

Do Coding Agents Still Need a Debugger?

Frontier-Model-Based Fault Localization at the Accuracy Ceiling

Christoph Daxerer
Software Competence Center
Hagenberg GmbH (SCCH)
Hagenberg, Austria
0009-0002-6397-5959

Johann Glock
Software Competence Center
Hagenberg GmbH (SCCH)
Hagenberg, Austria
0000-0002-0152-8611

Christian Macho
Department of Informatics Systems
University of Klagenfurt
Klagenfurt, Austria
0000-0001-8182-7277

Rudolf Ramler
Software Competence Center
Hagenberg GmbH (SCCH)
Hagenberg, Austria
0000-0001-9903-6107

Martin Pinzger
Department of Informatics Systems
University of Klagenfurt
Klagenfurt, Austria
0000-0002-5536-3859

Josef Pichler
School of Informatics, Communications and Media
University of Applied Sciences Upper Austria
Hagenberg, Austria
0009-0007-6908-0474

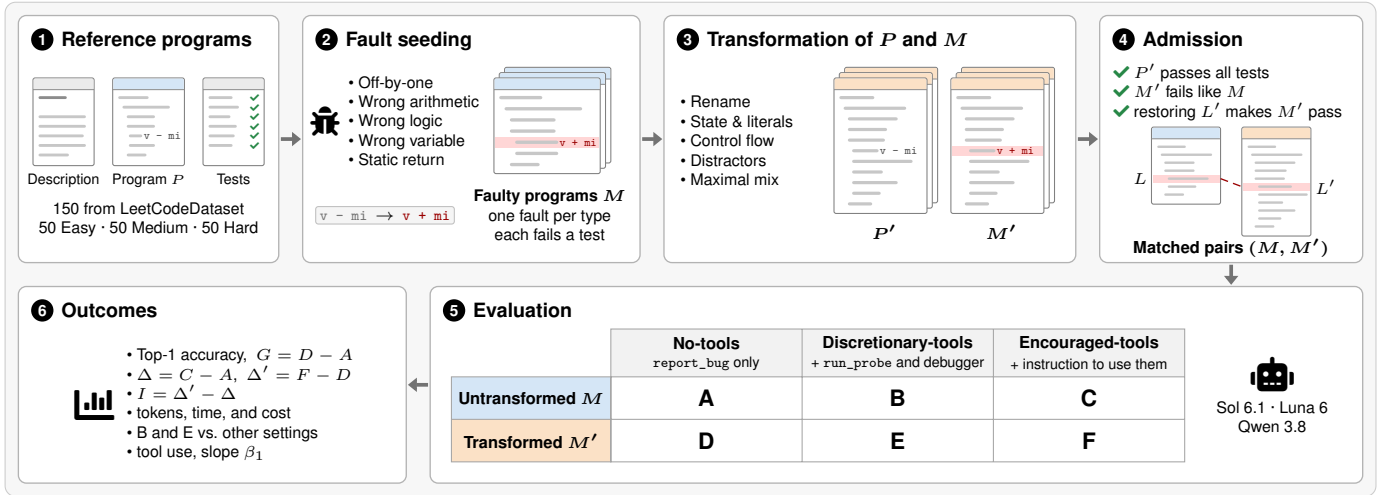


Fig. 1. Study design. From 150 LeetCodeDataset reference programs P , we seed one single-line fault per fault type to obtain faulty programs M and rewrite P and M alike with five transformation recipes. Admitted pairs (M, M') preserve the passing behavior of P , the failing assertion of M , and the restorable fault at line L' . Three LLM agents then localize every fault under three tool-use settings (Conditions A–F).

Abstract—Background: To localize a fault, a coding agent can reason from source code alone or use *debugger assistance* by running the program and stepping through it in an interactive debugger. While earlier agents struggled with static execution tracing and debugger operation, frontier LLMs have improved at both. **Objective:** We evaluate how accurately coding agents localize faults without tools across original and transformed programs, how much debugger assistance improves accuracy and at what cost, and how agents use it when optional. **Methods:** We seed 610 single-line faults into 144 LeetCodeDataset solutions and rewrite each faulty program with five recipes of behavior-preserving transformations. Agents based on Sol 6.1, Luna 6, and Qwen 3.8 localize each fault with no, optional, or encouraged debugger assistance. **Results:** Without tools, the agents localized 98.2 to 99.9% of faults in original and 96.7 to 99.9% in transformed

programs, losing accuracy mainly where recipes restructured loops and conditions. Encouraged assistance removed 63% and 61% of Luna 6’s and Qwen 3.8’s remaining transformed errors (+2.1 and +1.3 percentage points) at 14.7 to 27.0 times the token use, while Sol 6.1 stayed at 99.9%. With optional assistance, Luna 6 and Qwen 3.8 still gained +1.5 and +1.3 points, though the agents used the interactive debugger in only 0.2 to 6.2% of runs. **Conclusions:** On short, precisely specified programs, current coding agents localize nearly all faults from source code alone, and debugger assistance removes most errors that remain. Given the choice, agents executed the program far more often than they stepped through it, yet kept most of the benefit.

Index Terms—agentic software engineering, fault localization, large language models, debugging

I. INTRODUCTION

Fault localization is central to debugging, requiring a developer or coding agent to identify where a program deviates from its intended behavior [1], [2], [3]. Rather than reasoning about execution from source code alone, a coding agent can use *debugger assistance* by running the program and stepping through it in an interactive debugger. Earlier large language models (LLMs) struggled to trace execution statically, predicting 46 to 81% of outputs on 3-to-13-line Python functions in CRUXEval [4], [5]. Debugging systems therefore supplied runtime evidence through recorded traces [6], [7] or interactive debuggers [8], [9], [10], [11]. Yet operating a stateful debugger proved difficult for earlier agents, and direct `pdb` access often lowered their repair rates [9], [10]. Recent reasoning models have changed both sides of this trade-off: `pdb` improved their repair success in debug-gym [9], while on CRUXEval they predict 93 to 98% of outputs from source code alone [12]. On obfuscated LeetCode bugs, optional probes and debugging raised Qwen 3.6’s repair accuracy from 78.0% to 94.8%, whereas GPT-5.4 moved only from 91.3% to 91.7% [13].

This rapid progress in model capabilities raises three questions: how accurately the latest generation of frontier models localize faults from source code alone, how sensitive that accuracy is to behavior-preserving transformations, and how much debugger assistance still improves outcomes on top of static reasoning. However, three gaps in prior evaluations leave these questions open. First, most agentic debugging studies measure end-to-end program repair [7], [9], [10], [11], which confounds fault localization with patch generation because correct patches can arise from imprecise locations [14] and plausible patches can overfit to tests [15], [16], [17]. Second, high static accuracy on public benchmarks can reflect training-data exposure [18], [19]. Behavior-preserving source transformations reduce verbatim overlap with pretraining corpora [20] and caused earlier static LLM localizers to miss 78% of the faults they had localized in untransformed programs [21], but prior work has not measured how much such transformations still affect frontier models or whether debugger assistance recovers any lost accuracy. Third, prior studies either enforce fixed runtime instrumentation [7], [22] or evaluate a single combined tool loop [9], [10], [13], without separating encouraged from optional tool use, comparing the costs of both [23], [24], or tracking whether agents given a choice step through the program in the debugger or simply run it.

We address all three gaps in a controlled experiment on self-contained algorithmic programs, the regime that originally motivated runtime feedback [4], [6], [7]. From 150 LeetCode-Dataset problems [25], each with a problem description, a verified Python *reference program*, and an executable test suite, we seed 610 single-line faults across five fault types (Fig. 1). We rewrite each reference and faulty program with five fixed recipes of behavior-preserving transformations, ranging from variable renaming and dead-code distractors to restructuring loops and conditions (3,022 transformed programs, Fig. 2). We admit only matched pairs where the transformation pre-

Untransformed	Transformed (State and literals)
1 class Solution:	1 class Solution:
2 def maxProfit(self, prices):	2 def maxProfit(self, prices):
3 ans, mi = 0, inf	3 _frag_0 = (7 - 7), inf
4 for v in prices:	4 ans, mi = _frag_0
5 - ans = max(ans, v - mi)	5 for v in prices:
6 + ans = max(ans, v + mi)	6 - _frag_1 = (v - mi)
7 mi = min(mi, v)	6 + _frag_1 = (v + mi)
8 return ans	7 _frag_2 = max(ans, _frag_1)
	8 ans = _frag_2
	9 _frag_3 = min(mi, v)
	10 mi = _frag_3
	11 return ans

Fig. 2. A seeded fault in an untransformed faulty program (left) and in its transformed counterpart (right). Lines marked — belong to the reference programs, and lines marked + to the faulty programs. The *State and literals* recipe rewrites literals and splits the faulty statement across lines 6 to 8, moving the mutated expression from line 5 to line 6.

serves the reference program’s passing behavior, reproduces the faulty program’s exact test failure, and keeps the fault restorable at the tracked line. Agents driven by Sol 6.1, Luna 6, and Qwen 3.8 then localize each fault across 65,376 agent runs (*episodes*) in three tool-use settings: *no-tools* (problem description and source code alone), *discretionary-tools* (adding `run_probe` alongside a `pdb`-style debugger for optional use), and *encouraged-tools* (providing the same tools and instructing the agent to gather runtime evidence before deciding).

