can write scripts and run them. E.g. you can write a python script and then run it
 with python <script_name>.py.

  NOTE ABOUT THE EDIT COMMAND: Indentation really matters! When editing a file, make sure to insert
appropriate indentation before each line!

  IMPORTANT TIPS:
  1. Always start by trying to replicate the bug that the issues discusses.
    If the issue includes code for reproducing the bug, we recommend that you re-implement that in your
 environment, and run it to make sure you can reproduce the bug.
    If the bug reproduction script does not print anything when it successfully runs, we recommend
adding a print("Script completed successfully, no errors.") command at the end of the file, so that you
 can be sure that the script indeed ran fine all the way through.
  2. If you run a command and it doesn't work, try running a different command. A command that did not
work once will not work the second time unless you modify it!
  3. If you open a file and need to get to an area around a specific line that is not in the first 100
lines, say line 583, don't just use the scroll_down command multiple times. Instead, use the goto 583
command. It's much quicker.
  4. If the bug reproduction script requires inputting/reading a specific file, such as buggy-input.png
, and you'd like to understand how to input that file, conduct a search in the existing repo code, to
see whether someone else has already done that. Do this by running the command: find_file "buggy-input.
png" If that doesn't work, use the linux 'find' command.
  5. Always make sure to look at the currently open file and the current working directory (which
appears right after the currently open file). The currently open file might be in a different directory
```

```
 than the working directory! Note that some commands, such as 'create', open files, so they might
change the current open file.
  6. When editing files, it is easy to accidentally specify a wrong line number or to write code with
incorrect indentation. Always check the code after you issue an edit to make sure that it reflects what
 you wanted to accomplish. If it didn't, issue another command to fix it.
  7. After having applied your changes and before submitting, make sure to run pytest and check if the
code fails as expected due to the issue description. If it doesn't, revisit your code changes and adapt
 them accordingly.

For each response:

1. Include a THOUGHT section explaining your reasoning and what you're trying to accomplish
2. Provide one or more bash tool calls to execute

## Important Boundaries

- MODIFY: Tests in /testbed (this is the working directory for all your subsequent commands)
- DO NOT MODIFY: Regular source code files, configuration files (pyproject.toml, setup.cfg, etc.)

## Command Execution Rules

You are operating in an environment where

1. You issue at least one command
2. The system executes the command(s) in a subshell
3. You see the result(s)
4. You write your next command(s)

Each response should include:

1. **Reasoning text** where you explain your analysis and plan
2. At least one tool call with your command

**CRITICAL REQUIREMENTS:**

- Your response SHOULD include reasoning text explaining what you're doing
- Your response MUST include AT LEAST ONE bash tool call. You can make MULTIPLE tool calls in a single
response when the commands are independent (e.g., searching multiple files, reading different parts of
the codebase).
- Directory or environment variable changes are not persistent. Every action is executed in a new
subshell.
- However, you can prefix any action with `MY_ENV_VAR=MY_VALUE cd /path/to/working/dir && ...` or write
/load environment variables from files

Example of a CORRECT response:
<example_response>
I need to understand the Builder-related code. Let me find relevant files and check the project
structure.

[Makes multiple bash tool calls: {"command": "ls -la"}, {"command": "find src -name '*.java' | grep -i
builder"}, {"command": "cat README.md | head -50"}]
</example_response>

## Environment Details

- You have a full Linux shell environment
- Always use non-interactive flags (-y, -f) for commands
- Avoid interactive tools like vi, nano, or any that require user input
- You can use bash commands or invoke any tool that is available in the environment
- You can also create new tools or scripts to help you with the task
- If a tool isn't available, you can also install it

## Submission

When you've completed your work, you MUST submit your changes as a git patch.
Follow these steps IN ORDER, with SEPARATE commands:

Step 1: Create the patch file
Run `git diff -- path/to/file1 path/to/file2 > patch.txt` listing only the source files you modified.
Do NOT commit your changes.

<IMPORTANT>
The patch must only contain changes to the specific source files you modified to fix the issue.
Do not submit file creations or changes to any of the following files:

- tools that you created
- installation, build, packaging, configuration, or setup scripts unless they are directly part of the
issue you were fixing (you can assume that the environment is already set up for your client)
- binary or compiled files
```

````
</IMPORTANT>

Step 2: Verify your patch
Inspect patch.txt to confirm it only contains your intended changes and headers show `--- a/` and `+++
b/` paths.

Step 3: Submit (EXACT command required)
You MUST use this EXACT command to submit:

```bash
echo COMPLETE_TASK_AND_SUBMIT_FINAL_OUTPUT && cat patch.txt
```

If the command fails (nonzero exit status), it will not submit.