From source code alone, all three agents approached the accuracy ceiling, localizing 98.2 to 99.9% of faults in untransformed programs and 96.7% (Luna 6), 97.9% (Qwen 3.8), and 99.9% (Sol 6.1) in transformed programs. Variable renaming and distractor insertion barely changed static accuracy, whereas restructuring loops and wrapping conditions in double negations lowered it by up to 3.8 percentage points in a single recipe, where most new errors blamed doubly negated conditions. Encouraging debugger assistance raised transformed accuracy to 98.8% for Luna 6 (+2.1 points, removing 63% of its remaining errors) and 99.2% for Qwen 3.8 (+1.3 points, removing 61%), with positive effects across every recipe and fault type, while Sol 6.1 stayed at 99.9%. These gains were +1.4 and +1.0 percentage points larger on transformed programs than on their untransformed counterparts, closing the transformation gap to within −0.1 to +0.1 points. However, encouraged assistance multiplied token use by 14.7 to 27.0, wall time by 5.3 to 7.7, and monetary cost by 3.4 to 5.0.

When debugger assistance was optional, the agents rarely stepped through code in the interactive debugger. Under *discretionary-tools*, they took debugger actions in only 0.2 to 6.2% of episodes on transformed programs (against 41.7 to 100.0% when encouraged) and instead ran execution probes or answered from source code alone. Even so, Luna 6 and Qwen 3.8 gained +1.5 and +1.3 points over *no-tools*, and Qwen 3.8 ran more probes on the faults where debugger assistance helped most. Encouraging debugger use added only +0.0 to +0.6 points over optional access. In this evaluation, coding agents rarely needed debugger assistance, and when given the choice, they obtained most of its benefit by running the program rather than stepping through it.

This paper makes the following contributions:

- **Static localization across transformations:** We evaluate

fault localization accuracy across 610 untransformed and 3,022 transformed programs spanning five fault types and five transformation recipes (Section IV-A).

- **Effect and cost of debugger assistance:** We measure the accuracy gain, error reduction, and resource cost of encouraged debugger assistance across original and transformed programs and three models (Section IV-B).
- **Discretionary versus encouraged use:** We quantify how often agents choose execution probes over the interactive debugger and how much of the accuracy benefit optional tool use retains at lower cost (Section IV-C).
- **Replication package:** We release our pipeline, programs, 65,376 agent traces, and analysis scripts [26].

II. RELATED WORK

a) LLM-based fault localization: Most LLM-based fault localization pipelines use fixed execution rules to filter candidates before the model reads the code. AutoFL lets an LLM retrieve the classes and methods covered by failing tests on Defects4J and BugsInPy [27], FlexFL narrows candidate methods with traditional spectrum-based and information-retrieval rankings before an LLM inspects their source [28], and SieveFL combines failing-test coverage with semantic similarity to prune 79% of candidate methods and 49% of input tokens at unchanged Top-1 accuracy [29]. SemLoc localizes faulty lines by running instrumented code as a fixed pipeline step to check model-inferred semantic constraints [22]. In PingFL, the agent inserts print statements into suspicious methods and reruns failing tests [30]. Apart from SemLoc, these approaches rank methods rather than lines or statements, and none compares lightweight script execution against interactive breakpoint stepping.

b) Source transformations and benchmark familiarity: Benchmark designers counter training-data exposure by selecting problems released after model training cutoffs, as in DebugBench [31] and LiveCodeBench [32], or by rewriting existing benchmarks. Rewrites either apply behavior-preserving code transformations, as in CodeCleaner [20], or wrap the same algorithms in new problem contexts, as in DyCodeEval [33]. Even after surface rewrites, semantic overlap with pretraining data can remain and raise benchmark scores [34]. Prior studies also use behavior-preserving transformations to test the robustness of static fault localization, program repair, and code reasoning. Haroon et al. evaluate line-level localization by ten models given only the source code and specification across 750,013 fault-localization tasks [21]. Li et al. transform 438 single-function Defects4J bugs and observe 49% fewer correct patches, with many edits targeting inserted dead code or rewritten predicates [35], and HEJ-Robust observes its largest Java repair drops after local variable renaming [36]. Conversely, variable renaming can raise output-prediction accuracy on LiveCodeBench [37] and refactoring can improve code completion [38]. On 16 bootstrap-filtered LeetCode programs, Daxerer finds that combined structural-lexical obfuscation lowers judge-approved bug fixing much more for Qwen 3.6 than for GPT-5.4 and that optional runtime

tools recover Qwen 3.6’s performance [13]. Whereas prior work evaluates static predictions or repair proposals under a single setup, we isolate fault localization on matched pairs across five transformation recipes and three tool-use settings.

c) Runtime evidence in LLM-based debugging: Debugging pipelines supply LLMs with runtime information at different levels of agent control. Fixed pipelines execute the program outside the model’s control and append the resulting outputs or traces to the prompt: Self-Debugging feeds back unit-test results alongside model-generated code explanations [6], and LDB instruments basic-block boundaries to record intermediate variable values before the model refines a generated program [7]. Agentic systems let the model issue debugger commands directly, but must guard against invalid commands and verbose outputs. ChatDBG lets GPT-4 query `pdb`, `lldb`, or `gdb` to explain crashes and propose fixes [8]. In debug-gym, exposing `pdb` on SWE-bench-Lite lowered repair rates for GPT-4o and GPT-4o-mini but improved them for reasoning models such as o1-preview, o3-mini, and Claude 3.7 Sonnet [9]. InspectCoder interposes a state-tracking middleware between the agent and `pdb` on BigCodeBench and LiveCodeBench, where direct `pdb` interaction without the middleware lowered repair success from 67.9% to 25.7% [10]. DebugHarness combines `gdb`, deterministic replay, and script-based output distillation to patch C and C++ vulnerabilities [11]. These studies evaluate end-to-end repair or explanation quality under a single tool configuration, whereas we isolate fault localization and compare no, optional, and encouraged debugger assistance across original and transformed programs.

III. STUDY DESIGN

Fig. 1 summarizes our study design. We seed single-line faults into reference programs from a public dataset and rewrite each resulting faulty program with behavior-preserving transformations. Agents localize the faults in both representations (untransformed and transformed) under three tool-use settings: no-tools, discretionary-tools, and encouraged-tools. Crossing the two representations with the three settings yields Conditions A to C on untransformed programs and Conditions D to F on transformed programs. A *localization case* pairs a faulty program with the natural-language *problem description* of the underlying LeetCode task, and an *episode* is a single agent run on a case under a given tool-use setting.

Both discretionary-tools and encouraged-tools provide debugger assistance. We compare Top-1 accuracy, episode cost, and tool use across settings. Defining the *no-tools change* (G) as transformed minus untransformed static accuracy and the *assistance effect* (Δ, Δ') as encouraged-tools minus no-tools accuracy, we address three research questions:

- **RQ1 (Static localization and transformation sensitivity):** How accurately do coding agents localize faults from source code alone, and how much do behavior-preserving source transformations change this accuracy?

TABLE I
CONSTRUCTED PROGRAMS. PROGRAMS GIVES ADMITTED OUT OF ATTEMPTED. LOC AND CC GIVE MEDIAN WITH INTERQUARTILE RANGES, AND FOR TRANSFORMED PROGRAMS THE MEDIAN CHANGE FROM THE UNTRANSFORMED FAULTY PROGRAM, IN PERCENT FOR LOC. KEPT IS THE MEDIAN SHARE OF PRESERVED FOUR-TOKEN SEQUENCES.

	Programs	LOC	CC	Kept
<i>Reference programs</i>				
Easy	50/50	8 [4–11]	3 [2–4]	
Medium	48/50	11 [7–14]	4 [3–5]	
Hard	46/50	19 [14–24]	8 [5–11]	
<i>Faulty programs, by fault type</i>				
Off-by-one	125/150	13 [9–18]	5 [3–7]	
Wrong arithmetic	104/150	13 [9–18]	5 [3–8]	
Wrong logic	108/150	14 [10–19]	5 [4–8]	
Wrong variable	132/150	12 [9–18]	4 [3–7]	
Static return	141/150	12 [8–18]	4 [3–7]	
<i>Transformed programs, change by recipe</i>				
Rename	584/610	+0% [0–0]	+0 [0–0]	40%
State & literals	610/610	+71% [50–100]	+0 [0–0]	49%
Control flow	610/610	+169% [103–234]	+1 [1–2]	51%
Distractors	610/610	+400% [377–429]	+11 [8–15]	71%
Maximal mix	608/610	+450% [347–601]	+5 [3–8]	37%

- **RQ2 (Debugger effect):** How much does encouraged debugger assistance improve localization accuracy on un-/transformed programs, and at what cost?
- **RQ3 (Discretionary use):** How often do agents use debugger assistance when it is optional, and how does this affect accuracy relative to no and encouraged assistance?