<CRITICAL>
- Creating/viewing the patch and submitting it MUST be separate commands (not combined with &&).
- If you modify patch.txt after verifying, you SHOULD verify again before submitting.
- You CANNOT continue working (reading, editing, testing) in any way on this task after submitting.
</CRITICAL>
</instructions>
````

SWE-Agent SWE-Agent [Yang et al., 2024] equips the LLM with direct access to a limited shell environment through an agent-computer interface (ACI), providing specialized tools for file searching, viewing, and editing. Unlike more structured agents, SWE-Agent provides minimal guardrails and lets the LLM freely interact with the environment over multiple turns. For the test case generation task, we adopt the prompt from SWT-Bench [Mündler et al., 2024], which adapts the original SWE-Agent instructions to focus on generating fail-to-pass unit tests rather than fixing the issue. The agent is instructed to first reproduce the bug, then write tests that fail on the current codebase but pass once the issue is resolved, and verify the tests fail as expected before submitting. The full prompt is shown below.

SWE-Agent Prompt

```
You are a helpful assistant that can interact with a computer to solve tasks.

<uploaded_files>
{{working_dir}}
</uploaded_files>
I've uploaded a python code repository in the directory {{working_dir}}. We have
received following issue within our repository. Here's the issue text:

<pr_description>
{{problem_statement}}
</pr_description>

INSTRUCTIONS:
Now, you're going to create unit tests that cover the issue. In other words, you should
write unit tests that fail in the current state of the repository but will pass when the
issue has been resolved. Essentially, you'll want to write a unit test that reproduces
the described issue.
Your terminal session has started and you're in the repository's root directory. You can
use any bash commands or the special interface to help you. Edit all the files you need
to and run any checks or tests that you want.
Remember, YOU CAN ONLY ENTER ONE COMMAND AT A TIME. You should always wait for feedback
after every command.
When you're satisfied with all of the changes you've made, you can submit your changes
to the code base by simply running the submit command.
Note however that you cannot use any interactive session commands (e.g. python, vim) in
this environment, but you can write scripts and run them. E.g. you can write a python
script and then run it with python <script_name>.py.

NOTE ABOUT THE EDIT COMMAND: Indentation really matters! When editing a file, make sure
to insert appropriate indentation before each line!

IMPORTANT TIPS:
1. Always start by trying to replicate the bug that the issues discusses.
   If the issue includes code for reproducing the bug, we recommend that you re-implement
   that in your environment, and run it to make sure you can reproduce the bug.
   Then start trying to fix it.
   When you think you've fixed the bug, re-run the bug reproduction script to make sure
   that the bug has indeed been fixed.
   If the bug reproduction script does not print anything when it successfully runs, we
```

```
   recommend adding a print("Script completed successfully, no errors.") command at the
   end of the file, so that you can be sure that the script indeed ran fine all the way
   through.

2. If you run a command and it doesn't work, try running a different command. A command
   that did not work once will not work the second time unless you modify it!

3. If you open a file and need to get to an area around a specific line that is not in
   the first 100 lines, say line 583, don't just use the scroll_down command multiple
   times. Instead, use the goto 583 command. It's much quicker.

4. If the bug reproduction script requires inputting/reading a specific file, such as
   buggy-input.png, and you'd like to understand how to input that file, conduct a
   search in the existing repo code, to see whether someone else has already done that.
   Do this by running the command: find_file "buggy-input.png". If that doesn't work,
   use the linux 'find' command.

5. Always make sure to look at the currently open file and the current working directory
   (which appears right after the currently open file). The currently open file might be
   in a different directory than the working directory! Note that some commands, such as
   'create', open files, so they might change the current open file.

6. When editing files, it is easy to accidentally specify a wrong line number or to
   write code with incorrect indentation. Always check the code after you issue an edit
   to make sure that it reflects what you wanted to accomplish. If it didn't, issue
   another command to fix it.

7. After having applied your changes and before submitting, make sure to run pytest and
   check if the code fails as expected due to the issue description. If it doesn't,
   revisit your code changes and adapt them accordingly.
Table
```

Claude Code Claude Code is Anthropic's official agentic coding assistant that operates via a curated set of file system and shell tools, including file read/write, bash execution, and search. For the test case generation task, we run Claude Code inside the Docker containers provided by SWT-Bench, where the repository is pre-configured at `/testbed`. We provide the agent with the issue description and optional hints, and instruct it to write fail-to-pass unit tests without modifying any source code. The final test patch is extracted via `git diff` and evaluated under the same harness. The full prompt is shown below.

Claude Code Prompt

```
<uploaded_files>
/testbed
</uploaded_files>
I've uploaded a python code repository in the directory /testbed. We have received the
following issue within our repository. Here's the issue text:

<pr_description>
{problem_statement}
</pr_description>

<hints>
{hints_text}
</hints>

INSTRUCTIONS:
Now, you're going to create unit tests that cover the issue. In other words, you should
write unit tests that fail in the current state of the repository but will pass when the
issue has been resolved. Essentially, you'll want to write a unit test that reproduces
the described issue.