A. Localization Cases

a) Source programs: We draw reference programs from LeetCodeDataset [25], which pairs LeetCode problems with natural-language descriptions, verified Python reference solutions, and executable tests. Self-contained algorithmic programs, which prior runtime-debugging and transformation studies also evaluate [7], [21], [31], provide the dense test suites needed to verify behavior preservation and identical test failures across transformations. We sample from the dataset’s training split, which contains problems released before July 2024 according to the authors’ estimates and is intended for training code models [25], drawing 50 reference programs at random from each of the Easy, Medium, and Hard difficulty groups (150 in total, Table I).

b) Fault seeding: From each reference program P , we derive up to five faulty programs M , one per fault type. Each fault is a single edit to one expression on one line. *Off-by-one* changes a numeric literal by one, and *wrong arithmetic* replaces an arithmetic operator. *Wrong logic* swaps `and` and `or`, changes a comparison operator, flips a Boolean literal, or adds or removes a negation. *Wrong variable* replaces a variable with another variable in scope, and *static return* replaces a returned expression with a constant such as `0` or `None`. Except for wrong variable, these fault types adapt mutation operators from PIT [39], [40]. Wrong variable corresponds to the changed-identifier pattern that Kamienski et al. report among single-statement bugs in Python projects [41]. For each

reference program and fault type, we apply a randomly chosen edit of that type that fails at least one test and record the edited line L , or skip the fault type if no candidate edit qualifies.

c) Transformations: We rewrite every faulty program with five fixed recipes, each consisting of an ordered sequence of behavior-preserving operations. *Rename* replaces variable names with single letters. *State and literals* routes attribute access through `getattr` calls, rewrites numeric and Boolean literals as equivalent expressions, and splits expressions into intermediate assignments. *Control flow* converts `for` loops into explicit iterator loops, expands conditions with double negations, inserts dead code, and splits expressions. *Distractors* wraps statements in opaque predicates that always evaluate to true, inserts dead code and generic comments such as `# WARNING: do not modify`, and splits expressions. *Maximal mix* chains 14 steps of these operations and additionally extracts statement blocks into helper functions and constructs strings from character codes. Fig. 2 illustrates the State and literals recipe applied to a faulty program. Prior studies use similar operations to evaluate fault localization [21] and program repair [35] on rewritten code, to test whether LLMs reason about program semantics [37], [42], and to reduce overlap with training corpora [20], [43].

d) Admission: We apply each recipe to P and M alike to obtain P' and M' , track the mutated expression to identify its new line L' in M' , and admit (M, M') as a *matched pair* only if P' passes all tests, M' fails on the exact same test assertion as M , and restoring the original expression at L' causes M' to pass all tests. On the dataset’s test suites, which have a median of 99 assertions per problem, these checks verify that the recipe preserves tested behavior, that the fault causes the same failure, and that line L' holds the fault.

Table I characterizes the constructed programs using non-empty lines of code including comments (LOC), McCabe’s cyclomatic complexity (CC) [44], and the share of untransformed four-token sequences preserved after transformation (Kept), adapting CodeCleaner’s n-gram overlap ratio [20]. In total, admission retained 144 of 150 reference programs, 610 of 750 seeded faults (104 to 141 per fault type), and 3,022 of 3,050 transformed programs. Admitted reference programs have median LOC of 8, 11, and 19 and median CC of 3, 4, and 8 across Easy, Medium, and Hard. Across recipes, median changes ranged from +0% (Rename) to +450% (Maximal mix) in LOC and +0 to +11 (Distractors) in CC, while median Kept ranged from 37% (Maximal mix) to 71% (Distractors).

B. Agent Interaction

a) Task: Each episode begins with a system prompt and a task message (Listing 1). The task message provides the problem description, the faulty program with 1-based line numbers, and shared supporting code from the dataset, such as standard imports and data-structure definitions. It concludes by instructing the agent to report the single most likely affected line. The agent submits its answer by calling `report_bug` with the line number and an explanation. Evaluation tests are withheld from the agent in all settings.

System prompt:

You are a Python debugging agent. Find the injected defect in the supplied implementation. You do not have access to hidden evaluation tests.

Appended in encouraged-tools:

Actively use the probe and debugger tools to gather runtime evidence before deciding. Inspect relevant state and execution flow.

End of task message:

The implementation contains one injected bug. Locate it and call 'report_bug' exactly once. Provide the single most likely affected line's 1-based line number.

Listing 1. Prompt text shared by all tool-use settings, with the instruction that encouraged-tools appends to the system prompt.

b) Tool-use settings: Each setting adds one element to the previous one. In no-tools, `report_bug` is the only available tool, so the agent must localize the fault from the problem description and source code alone. Discretionary-tools adds tools to execute and debug the program along with their documentation. Using `run_probe`, the agent can execute a custom Python script, for example one that invokes the target method on chosen inputs, and observe its output and any raised exception. The agent can also launch such a script in an interactive debugger, set breakpoints on program lines, and step into, over, or out of function calls. Whenever execution pauses, the agent can inspect local variables, the call stack, program output, and the values of arbitrary Python expressions. Encouraged-tools provides the same toolset and appends a two-sentence instruction directing the agent to gather runtime evidence before deciding (Listing 1).

c) Prompt design: Aside from the appended instruction in encouraged-tools, the prompt text is identical across all settings, avoiding prompt-wording variations that can alter LLM performance [45], [46]. The prompts specify the task and required output format without prescribing a debugging strategy, leaving tool selection to the model. A 100-episode pilot run verified that the agents invoked the tools and called `report_bug` as specified. Following reporting guidelines for LLM studies [47], [48], our replication package [26] provides the tool descriptions and full prompts.

d) Models and limits: We evaluate Sol 6.1, Luna 6, and Qwen 3.8, running two independent episodes per model, case, and setting. Sol 6.1 and Luna 6 are accessed through OpenAI’s Responses API under the identifiers `gpt-6.1-sol` and `gpt-6-luna` at the provider’s default reasoning effort (medium). Qwen 3.8 runs from its 27-billion-parameter FP8 checkpoint (Qwen/Qwen3.8-27B-FP8) on our own vLLM server with thinking enabled at the model’s default reasoning effort (xhigh). All models use default sampling parameters. We collected all episodes between October 6 and 14, 2026. Each untransformed case is evaluated once and shared across its matched pairs (65,376 episodes in total). An episode terminates when the agent calls `report_bug`. If an agent responds without calling any tool or reaches 50 tool rounds, where a round is one model response and its requested tool calls, it receives one final prompt requiring a `report_bug`

call. Across conditions, at most 0.3% of episodes needed this fallback, and at most 0.1% ended without an answer.

C. Measurements

a) Localization: We grade each reported line under two criteria. Statement-level grading accepts any line belonging to the faulty statement, which a transformation recipe can split across multiple lines, as in lines 6 to 8 of Fig. 2. Recipes split the faulty statement in 47% of transformed cases. Line-level grading accepts only the exact line holding the mutated expression (L or L'). We measure Top-1 accuracy as the share of episodes that successfully localize the fault, counting episodes without an answer as unsuccessful.

b) Tool use and cost: Beyond `report_bug`, agents can invoke two categories of runtime tools. Probe runs execute agent-written scripts through `run_probe` without the debugger. Debugger actions encompass all calls that start or stop a debug session, manage breakpoints, step through execution, or inspect the paused program state. For every episode, we record total tokens, wall-clock time, and monetary cost at the providers’ list prices as of October 7, 2026, accounting for cached-input rates. For self-hosted Qwen 3.8, we estimate monetary cost from DeepInfra’s list prices [49].

D. Analysis

We analyze each model separately. For a case X , let $a_{\text{no}}(X)$, $a_{\text{disc}}(X)$, and $a_{\text{enc}}(X)$ denote Top-1 accuracy under no-tools, discretionary-tools, and encouraged-tools.

a) RQ1 and RQ2: To measure how transformations affect static accuracy, the no-tools change (G) averages the per-pair difference over all matched pairs $\mathcal{E}$:

$$G = \frac{1}{|\mathcal{E}|} \sum_{(M, M') \in \mathcal{E}} (a_{\text{no}}(M') - a_{\text{no}}(M)),$$

while the assistance effects Δ and Δ' average $a_{\text{enc}}(M) - a_{\text{no}}(M)$ and $a_{\text{enc}}(M') - a_{\text{no}}(M')$ over the same pairs $\mathcal{E}$. RQ1 examines $a_{\text{no}}(M)$, $a_{\text{no}}(M')$, and G across all programs and within each transformation recipe and fault type, along with the error patterns on transformed programs. Because an agent can improve by at most its no-tools error rate, RQ2 examines Δ and Δ' alongside the share of no-tools errors they remove, $\Delta/(1 - \bar{a}_{\text{no}})$ and $\Delta'/(1 - \bar{a}'_{\text{no}})$. It also evaluates the interaction difference $I = \Delta' - \Delta$, which is positive when debugger assistance yields a larger gain on transformed than on untransformed programs, and the ratios of mean tokens, wall time, and cost between encouraged-tools and no-tools.

b) RQ3: RQ3 investigates how often agents take debugger actions and run probes under discretionary-tools and compares their debugger use with encouraged-tools, which checks that the instruction changes agent behavior. It then compares discretionary-tools against no-tools and encouraged-tools through episode cost and the mean per-case accuracy differences $a_{\text{disc}}(X) - a_{\text{no}}(X)$ and $a_{\text{disc}}(X) - a_{\text{enc}}(X)$ across the cases of each representation. To test whether agents invoke tools more often on cases where debugger assistance raises accuracy more, let $u(X)$ denote the mean number of probe

runs, or of debugger actions, across the discretionary-tools episodes of case X . Let $r(X)$ be the fractional rank of $u(X)$ across all cases for a model and representation, scaled from 0 to 1 with average ranks for ties. For each model, representation, and tool category, we fit

$$a_{\text{enc}}(X) - a_{\text{no}}(X) = \beta_0 + \beta_1 r(X) + \varepsilon(X)$$

by ordinary least squares. The slope β_1 estimates the fitted difference in the assistance effect between the bottom and top of the tool-use ranking. Using ranks prevents a few cases with many debugger actions from dominating the fit.

c) Uncertainty: We report 95% percentile confidence intervals from a nonparametric cluster bootstrap with 2,000 resamples [50]. Because cases derived from the same reference program share its problem specification and underlying algorithm, their outcomes are likely correlated, and treating them as independent observations can understate uncertainty [51]. Each bootstrap resample therefore draws whole reference programs with replacement, stratified by difficulty group, and retains all associated cases and episodes, capturing both program-level sampling variation and episode-to-episode variance within each case [50]. Every resample recomputes all summary statistics, tool-use ranks, and RQ3 regressions.

Under discretionary-tools, tool invocations can be concentrated in a small number of reference programs. A bootstrap resample that draws none of those programs leaves β_1 undefined, and computing an interval from the remaining resamples is invalid [52]. Because the number of clusters with tool use required for valid inference depends on the empirical distribution [53], [54], we validate each β_1 interval using 1,000 cluster-level placebo datasets [55]. Each placebo dataset keeps localization outcomes fixed and reassigns observed tool use within each difficulty group to randomly selected reference programs from the same group in proportion to their case counts, making tool use independent of the assistance effect within each group. We report the confidence interval for β_1 only if β_1 is defined in all 2,000 bootstrap resamples and the share of the 1,000 placebo intervals that miss the mean placebo slope is not significantly above 5% under a one-sided exact binomial test at the 5% level. Otherwise, Table V marks β_1 with a dagger ($\dagger$) or shows a dash when fewer than two reference programs have such tool use.

IV. RESULTS

A. RQ1: Static Localization and Transformation Sensitivity

Without tools, all three agents operated near the accuracy ceiling, localizing 98.2% (Luna 6), 98.8% (Qwen 3.8), and 99.9% (Sol 6.1) of faults in untransformed programs, and 96.7%, 97.9%, and 99.9% in transformed programs (Table II). Behavior-preserving transformations changed overall static accuracy by $G = -1.5$ [−2.3, −0.6] percentage points for Luna 6 and -0.9 [−1.6, −0.1] points for Qwen 3.8, while Sol 6.1 remained unchanged ($G = +0.0$ [−0.1, +0.0] points). Across the five fault types, untransformed no-tools accuracy was lowest on off-by-one (96.4% for Luna 6 and 97.2% for Qwen 3.8) and wrong-logic faults (97.7% and 97.2%, Fig. 3).

Transformations lowered Luna 6’s static accuracy most on those two fault types (to 93.8% and 94.9%, $G = -2.6$ and -2.8 points) and Qwen 3.8’s most on static-return faults (from 99.3% to 97.0%, $G = -2.2$ points).

Across the five transformation recipes, the static accuracy loss concentrated in Control flow and Maximal mix, the two recipes that rewrite `for` loops into explicit iterator loops and wrap conditions in double negations (Fig. 3). Without tools, these two recipes lowered Luna 6’s accuracy from 98.2% on the untransformed counterparts to 96.1% and 94.4% ($G = -2.1$ [−3.7, −0.7] and -3.8 [−5.7, −1.9] points) and Qwen 3.8’s accuracy from 98.8% to 96.4% and 96.7% (-2.4 [−4.0, −0.8] and -2.1 [−3.3, −0.7] points). By contrast, Rename (preserving 40% of four-token sequences, Table I), State and literals, and Distractors (increasing line count by 400% and cyclomatic complexity by 11) changed no-tools accuracy by only -1.0 to $+0.1$ points across all three models, with every 95% interval including zero.

Inspecting the failed no-tools episodes on Control flow and Maximal mix shows that 91 of Luna 6’s 116 wrong answers and 66 of Qwen 3.8’s 79 wrong answers occurred on faults that the same agent localized in both no-tools episodes on the untransformed program. Among these *new errors*, 71.4% for Luna 6 and 68.2% for Qwen 3.8 named a condition that the recipe had wrapped in a double negation, even though doubly negated conditions make up only 7.8% of the lines in Control flow programs and 4.4% in Maximal mix programs.

Answer to RQ1: Without tools, the agents localized 98.2 to 99.9% of faults in untransformed and 96.7 to 99.9% in transformed programs ($G = -1.5$ percentage points for Luna 6, -0.9 for Qwen 3.8, and $+0.0$ for Sol 6.1). Variable renaming and dead-code distractors changed static accuracy by at most -1.0 to $+0.1$ points, whereas restructuring loops and doubly negating conditions lowered it by 2.1 to 3.8 points for Luna 6 and Qwen 3.8, with 68.2 to 71.4% of new errors blaming doubly negated conditions.

B. RQ2: Debugger Effect

Encouraging debugger assistance improved accuracy for the two models not already at 99.9% and closed almost the entire static transformation gap (Table II). On untransformed programs, Luna 6 gained $\Delta = +0.7$ [+0.2, +1.4] percentage points (41% of its no-tools errors) and Qwen 3.8 gained $+0.3$ [−0.3, +1.1] points (27%). On transformed programs, where no-tools error rates were higher, their assistance effects rose to $\Delta' = +2.1$ [+1.4, +2.8] and $+1.3$ [+0.8, +1.8] points, removing 63% and 61% of their no-tools errors. These larger gains ($I = \Delta' - \Delta$ of $+1.4$ [+0.6, +2.1] and $+1.0$ [+0.1, +1.7] points) left encouraged-tools accuracy within -0.1 to $+0.1$ points between representations. Sol 6.1 did not improve in either representation ($\Delta = -0.1$ [−0.3, +0.0], $\Delta' = +0.0$ [+0.0, +0.1], and $I = +0.1$ [+0.0, +0.3] points).

For both Luna 6 and Qwen 3.8, Δ' was positive in all five transformation recipes and all five fault types, with 95% intervals strictly above zero in 9 of the ten subsets for Luna 6

TABLE II

TOP-1 LOCALIZATION ACCURACY IN PERCENT UNDER STATEMENT-LEVEL GRADING. NO-TOOLS CHANGE IS TRANSFORMED MINUS UNTRANSFORMED STATIC ACCURACY (G), EFFECT IS ENCOURAGED-TOOLS MINUS NO-TOOLS (Δ AND Δ' ON UN-/TRANSFORMED PROGRAMS), AND DIFFERENCE IS $I = \Delta' - \Delta$, ALL IN PERCENTAGE POINTS. UNTRANSFORMED ACCURACIES WEIGHT EACH PROGRAM BY ITS NUMBER OF MATCHED TRANSFORMED VERSIONS, AS G AND Δ DO. BRACKETS GIVE 95% INTERVALS. FIG. 3 BREAKS G AND Δ' DOWN BY RECIPE AND FAULT TYPE.