Your terminal session has started and you're in the repository's root directory. You can
use any bash commands or the special interface to help you. Edit all the files you need
to and run any checks or tests that you want.

Note however that you cannot use any interactive session commands (e.g. python, vim) in
this environment, but you can write scripts and run them. E.g. you can write a python
script and then run it with python <script_name>.py.

The project's Python environment and dependencies are available in this environment.
You can run tests and Python commands directly.

IMPORTANT TIPS:
```

```
1. Always start by trying to replicate the bug that the issue discusses.
   If the issue includes code for reproducing the bug, we recommend that you re-implement
   that in your environment, and run it to make sure you can reproduce the bug.
   Then start trying to fix it.
   When you think you've fixed the bug, re-run the bug reproduction script to make sure
   that the bug has indeed been fixed.

   If the bug reproduction script does not print anything when it successfully runs, we
   recommend adding a print("Script completed successfully, no errors.") command at the
   end of the file, so that you can be sure that the script indeed ran fine all the way
   through.

2. If you run a command and it doesn't work, try running a different command. A command
   that did not work once will not work the second time unless you modify it!

3. After having applied your changes, make sure to run the relevant tests and check if
   the code fails as expected due to the issue description. If it doesn't, revisit your
   code changes and adapt them accordingly.

4. Do NOT modify any source code files -- only modify or create test files.

5. Do NOT run `git add` or `git commit`. Do NOT reset or checkout any files.

6. When you are done, run `git diff` to generate the final patch. Make sure the diff is
   non-empty.
```

Aider and Terminus-2 Aider and Terminus-2 [Merrill et al., 2026] are both evaluated using the Harbor benchmarking framework [Harbor Framework Team, 2026] with default configurations.

C.2 Evaluation Setup and Metrics

Evaluation Harness We adopt the Docker-based evaluation harness from SWT-Bench [Mündler et al., 2024]. For each instance, the generated test patch is applied to the original codebase C via `git apply`. The contributed tests are then executed on C (expected to fail) and on the fixed codebase C' obtained by applying the golden patch (expected to pass). A test suite T_{pred} is considered successful if at least one $t_i \in T_{\text{pred}}$ satisfies the fail-to-pass criterion and no test transitions from passing to failing.

Metrics Here we provide formal definitions supplementing the metric descriptions in the main text.

Success Rate $\mathcal{S}$ is the fraction of instances for which T_{pred} successfully reproduces the issue. A generated test $t \in T_{\text{pred}}$ is considered fail-to-pass ($F \to P$) if it fails on the original codebase but passes after applying the golden patch: $\text{exec}(t, C) = F$ and $\text{exec}(t, C') = P$. A test suite T_{pred} reproduces the issue if it contains at least one $F \to P$ test and no test fails after the fix is applied (i.e., no $\times \to F$ test).

Mean Coverage Let $\mathcal{X}_r^*$ and $\mathcal{X}_a^*$ denote the executable removed and added lines of the golden patch, respectively, where a line is executable if it is executed by either the original test suite or the golden tests on both C and C'. $\mathbf{Cov.}$ measures the fraction of these lines covered by T_{pred}:

$$\mathbf{Cov.} = \frac{|\{l \in \mathcal{X}_r^* \mid C^C_{T_{\text{pred}}}(l) > 0\}| + |\{l \in \mathcal{X}_a^* \mid C^{C'}_{T_{\text{pred}}}(l) > 0\}|}{|\mathcal{X}_r^*| + |\mathcal{X}_a^*|} \tag{2}$$

where $C^C_T(l)$ denotes the number of times line l is executed when running T on codebase C. Instances with $|\mathcal{X}_r^*| + |\mathcal{X}_a^*| = 0$ are excluded from the coverage analysis.

Change Coverage Let T_R denote the original test suite of the repository. $\mathbf{\Delta C}$ further restricts to lines in $\mathcal{X}_r^*$ and $\mathcal{X}_a^*$ not already covered by T_R:

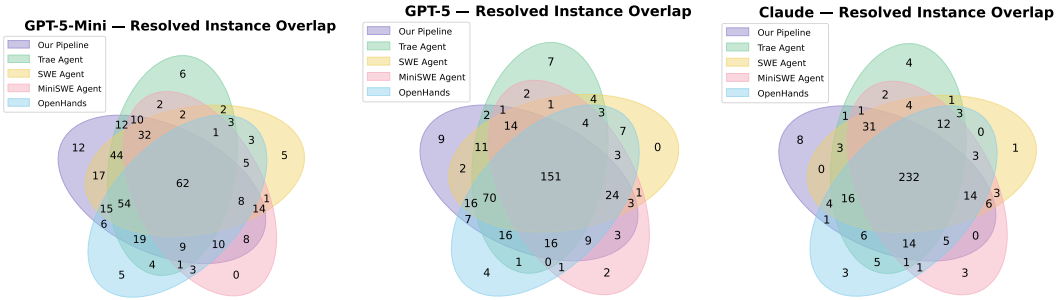