Model	Programs	No-tools	Discretionary	Encouraged	No-tools change (G)	Effect (Δ , Δ')	Difference ($I = \Delta' - \Delta$)
Luna 6	Untransformed	98.2	98.1	98.9		+0.7 [+0.2, +1.4]	
	Transformed	96.7	98.2	98.8	−1.5 [−2.3, −0.6]	+2.1 [+1.4, +2.8]	+1.4 [+0.6, +2.1]
Sol 6.1	Untransformed	99.9	99.8	99.8		−0.1 [−0.3, +0.0]	
	Transformed	99.9	99.9	99.9	+0.0 [−0.1, +0.0]	+0.0 [+0.0, +0.1]	+0.1 [+0.0, +0.3]
Qwen 3.8	Untransformed	98.8	98.9	99.1		+0.3 [−0.3, +1.1]	
	Transformed	97.9	99.1	99.2	−0.9 [−1.6, −0.1]	+1.3 [+0.8, +1.8]	+1.0 [+0.1, +1.7]

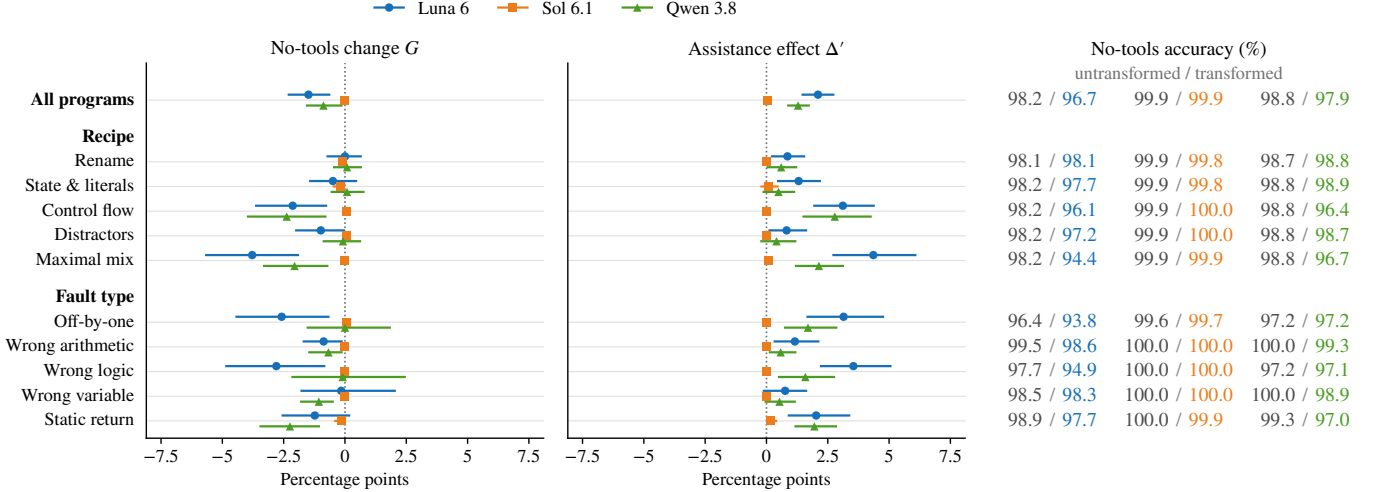


Fig. 3. No-tools change G (left, transformed minus untransformed accuracy under no-tools) and assistance effect Δ' on transformed programs (right, encouraged-tools minus no-tools) by recipe and fault type, per model, in percentage points, with 95% intervals. The rightmost column gives no-tools accuracy in percent on the matched untransformed (in gray) and transformed programs.

and 6 of ten for Qwen 3.8 (Fig. 3). Across recipes, the largest gains occurred on Control flow and Maximal mix, where encouraged assistance raised accuracy by +3.1 and +4.4 points for Luna 6 (removing 79% and 78% of the faults missed without tools, $I = +2.4$ and +3.6) and by +2.8 and +2.1 points for Qwen 3.8 (removing 77% and 65%, $I = +2.5$ and +1.8). Rename, State and literals, and Distractors still gained +0.8 to +1.3 points for Luna 6 and +0.4 to +0.6 for Qwen 3.8 ($I = +0.1$ to +0.6). Across fault types, Δ' was largest on off-by-one (+3.1 and +1.7 points), wrong-logic (+3.5 and +1.6), and static-return faults (+2.0 for both models). Across all five fault types, encouraged assistance removed 51 to 87% of Luna 6’s and 50 to 86% of Qwen 3.8’s no-tools errors.

Encouraging debugger assistance multiplied the resource cost of an episode in both representations (Table III). Without tools, episodes averaged 1.3 to 3.3 thousand tokens and 4.9 to 37.9 s on untransformed programs, and 1.7 to 5.0 thousand tokens and 5.2 to 66.4 s on transformed programs, costing 0.22 to 7.65 US dollars per 1,000 episodes. Under encouraged-tools, token usage rose to 25.0 to 41.8 thousand on untransformed and 32.2 to 73.5 thousand on transformed programs (11.2 to 29.7 and 14.7 to 27.0 times no-tools), and wall time rose to

37.8 to 213.6 s and 39.9 to 350.4 s (5.6 to 7.7 and 5.3 to 7.7 times). Monetary cost rose to 1.26 to 15.68 and 1.60 to 26.06 US dollars per 1,000 episodes (3.3 to 5.7 and 3.4 to 5.0 times, smaller than the token ratios because multi-turn tool calls reuse cached prompt prefixes). Sol 6.1 used 18.4 to 19.8 times as many tokens and 7.7 times the wall time for no accuracy gain.

Answer to RQ2: Encouraged debugger assistance raised accuracy for Luna 6 and Qwen 3.8 by $\Delta = +0.7$ and +0.3 percentage points on untransformed programs (removing 41% and 27% of no-tools errors) and by $\Delta' = +2.1$ and +1.3 points on transformed programs (63% and 61%), while Sol 6.1 stayed at 99.9%. Encouraged tools closed the static transformation gap to −0.1 to +0.1 points ($I = +1.4$ and +1.0 points, up to +3.6 on individual recipes), with the largest gains where control-flow rewrites had lowered static accuracy. On transformed programs, it multiplied tokens by 14.7 to 27.0, wall time by 5.3 to 7.7, and cost by 3.4 to 5.0.

C. RQ3: Discretionary Use

When debugger assistance was optional, the agents rarely used the interactive debugger, taking debugger actions in 0.0 to 0.9% of episodes on untransformed programs and 0.2 to

TABLE III

EPISODE COST. TOKENS AND WALL TIME ARE MEANS PER EPISODE, AND MONETARY COST IS THE MEAN COST OF 1,000 EPISODES, ESTIMATED FOR QWEN 3.8 FROM A PROVIDER’S LIST PRICES. THE LAST TWO COLUMNS DIVIDE ENCOURAGED-TOOLS BY NO-TOOLS.

Model	Measure	Untransformed			Transformed			Encouraged / no-tools	
		No-tools	Discr.	Encour.	No-tools	Discr.	Encour.	Untr.	Transf.
Luna 6	Tokens (thousands)	1.4	2.2	41.8	2.0	3.3	55.0	29.7×	27.0×
	Wall time (s)	5.9	4.5	41.2	7.5	6.4	47.0	7.0×	6.2×
	Cost (US\$ per 1,000 episodes)	0.22	0.25	1.26	0.32	0.38	1.60	5.7×	5.0×
Sol 6.1	Tokens (thousands)	1.3	3.6	25.0	1.7	4.9	32.2	19.8×	18.4×
	Wall time (s)	4.9	14.0	37.8	5.2	15.0	39.9	7.7×	7.7×
	Cost (US\$ per 1,000 episodes)	2.86	4.35	11.81	3.41	5.30	13.98	4.1×	4.1×
Qwen 3.8	Tokens (thousands)	3.3	19.0	37.4	5.0	38.3	73.5	11.2×	14.7×
	Wall time (s)	37.9	136.1	213.6	66.4	245.0	350.4	5.6×	5.3×
	Cost (US\$ per 1,000 episodes)	4.76	11.60	15.68	7.65	18.92	26.06	3.3×	3.4×

6.2% on transformed programs (Table IV). Instead, Sol 6.1 and Qwen 3.8 ran execution probes without debugger actions in 86.7 and 99.1% of untransformed episodes and 98.4 and 93.6% of transformed episodes. Luna 6 called no tool in 91.2% of untransformed and 87.9% of transformed discretionary episodes and ran only probes in 8.8 and 11.8%. Across the executed probe scripts on transformed programs, 57.9% (Qwen 3.8) to 99.8% (Sol 6.1) called the program under test, and 1.2% (Sol 6.1) to 25.6% (Qwen 3.8) checked the return value via an in-script assertion or equality test instead of only printing outputs to inspect. Encouraging tool use raised the share of episodes with a debugger action to 20.2 to 100.0% on untransformed programs and 41.7 to 100.0% on transformed programs, with Luna 6 and Sol 6.1 averaging 12.5 and 8.4 debugger actions per transformed episode while Qwen 3.8 still ran only probes in 58.1%. Among encouraged debugger actions on transformed programs, stepping accounted for 41.7% (Luna 6) and 44.0% (Sol 6.1) versus 22.9% (Qwen 3.8), breakpoints for 17.5% to 26.1%, and state inspection for 12.4% to 20.2% of debugger actions.