$$\mathbf{\Delta C} = \frac{|\{l \in \mathcal{X}_r^* \mid C^C_{T_R \cup T_{\text{pred}}}(l) > C^C_{T_R}(l)\}| + |\{l \in \mathcal{X}_a^* \mid C^{C'}_{T_R \cup T_{\text{pred}}}(l) > C^{C'}_{T_R}(l)\}|}{|\mathcal{X}_r^*| + |\mathcal{X}_a^*|} \tag{3}$$

TDD Coverage $\mathbf{TDD}$ credits coverage only from successful instances:

$$\mathbf{TDD} = \frac{1}{N} \sum_{i:\, \mathcal{S}_i = 1} Cov._i \tag{4}$$

Table 6: Repository-level pass rate on SWT-Bench Verified.

Model	Method	django	sympy	matplotlib	sphinx	scikit-learn	astropy	pytest	xarray	pylint	requests	seaborn	flask
GPT-5-Mini	Aider	31.5	46.6	65.6	32.1	70.8	52.9	40.0	46.7	33.3	50.0	0.0	0.0
	MiniSWE Agent	36.6	11.0	65.6	32.1	95.8	29.4	60.0	73.3	33.3	25.0	0.0	0.0
	SWE Agent	55.9	65.8	82.3	27.4	83.3	68.6	77.8	62.2	27.8	58.3	33.3	33.3
	Terminus-2	47.2	43.8	31.2	14.3	37.5	41.2	73.3	40.0	0.0	75.0	0.0	100.0
	Trae Agent	61.6	61.6	75.0	25.0	87.5	64.7	40.0	80.0	16.7	25.0	50.0	100.0
	Ours	**74.8**	**75.8**	**85.4**	**41.7**	**94.4**	**84.3**	**80.0**	**68.9**	**27.8**	**66.7**	**50.0**	**100.0**
GPT-5	Aider	28.7	45.2	34.4	10.7	66.7	52.9	46.7	60.0	0.0	25.0	0.0	0.0
	MiniSWE Agent	48.6	52.1	75.0	46.4	70.8	58.8	86.7	73.3	16.7	75.0	0.0	0.0
	SWE Agent	73.0	71.2	93.8	42.9	95.8	70.6	86.7	80.0	16.7	25.0	50.0	0.0
	Terminus-2	47.7	47.9	43.8	21.4	37.5	41.2	40.0	46.7	16.7	75.0	0.0	0.0
	Trae Agent	73.1	64.4	78.1	50.0	91.7	58.8	80.0	73.3	33.3	50.0	0.0	0.0
	Ours	**84.7**	**83.6**	**87.5**	**53.6**	**95.8**	**88.2**	**93.3**	**80.0**	**0.0**	**50.0**	**50.0**	**0.0**
Claude-Opus-4.5	Aider	49.5	52.1	46.9	28.6	83.3	47.1	33.3	53.3	16.7	25.0	50.0	0.0
	MiniSWE Agent	76.9	80.8	90.6	50.0	95.8	70.6	73.3	80.0	33.3	75.0	0.0	100.0
	OpenHands	84.3	80.8	87.5	0.0	91.7	0.0	80.0	80.0	16.7	50.0	50.0	100.0
	SWE Agent	77.8	79.5	87.5	57.1	87.5	76.5	73.3	80.0	16.7	75.0	50.0	100.0
	Terminus-2	73.6	64.4	75.0	46.4	95.8	64.7	40.0	66.7	0.0	50.0	0.0	0.0
	Trae Agent	78.7	83.6	90.6	57.1	91.7	64.7	80.0	80.0	0.0	25.0	50.0	100.0
	Ours	**82.9**	**75.3**	**93.8**	**46.4**	**95.8**	**70.6**	**80.0**	**86.7**	**16.7**	**25.0**	**100.0**	**100.0**



(a) GPT-5-Mini (b) GPT-5 (c) Claude-Opus-4.5

Figure 9: Overlap of resolved instances between our pipeline and the top-tier agent.

where N is the total number of instances. This penalizes systems that achieve high coverage without actually reproducing the issue.

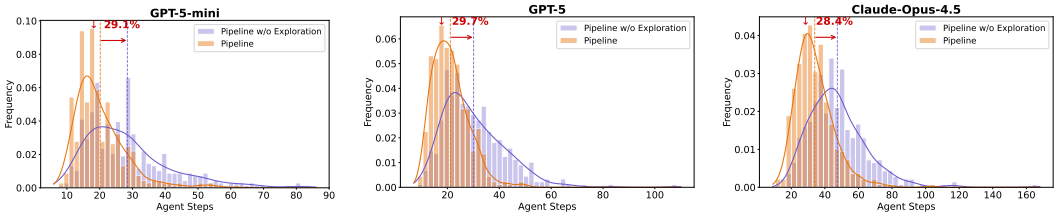
D More Analysis

Exploration Enables Precise Test Localization DPIAgent directly improves defect localization by dedicating a full phase to repository understanding before test writing begins. Beyond end-to-end reproduction success, we further evaluate whether DPIAgent correctly identifies the code regions that should be tested. To this end, we parse each ground-truth test patch via Abstract Syntax Tree (AST) analysis to extract the functions directly invoked by the golden tests, treating them as the localization ground truth. We then modify the agent's prompt to perform test localization as a standalone task and compare its predicted target functions against the ground truth using three standard metrics: Acc@k ($k \in \{1, 5\}$) measures whether at least one ground-truth target appears in the top-k predictions [Jiang et al., 2025], and Recall is defined as $|P \cap G|/|G|$, where P and G denote the predicted and ground-truth sets. Table 7 shows that DPIAgent consistently improves localization at both file and function levels across all three backbones, with the largest gains at the function level. On Claude-Opus-4.5, DPIAgent achieves 76.27% function-level Acc@5, surpassing SWE-Agent by 24.75 points, with similar gains on GPT-5-Mini (+25.67 points). The improvement is most pronounced at the function level, where precise localization requires tracing call chains across files. This is exactly where the structured exploration tools and the bug-location field in the summary provide the strongest signal.

DPIAgent Broadens Issue Coverage Beyond aggregate pass rates, we examine which specific instances each method resolves. As shown in Figure 9, DPIAgent resolves the largest number of instances uniquely missed by all baselines across all three backbones (18 on GPT-5-Mini, 16 on GPT-5, and 8 on Claude-Opus-4.5), while also broadly covering the core set of issues jointly resolved by all methods. These uniquely resolved cases typically require understanding code across multiple files, where baseline agents tend to miss key context while searching and editing in a single reactive