Although the agents rarely invoked the debugger under discretionary-tools, Luna 6 and Qwen 3.8 obtained most of their encouraged-tools accuracy gains at a fraction of the cost. On transformed programs, discretionary-tools reached 98.2% accuracy for Luna 6, 99.1% for Qwen 3.8, and 99.9% for Sol 6.1 (Table II). For Luna 6 and Qwen 3.8, this raised accuracy over no-tools by +1.5 [+0.9, +2.2] and +1.3 [+0.8, +1.8] points, within −0.6 [−1.0, −0.2] and +0.0 [−0.3, +0.2] points of encouraged-tools (Table V). On untransformed programs, where baseline accuracy was already 98.2 to 99.9%, discretionary-tools differed from no-tools by −0.2 to +0.2 points and from encouraged-tools by −0.8 to −0.1 points. Like encouraged-tools, discretionary-tools accuracy on transformed programs differed from the untransformed counterparts by only +0.1 to +0.2 points. Meanwhile, discretionary-tools episodes on transformed programs used 3.3 thousand tokens (6.4s, 0.38 US dollars per 1,000 episodes) for Luna 6, 4.9 thousand tokens for Sol 6.1, and 38.3 thousand tokens (245.0s, 18.92 US dollars) for Qwen 3.8, compared with 55.0, 32.2, and 73.5 thousand under encouraged-tools (Table III).

TABLE IV

TOOL USE IN THE TOOL-ENABLED SETTINGS ON UNTRANSFORMED AND TRANSFORMED PROGRAMS. THE FIRST TWO ROWS PER MODEL GIVE THE SHARES OF EPISODES WITH AT LEAST ONE DEBUGGER ACTION AND OF EPISODES WITH PROBE RUNS BUT NO DEBUGGER ACTION. THE NEXT TWO GIVE THE MEAN NUMBERS OF DEBUGGER ACTIONS AND PROBE RUNS PER EPISODE. THE LAST THREE GIVE THE SHARES OF DEBUGGER ACTIONS THAT MANAGE BREAKPOINTS, STEP, AND INSPECT STATE.

Model	Measure	Discretionary		Encouraged	
		Untr.	Transf.	Untr.	Transf.
Luna 6	Any debugger action	0.0%	0.3%	100.0%	100.0%
	Probe runs only	8.8%	11.8%	0.0%	0.0%
	Debugger actions	0.0	0.0	11.6	12.5
	Probe runs	0.1	0.2	1.1	1.2
	Breakpoints	–	18.4%	14.6%	17.5%
	Stepping	–	27.6%	45.8%	41.7%
	State inspection	–	13.2%	18.8%	16.0%
Sol 6.1	Any debugger action	0.0%	0.2%	100.0%	100.0%
	Probe runs only	86.7%	98.4%	0.0%	0.0%
	Debugger actions	0.0	0.0	8.0	8.4
	Probe runs	0.9	1.0	1.1	1.1
	Breakpoints	–	23.1%	20.1%	21.3%
	Stepping	–	23.1%	45.5%	44.0%
	State inspection	–	17.9%	19.7%	20.2%
Qwen 3.8	Any debugger action	0.9%	6.2%	20.2%	41.7%
	Probe runs only	99.1%	93.6%	79.8%	58.1%
	Debugger actions	0.1	0.4	1.5	3.1
	Probe runs	1.8	2.2	2.3	2.7
	Breakpoints	31.5%	26.1%	22.6%	26.1%
	Stepping	17.8%	19.2%	27.3%	22.9%
	State inspection	8.2%	11.6%	13.4%	12.4%

Qwen 3.8 also ran more probes on the cases where debugger assistance raised its accuracy more. Regressing the per-case assistance effect on the discretionary probe-run rank gives a slope β_1 of +5.3 [+3.2, +7.5] percentage points from the bottom to the top of the probe-use ranking on transformed programs (placebo miss rate 4.2%) and +1.9 [−1.0, +5.5] points on untransformed programs (Table V). Luna 6 made probe runs on 596 of the 3,022 transformed cases, with a slope estimate of +15.0 points that is reported without an interval because 6.6% of its placebo intervals missed the target. By debugger-action rank on transformed programs, Luna 6 and Qwen 3.8 had descriptive slopes of +66.7 points (across 17

TABLE V

DISCRETIONARY USE. MEAN PER-CASE ACCURACY DIFFERENCES BETWEEN DISCRETIONARY-TOOLS AND THE OTHER TWO SETTINGS, AND THE SLOPES β_1 OF THE PER-CASE ASSISTANCE EFFECT ON THE RANK OF PROBE RUNS AND OF DEBUGGER ACTIONS UNDER DISCRETIONARY-TOOLS (SECTION III-D), ALL IN PERCENTAGE POINTS. BRACKETS GIVE 95% INTERVALS. A DAGGER MARKS A SLOPE WITHOUT AN INTERVAL AND A DASH A SLOPE THAT IS NOT REPORTED.

	Discretionary minus		Slope β_1 by	
	No-tools	Encouraged	Probe runs	Debugger
<i>Luna 6</i>				
Untr.	-0.1 [-1.0, +0.7]	-0.8 [-1.6, -0.2]	+3.2 [†]	-
Transf.	+1.5 [+0.9, +2.2]	-0.6 [-1.0, -0.2]	+15.0 [†]	+66.7 [†]
<i>Sol 6.1</i>				
Untr.	-0.2 [-0.4, +0.0]	-0.1 [-0.2, +0.0]	-1.1 [†]	-
Transf.	+0.0 [-0.1, +0.0]	+0.0 [-0.2, +0.0]	+0.0 [+0.0, +0.1]	-0.1 [-0.2, +0.0]
<i>Qwen 3.8</i>				
Untr.	+0.2 [-0.3, +0.7]	-0.2 [-0.7, +0.3]	+1.9 [-1.0, +5.5]	-0.7 [-2.1, +0.7]
Transf.	+1.3 [+0.8, +1.8]	+0.0 [-0.3, +0.2]	+5.3 [+3.2, +7.5]	+2.7 [†]

cases, placebo miss rate 36.2%) and +2.7 points (miss rate 6.7%). Because encouraged-tools raised Sol 6.1’s accuracy over no-tools on 2 transformed cases and lowered it on 1, its probe and debugger slopes stayed near zero.

Answer to RQ3: When assistance was optional, the agents took debugger actions in 0.0 to 0.9% of untransformed and 0.2 to 6.2% of transformed episodes. Sol 6.1 and Qwen 3.8 mostly ran probes without the debugger, and Luna 6 called no tool in 87.9% of transformed episodes. Even so, optional assistance raised transformed accuracy over no-tools by +1.5 points for Luna 6 and +1.3 for Qwen 3.8 (versus +2.1 and +1.3 when encouraged) using 3.3 and 38.3 thousand tokens instead of 55.0 and 73.5 thousand, and Qwen 3.8 ran more probes where assistance helped more.

D. Robustness

To test sensitivity to our grading criterion, we re-evaluated all episodes under line-level grading, which accepts only the tracked line L' holding the edited expression rather than any line of a split faulty statement, such as lines 7 and 8 in Fig. 2 (Section III-C). Naming another line of a split faulty statement accounted for 7.3% of Luna 6’s no-tools answers on transformed programs and 0.1% of Sol 6.1’s. Line-level grading lowered no-tools accuracy on transformed programs to 89.4% for Luna 6 and 96.4% for Qwen 3.8 ($G = -8.8$ and -2.3 points), raising their Δ' estimates to +3.4 and +1.8 points and I to +2.6 [+1.6, +3.6] and +1.5 [+0.6, +2.3] points.

To check whether answers accepted only by statement-level grading diagnosed the seeded fault, we conducted a blinded, LLM-assisted audit (verified by an author) of 45 such answers per model across the three settings (which for Sol 6.1 were all of them) alongside 20 line-level successes per model. The agent’s explanation identified the seeded fault in 133 of the 135 statement-only answers and in 59 of the 60 line-level successes, showing that statement-only answers diagnosed the fault as often as line-level successes did.

V. DISCUSSION

a) Fault localization at the accuracy ceiling: Without tools, the three agents localized 98.2 to 99.9% of faults in untransformed and 96.7 to 99.9% in transformed programs, and neither replacing most four-token sequences in Rename nor quadrupling line count in Distractors lowered static accuracy by more than 1.0 percentage point (Section IV-A). This contrasts with earlier evaluations in which models without runtime feedback left substantial shares of faulty programs unrepaired [7], [31] and static localizers lost 78% of previously localized faults after behavior-preserving transformations [21]. Three differences in Haroon et al.’s setup may account for their larger losses: their metric conditions on solving the original program, their programs average 250 lines, and most of their models date from March 2025. Meanwhile, unlike earlier agents whose repair rates dropped under direct pdb access or unfiltered output [9], [10], encouraged debugger use never caused an accuracy collapse. Even when Luna 6 and Sol 6.1 used the interactive debugger in 100.0% of transformed episodes, forced-answer and no-answer rates stayed at or below 0.3% and 0.1%, Sol 6.1 remained at 99.9%, and Luna 6 and Qwen 3.8 removed 63% and 61% of their remaining errors. With structured tool interfaces and current reasoning models, the main limitation of debugger assistance on short, well-specified programs is no longer tool reliability, but that static reasoning already solves nearly every case.