Table 7: Test Localization results on SWT-Bench Verified.

Model	Method	File-level			Function-level		
		Acc@1 ↑	Acc@5 ↑	Recall ↑	Acc@1 ↑	Acc@5 ↑	Recall ↑
GPT-5-Mini	SWE-Agent	59.45	82.62	66.27	37.20	52.13	37.72
	Ours	**62.71**	**91.53**	**75.12**	**58.50**	**77.80**	**55.10**
GPT-5	SWE-Agent	61.89	84.45	66.68	35.67	56.71	39.58
	Ours	**62.71**	**88.14**	**72.38**	**38.98**	**66.10**	**53.56**
Claude-opus-4.5	SWE-Agent	56.71	79.27	60.55	27.44	51.52	36.62
	Ours	**72.88**	**93.22**	**76.36**	**49.15**	**76.27**	**59.37**



(a) GPT-5-Mini (b) GPT-5 (c) Claude-Opus-4.5

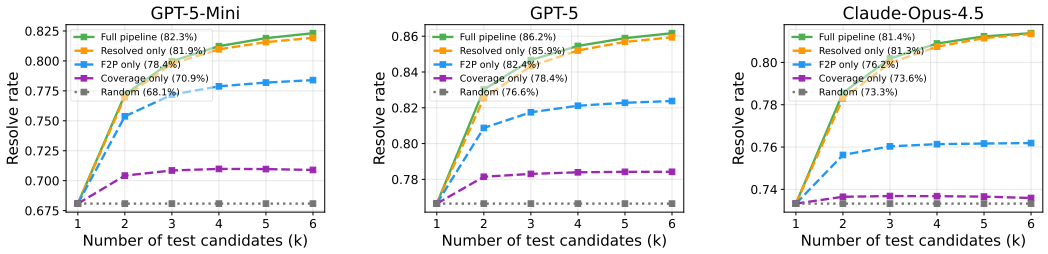
Figure 10: Distribution of agent steps for DPIAgent and the ablation without exploration across three backbones.

loop. The depth of our exploration phase makes the difference: rather than gathering just enough context to start editing, the agent traces dependencies across the repository and produces a plan that captures the full scope of relevant code, allowing the test generator to address issues that demand a complete picture rather than a local fix.

Repository-Level Analysis Table 6 breaks down the pass rate by repository. Our pipeline achieves the highest accuracy on the majority of repositories across all three backbones, with particularly strong gains on the most populated ones: django, sympy, and astropy. On scikit-learn, all configurations reach above 94%, suggesting that its test structure aligns well with automated generation. Sphinx remains the most challenging repository, where even our best configuration reaches only 53.6% (GPT-5), likely due to its complex build-time state and documentation-rendering logic. Overall, the gains from our pipeline are most pronounced on mid-to-large repositories where diverse bug types provide sufficient signal for the structured generation stages.

Tool Usage Reveals a Two-Phase Workflow To understand how our pipeline allocates exploration and generation effort, we visualize the tool call distribution across turns for all three backbones in Figure 12. Two consistent patterns emerge across backbones. First, `search_code` dominates the early turns and disappears after `ready_to_write_test` (typically around turns 8–16), reflecting our two-phase design that explicitly switches the tool set once exploration concludes; from that point on, the agent shifts to `str_replace_editor` for focused test writing. Second, Python execution remains active throughout the trajectory, suggesting that the agent continuously validates its understanding via code execution in both phases. We further observe that stronger backbones such as GPT-5 transition to test writing earlier than GPT-5-Mini, reaching sufficient repository understanding with less back-and-forth.

Exploration Tools Reduces Total Agent Steps Figure 10 shows the distribution of total agent steps for DPIAgent and the ablation without exploration across three backbones. Across all models, equipping the agent with exploration tools consistently shifts the step distribution leftward, with the median reduced by approximately 29% compared to the variant without them (29.1% for GPT-5-Mini, 29.7% for GPT-5, and 28.4% for Claude-Opus-4.5). Without these tools, the agent must rely on primitive file browsing and keyword search to understand the repository, leading to more back-and-forth navigation steps before it can write meaningful tests. In contrast, our exploration tools provide structured, semantically-aware code retrieval, allowing the agent to locate relevant code efficiently and proceed to test writing with fewer steps.



(a) GPT-5-Mini (b) GPT-5 (c) Claude-Opus-4.5

Figure 11: Ablation of selection signals. Each variant uses N=3 code patches. The resolved count is the strongest individual signal, while coverage serves primarily as a tiebreaker.

Table 8: Distribution of selected test patch sources (%) after test selection (k=6, N=3). Bold indicates the most frequently selected source for each backbone.

Model	dropCode	initPatch	initTest	simple	standard	Pipeline
`GPT-5-Mini`	**26.6**	21.1	18.3	12.0	10.0	12.0
`GPT-5`	**24.3**	21.5	16.9	11.6	12.7	13.0