b) Syntactic rewrites versus familiarity: The static accuracy loss from transformation did not follow how much surface code a recipe changed. Without tools, Rename did not lower accuracy despite preserving only 40% of four-token sequences, whereas Control flow and Maximal mix lowered accuracy by 2.1 to 3.8 points, with 71.4% (Luna 6) and 68.2% (Qwen 3.8) of their new errors naming a doubly negated condition (Section IV-A). Because our wrong-logic faults add or remove a negation and the prompt states that the program contains one injected bug, a static reader can mistake a doubly negated condition for the seeded fault, much as LLMs fail code-equivalence checks under simple syntactic rewrites [56], [57] and follow misleading comments [12], [58]. Running the program shows that the condition evaluates as intended, which explains why both tool-enabled settings closed almost the entire accuracy gap between representations. Before attributing an accuracy drop on transformed code to reduced training-data familiarity, benchmark designers should therefore check whether new errors cluster on specific syntactic constructs.

c) Program execution versus interactive stepping: When debugger assistance was optional, the agents obtained most of its accuracy benefit while rarely invoking the interactive debugger. Instructing agents to use the tools raised the share of transformed episodes with a debugger action from 0.2 to 6.2% up to 41.7 to 100.0%, but raised accuracy by only +0.6 points for Luna 6 and +0.0 points for Qwen 3.8 (Section IV-C). On short programs, running `run_probe` on chosen inputs appears to supply most of the runtime evidence, paralleling the gains from print debugging in PingFL [30]. In an exploratory

post-hoc split of transformed discretionary-tools episodes, probe-only runs accounted for +1.1 of Qwen 3.8’s +1.3-point gain and +0.7 of Luna 6’s +1.5 points, with no-tool runs contributing another +0.7 points for Luna 6. Because agents choose tools in response to the fault and earlier outputs, conditioning on realized tool use mixes tool effects with selection bias, and extra model turns can also aid accuracy [23]. Isolating program execution from interactive debugging therefore requires a dedicated execution-only setting.

d) Cost-effectiveness and model selection: Across models and settings, how often an agent invoked tools did not match how much accuracy it gained from them (Tables II to V). Sol 6.1 ran probes in 98.4% of its discretionary episodes on transformed programs even though it already localized 99.9% of the faults without tools, consistent with reports that LLMs call tools when internal knowledge suffices [24]. Conversely, Luna 6 called no tool in 87.9% of its discretionary episodes yet retained +1.5 of its +2.1-point gain at 3.3 thousand tokens per episode versus 55.0 thousand under encouraged-tools. Qwen 3.8 matched its encouraged accuracy (+1.3 points) at roughly half the tokens (38.3 versus 73.5 thousand) while running more probes on the faults where assistance helped more. Moreover, Sol 6.1 without tools (99.9% accuracy, 5.2s, 3.41 US dollars per 1,000 transformed episodes) was both more accurate and faster than Luna 6 or Qwen 3.8 with encouraged debugging (98.8% in 47.0s and 99.2% in 350.4s). For agent builders, multi-turn debugging scaffolding on self-contained code cannot substitute for a stronger base model and is most cost-effective when left optional rather than forced by prompt instructions. Because most executed probe scripts did not check expected outputs even though every problem description includes example input-output pairs (Section IV-C), dedicated tools that run specification examples automatically could further reduce probing overhead.

e) Grading after transformations: Statement-level and line-level grading differ only where a transformation recipe splits the faulty statement across lines (47% of transformed cases). In our audit, answers accepted only by statement-level grading diagnosed the seeded fault as often as exact-line successes, and the share of such answers varied from 0.1% for Sol 6.1 to 7.3% for Luna 6 (Section IV-D). Strict line-level grading would count these valid diagnoses as failures, inflate $|G|$ and I , and make model comparisons depend on which fragment of a split statement each model reports. Evaluations on transformed code should therefore track the full span of the faulty statement across rewrites and report gains relative to the no-tools error rate alongside percentage-point effects when baseline accuracy is near the ceiling.

VI. THREATS TO VALIDITY

a) Comparing representations: Like prior studies [20], [35], [43], we use behavior-preserving transformations to reduce surface overlap with pretraining corpora. However, Kept measures four-token overlap between program versions rather than against a model’s training corpus, transformed programs remain semantic duplicates of public solutions [34], and the

problem description stays unchanged because fault localization requires a specification of intended behavior [21]. An agent can therefore still recognize an underlying LeetCode problem seen during training [32]. Residual familiarity makes the two representations more alike and attenuates any part of G or I driven by unfamiliarity, so we interpret G and I as sensitivity to the transformation recipes rather than to unfamiliarity alone.

b) Measuring the assistance effect: Compared with no-tools, encouraged-tools adds program execution, the interactive debugger, an explicit tool-use instruction, and additional model calls, so Δ and Δ' measure their combined effect. Top-1 accuracy grades the reported line number rather than the explanation, and LLMs can reach correct answers from flawed reasoning [37]. Coincidental hits would bias Δ or Δ' only if they occurred at different rates across settings, and our blinded audit found valid fault diagnoses in 59 of 60 exact-line and 133 of 135 statement-only successes (Section IV-D). The prompt asks for the “most likely affected line” (Listing 1), which an agent could interpret as a downstream line where the fault manifests, though all settings share this prompt. Finally, monetary costs depend on provider pricing and cached-input discounts, and our DeepInfra-based estimate for self-hosted Qwen 3.8 is an external proxy, so we report token counts and wall time alongside dollar costs for every condition.

c) Discretionary use: An agent may invoke a tool without expecting an accuracy gain [24], for example because a program looks complex. The RQ3 slope β_1 therefore measures whether discretionary tool use is associated with larger per-case assistance effects, not why the agent used a tool. With two episodes per case and setting, per-case noise can attenuate β_1 toward zero, so we draw no case-level conclusions.

d) Generalization: We evaluate short Python programs for precisely specified algorithmic problems because constructing verified matched pairs requires a problem description, a reference solution, and an executable test suite. Our findings do not generalize directly to multi-file repositories, incomplete specifications, or multi-statement defects, where the relevant state may not fit in the model’s context and interactive debugging has shown larger gains [9], [11]. Four of our five fault types instantiate single-statement bug patterns that account for a quarter of the 73,013 Python bugs mined by Kamienski et al. [41], and 73% of 357 real Java faults in Just et al. were coupled to mutants of common operators [59].

e) Reproducibility: Commercial API updates can alter model behavior over time [48]. We record model identifiers, reasoning settings, and collection dates (Section III-B), release all prompts, traces, and analysis scripts [26], and serve Qwen 3.8 ourselves from fixed open-weights checkpoints.

VII. CONCLUSION

Across 610 untransformed and 3,022 transformed Python programs (65,376 episodes), we evaluated how accurately coding agents localize single-line faults from source code alone and how much debugger assistance improves outcomes. Without tools, agents based on Sol 6.1, Luna 6, and Qwen 3.8 already approached the accuracy ceiling, localizing 98.2 to

99.9% of faults in untransformed and 96.7 to 99.9% in transformed programs. Variable renaming and distractors barely changed static accuracy, whereas restructured loops and doubly negated conditions lowered it for Luna 6 and Qwen 3.8. Encouraged debugger assistance removed 63% and 61% of their remaining transformed errors (+2.1 and +1.3 percentage points) and closed the transformation gap, while multiplying token use by 14.7 to 27.0 and wall time by 5.3 to 7.7. When debugger assistance was optional, the agents used the interactive debugger in only 0.2 to 6.2% of transformed episodes and instead ran execution probes or answered from source code alone, yet kept most of the accuracy benefit at a fraction of the token cost. Future work should isolate program execution from interactive debugging with an execution-only setting and evaluate larger, less precisely specified repository bugs where static reasoning leaves more room for state inspection.

DATA AVAILABILITY

Our replication package [26] provides our pipeline, programs, prompts, episode traces, and analysis scripts.

ACKNOWLEDGMENT

The research reported in this paper has been partly funded by the Federal Ministry for Innovation, Mobility and Infrastructure (BMIMI), the Federal Ministry for Economy, Energy and Tourism (BMWET), and the State of Upper Austria in the frame of the SCCH competence center INTEGRATE (FFG grant no. 892418) in the COMET - Competence Centers for Excellent Technologies Programme managed by Austrian Research Promotion Agency FFG. We used OpenAI GPT, Anthropic Claude, and Google Gemini models to assist with writing, coding, and literature search. The authors reviewed and refined the generated material and take full responsibility for the final manuscript.