`Claude-Opus-4.5`	20.4	16.4	13.9	10.2	9.7	**29.4**
Average	**23.8**	19.7	16.4	11.3	10.8	18.1

Ablation of Selection Signals We ablate the contribution of each signal in our selection pipeline (Figure 11), using N=3 code patches across all variants. The resolved count is the dominant signal, achieving within 0.3–0.4% of the full pipeline across all three backbones. The F2P filter alone provides moderate gains over random (e.g., 10.3 points on GPT-5-Mini), serving as a coarse screen that eliminates candidates showing no interaction with the code patch. Coverage ranking alone adds only 2–3 points beyond random, indicating its primary role as a tiebreaker. The full pipeline's marginal improvement over resolved-only ranking comes from F2P pre-screening, which removes low-signal candidates and helps disambiguate ties among top-ranked tests.

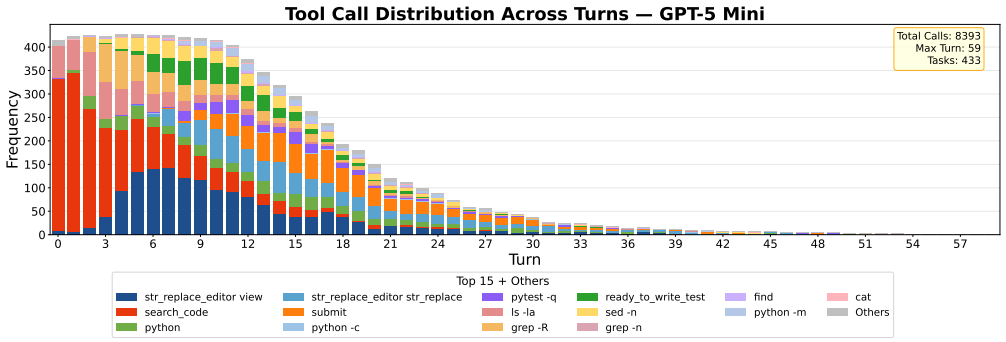
Selected Test Source Distribution Table 8 reports the distribution of selected test patches across pipeline configurations. The *dropCode* variant is the most frequently selected source for both GPT-5-Mini (26.6%) and GPT-5 (24.3%), suggesting that removing code context during test generation encourages more diverse and discriminative tests. For Claude-Opus-4.5, the full pipeline configuration dominates (29.4%), likely because its stronger baseline generation already produces high-quality tests that benefit less from constrained prompting. No single source accounts for more than 30% of selections across any backbone, confirming that diverse generation strategies contribute complementary test candidates and justifying the multi-configuration design.

E Limitations

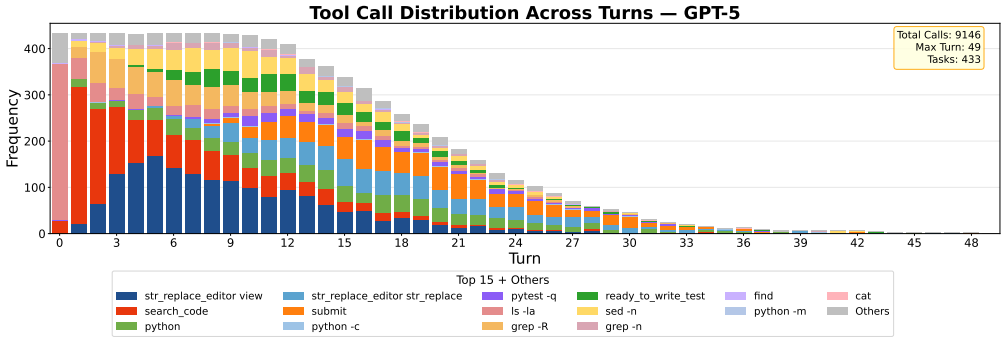
Our evaluation focuses on Python repositories from SWT-Bench; while the DPI principles are language-agnostic, validating on additional languages remains future work. Additionally, like all LLM-based approaches, inference cost scales with repository size, and further efficiency improvements are possible through more selective context management strategies.

F Broader impacts

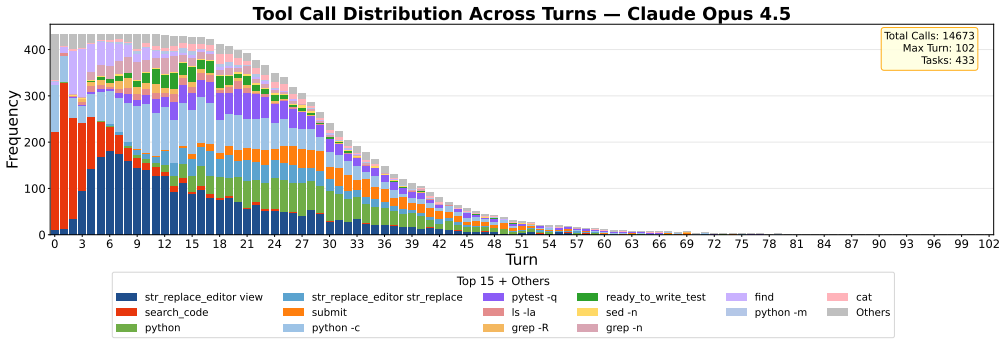
Our work focuses on automating reproduction test generation, which we believe carries a net positive societal impact by reducing developer burden and improving software reliability. We identify two potential risks and corresponding mitigations.



(a) GPT-5-Mini



(b) GPT-5



(c) Claude-Opus-4.5

Figure 12: Tool call distribution across turns for three backbones.

First, LM-generated tests may execute harmful operations during iterative verification (e.g., modifying or deleting files). To mitigate this, our pipeline runs all generated tests in isolated Docker containers, preventing generated code from affecting the host system.

Second, improved test generation could be misused to probe software for exploitable vulnerabilities. However, our system targets *known* bug reports and requires an existing issue description, limiting offensive use. Releasing our framework can help the community develop safeguards for LM-based software engineering agents, while the defensive benefits of automated testing outweigh the risks.

NeurIPS Paper Checklist

The checklist is designed to encourage best practices for responsible machine learning research, addressing issues of reproducibility, transparency, research ethics, and societal impact. Do not remove the checklist: **The papers not including the checklist will be desk rejected.** The checklist should follow the references and follow the (optional) supplemental material. The checklist does NOT count towards the page limit.

Please read the checklist guidelines carefully for information on how to answer these questions. For each question in the checklist:

- You should answer [Yes], [No], or [N/A].
- [N/A] means either that the question is Not Applicable for that particular paper or the relevant information is Not Available.
- Please provide a short (1–2 sentence) justification right after your answer (even for [N/A]).

The checklist answers are an integral part of your paper submission. They are visible to the reviewers, area chairs, senior area chairs, and ethics reviewers. You will also be asked to include it (after eventual revisions) with the final version of your paper, and its final version will be published with the paper.

The reviewers of your paper will be asked to use the checklist as one of the factors in their evaluation. While [Yes] is generally preferable to [No], it is perfectly acceptable to answer [No] provided a proper justification is given (e.g., error bars are not reported because it would be too computationally expensive" or "we were unable to find the license for the dataset we used"). In general, answering [No] or [N/A] is not grounds for rejection. While the questions are phrased in a binary way, we acknowledge that the true answer is often more nuanced, so please just use your best judgment and write a justification to elaborate. All supporting evidence can appear either in the main paper or the supplemental material, provided in appendix. If you answer [Yes] to a question, in the justification please point to the section(s) where related material for the question can be found.

IMPORTANT, please:

- **Delete this instruction block, but keep the section heading "NeurIPS Paper Checklist"**,
- **Keep the checklist subsection headings, questions/answers and guidelines below.**
- **Do not modify the questions and only use the provided macros for your answers**.

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The abstract and introduction clearly state the motivation, DPIAgent's Divide-Protocol-Isolate design, and the main empirical claims on SWT-bench Verified. These claims are supported by the methodology and experimental results reported in Sections 5.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

Justification: We discuss the limitations of the work in Appendix E.

Guidelines:

- The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
- The authors are encouraged to create a separate "Limitations" section in their paper.
- The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
- The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
- The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
- The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
- If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
- While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [N/A]

   Justification: This paper presents an empirical study and does not include any theoretical results, theorems, or formal proofs.

   Guidelines:

   - The answer [N/A] means that the paper does not include theoretical results.
   - All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
   - All assumptions should be clearly stated or referenced in the statement of any theorems.
   - The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
   - Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
   - Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: The paper describes the benchmark, baselines, evaluation metrics, backbone models, official Docker-based evaluation environment, and the number of runs used for reporting results in Sections 4 and Appendix C.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
- If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
- Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
- While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
  (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
  (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
  (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
  (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [No]

   Justification: The paper uses public benchmarks and describes the evaluation setup.

   Guidelines:

   - The answer [N/A] means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.

- At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes]

   Justification: Section 4 specifies SWT-bench Verified as the primary dataset, describes the compared baselines, defines the evaluation metrics, and states the backbone LLMs, Docker environment, default baseline configurations, and averaging over three runs.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [Yes]

   Justification: The paper reports all main results averaged over three runs and validates the findings across three different backbone LLMs, multiple baselines, ablation studies, localization analysis, and candidate-selection analysis. The consistency of gains across these settings provides appropriate evidence for the robustness of the empirical conclusions.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
   - The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
   - The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
   - The assumptions made should be given (e.g., Normally distributed errors).
   - It should be clear whether the error bar is the standard deviation or the standard error of the mean.
   - It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
   - For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
   - If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

Justification: The paper specifies that all generated tests are evaluated in the official SWT-bench Docker environment, ensuring consistent and reproducible execution across methods. The experiments use API-based backbone LLMs and repository-level Docker execution rather than model training, so the primary compute requirement is the standardized benchmark execution environment.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
- The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
- The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: The research uses existing public software-engineering benchmarks and does not involve human subjects, private user data, or sensitive personal information. To the best of our knowledge, the work conforms to the NeurIPS Code of Ethics.

   Guidelines:

   - The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [Yes]

    Justification: We discuss the broader impacts of the work in Appendix F.

    Guidelines:

    - The answer [N/A] means that there is no societal impact of the work performed.
    - If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
    - Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
    - The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
    - The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.

- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

    Answer: [N/A]

    Justification: The paper does not release a high-risk pretrained generative model, scraped dataset, or artifact with substantial dual-use risk.

    Guidelines:

    - The answer [N/A] means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: The paper properly credits the existing benchmarks, models, and systems used in the experiments, including SWT-bench, SWE-Agent, OpenHands, TraeAgent, Aider, Terminus-2, Claude Code, and the evaluated backbone LLMs. These assets are cited or referenced in the experimental setup and related work.

    Guidelines:

    - The answer [N/A] means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
    - If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
    - For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
    - If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [N/A]

Justification: The paper introduces a new method rather than a new dataset or pretrained model asset. The method itself is documented through the framework description, algorithms, and experimental setup.

Guidelines:

- The answer [N/A] means that the paper does not release new assets.
- Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
- The paper should discuss whether and how consent was obtained from people whose asset is used.
- At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

Answer: [N/A]

Justification: The paper does not involve crowdsourcing, user studies, or research with human subjects.

Guidelines:

- The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
- Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
- According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

Answer: [N/A]

Justification: The paper does not involve human subjects, so IRB approval or equivalent review is not applicable.

Guidelines:

- The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
- Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

Answer: [N/A]

Justification: LLMs were only used for writing, editing, or formatting assistance and did not affect the core methodology, scientific rigor, experimental design, results, or originality of the research.

Guidelines:

- The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
- Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.