REFERENCES

- [1] C. Parnin and A. Orso, “Are automated debugging techniques actually helping programmers?” in *Proceedings of the 20th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2011, pp. 199–209.
- [2] P. S. Kochhar, X. Xia, D. Lo, and S. Li, “Practitioners’ expectations on automated fault localization,” in *Proceedings of the 25th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2016, pp. 165–176.
- [3] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller, “Evaluating and improving fault localization,” in *Proceedings of the 39th IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 609–620.
- [4] A. Gu, B. Rozière, H. J. Leather, A. Solar-Lezama, G. Synnaeve, and S. I. Wang, “CRUXEval: A benchmark for code reasoning, understanding and execution,” in *Proceedings of the 41st International Conference on Machine Learning (ICML)*, ser. Proceedings of Machine Learning Research, vol. 235. PMLR, 2024, pp. 16 568–16 621.
- [5] J. Chen, Z. Pan, X. Hu, Z. Li, G. Li, and X. Xia, “Reasoning runtime behavior of a program with LLM: How far are we?” in *Proceedings of the 47th IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE, 2025, pp. 1869–1881.
- [6] X. Chen, M. Lin, N. Schärli, and D. Zhou, “Teaching large language models to self-debug,” in *Proceedings of the 12th International Conference on Learning Representations (ICLR)*, 2024, pp. 8746–8825.
- [7] L. Zhong, Z. Wang, and J. Shang, “Debug like a human: A large language model debugger via verifying runtime execution step by step,” in *Findings of the Association for Computational Linguistics: ACL 2024*. Association for Computational Linguistics, 2024, pp. 851–870.
- [8] K. H. Levin, N. van Kempen, E. D. Berger, and S. N. Freund, “ChatDBG: Augmenting debugging with large language models,” *Proceedings of the ACM on Software Engineering*, vol. 2, no. FSE, pp. 1892–1913, 2025.
- [9] X. Yuan, M. M. Moss, C. El Feghali, C. Singh, D. Moldavskaya, D. MacPhee, L. Caccia, M. Pereira, M. Kim, A. Sordoni, and M.-A. Côté, “debug-gym: A text-based environment for interactive debugging,” arXiv preprint arXiv:2503.21557, 2025.
- [10] Y. Zhang, Y. Zhang, G. Li, C. Zhi, B. Li, F. Huang, Y. Li, and S. Deng, “InspectCoder: Dynamic analysis-driven self repair through interactive LLM-debugger collaboration,” *Proceedings of the ACM on Programming Languages*, vol. 10, no. OOPSLA1, pp. 1041–1069, 2026.
- [11] M. Sun, Y. Yang, X. Liu, Y. Zhou, and B. Xu, “DebugHarness: Emulating human dynamic debugging for autonomous program repair,” arXiv preprint arXiv:2604.03610, 2026.
- [12] M. H. Lam, C. Wang, J. tse Huang, and M. R. Lyu, “CodeCrash: Exposing LLM fragility to misleading natural language in code reasoning,” in *Advances in Neural Information Processing Systems*, vol. 38. Neural Information Processing Systems Foundation, 2025, pp. 133 886–133 932.
- [13] C. Daxerer, “Effects of obfuscation and dynamic debugging contexts on bug fixing with large language models,” Master’s thesis, University of Applied Sciences Upper Austria, 2026.
- [14] S. B. Hossain, N. Jiang, Q. Zhou, X. Li, W.-H. Chiang, Y. Lyu, H. Nguyen, and O. Tripp, “A deep dive into large language models for automated bug localization and repair,” *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 1471–1493, 2024.
- [15] E. K. Smith, E. T. Barr, C. Le Goues, and Y. Brun, “Is the cure worse than the disease? Overfitting in automated program repair,” in *Proceedings of the 10th ACM Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. ACM, 2015, pp. 532–543.
- [16] Z. Qi, F. Long, S. Achour, and M. Rinard, “An analysis of patch plausibility and correctness for generate-and-validate patch generation systems,” in *Proceedings of the 24th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2015, pp. 24–36.
- [17] Y. Wang, M. Pradel, and Z. Liu, “Are ‘solved issues’ in SWE-bench really solved correctly? An empirical study,” in *Proceedings of the 48th IEEE/ACM International Conference on Software Engineering (ICSE)*. ACM, 2026, pp. 169–181.
- [18] M. Riddell, A. Ni, and A. Cohan, “Quantifying contamination in evaluating code generation capabilities of language models,” in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (ACL)*. Association for Computational Linguistics, 2024, pp. 14 116–14 137.
- [19] D. Ramos, C. Mamede, K. Jain, P. Canelas, C. Gamboa, and C. Le Goues, “Are large language models memorizing bug benchmarks?” in *Proceedings of the 2nd IEEE/ACM International Workshop on Large Language Models for Code (LLM4Code)*. IEEE, 2025, pp. 1–8.
- [20] J. Cao, S. Chen, W. Zhang, H. C. Lo, Y. Li, and S.-C. Cheung, “CodeCleaner: Mitigating data contamination for LLM benchmarking,” in *Proceedings of the 16th International Conference on Internetware (Internetware)*. ACM, 2025, pp. 71–83.
- [21] S. Haroon, A. F. Khan, A. Humayun, W. Gill, A. H. Amjad, A. R. Butt, T. Khan, and M. A. Gulzar, “Assessing the impact of code changes on the fault localizability of large language models,” in *Proceedings of the 19th IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2026, pp. 285–296.
- [22] Z. Yang, Q. Zhang, R. Gupta, and A. Kundu, “From syntactic to semantic spectra for fault localization,” in *Companion Proceedings of the 34th ACM International Conference on the Foundations of Software Engineering (FSE)*. ACM, 2026, pp. 1272–1276.
- [23] S. Kapoor, B. Stroebel, Z. S. Siegel, N. Nadgir, and A. Narayanan, “AI agents that matter,” *Transactions on Machine Learning Research*, 2025.
- [24] C. Qian, E. C. Acikgoz, H. Wang, X. Chen, A. Sil, D. Hakkani-Tür, G. Tur, and H. Ji, “SMART: Self-aware agent for tool overuse mitigation,” in *Findings of the Association for Computational Linguistics: ACL 2025*. Association for Computational Linguistics, 2025, pp. 4604–4621.
- [25] Y. Xia, W. Shen, Y. Wang, J. K. Liu, H. Sun, S. Wu, J. Hu, and X. Xu, “LeetCodeDataset: A temporal dataset for robust evaluation and efficient training of code LLMs,” arXiv preprint arXiv:2504.14655, 2025.

- [26] C. Daxerer, J. Glock, C. Macho, R. Ramler, M. Pinzger, and J. Pichler, "Replication package for: 'Do coding agents still need a debugger? Frontier-model-based fault localization at the accuracy ceiling,'" Zenodo, 2026, to be published after peer review.
- [27] S. Kang, G. An, and S. Yoo, "A quantitative and qualitative evaluation of LLM-based explainable fault localization," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 1424–1446, 2024.
- [28] C. Xu, Z. Liu, X. Ren, G. Zhang, M. Liang, and D. Lo, "FlexFL: Flexible and effective fault localization with open-source large language models," *IEEE Transactions on Software Engineering*, vol. 51, no. 5, pp. 1455–1471, 2025.
- [29] M. Farzandway and F. Ghassemi, "SieveFL: Hierarchical runtime-aware pruning for scalable LLM-based fault localization," arXiv preprint arXiv:2605.13491, 2026.
- [30] Y. Qin, S. Wang, B. Lin, X. Peng, S. Ouyang, L. Chen, and X. Mao, "Let the code speak: Incorporating program dynamic state for better method-level fault localization," in *Proceedings of the 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2025, pp. 804–815.
- [31] R. Tian, Y. Ye, Y. Qin, X. Cong, Y. Lin, Y. Pan, Y. Wu, H. Hui, W. Liu, Z. Liu, and M. Sun, "DebugBench: Evaluating debugging capability of large language models," in *Findings of the Association for Computational Linguistics: ACL 2024*. Association for Computational Linguistics, 2024, pp. 4173–4198.
- [32] N. Jain, K. Han, A. Gu, W.-D. Li, F. Yan, T. Zhang, S. I. Wang, A. Solar-Lezama, K. Sen, and I. Stoica, "LiveCodeBench: Holistic and contamination free evaluation of large language models for code," in *Proceedings of the 13th International Conference on Learning Representations (ICLR)*, 2025, pp. 58 791–58 831.
- [33] S. Chen, P. Pussarla, and B. Ray, "DyCodeEval: Dynamic benchmarking of reasoning capabilities in code large language models under data contamination," in *Proceedings of the 42nd International Conference on Machine Learning (ICML)*, ser. Proceedings of Machine Learning Research, vol. 267. PMLR, 2025, pp. 8890–8909.
- [34] A. Spiesberger, J. J. Vazquez, N. Pochinkov, T. Gavenčiak, P. Grietzer, G. Leech, and N. Schoots, "Soft contamination means benchmarks test shallow generalization," arXiv preprint arXiv:2602.12413, 2026.
- [35] F. Li, J. Jiang, J. Sun, and H. Zhang, "Evaluating the generalizability of LLMs in automated program repair," in *Proceedings of the 47th IEEE/ACM International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. IEEE, 2025, pp. 91–95.
- [36] F. Rabbi and J. Yang, "HEJ-Robust: A robustness benchmark for LLM-based automated program repair," in *Proceedings of the 3rd ACM International Conference on AI-Powered Software (AIware)*. ACM, 2026, pp. 371–375.
- [37] P. Orvalho and M. Kwiatkowska, "Are large language models robust in understanding code against semantics-preserving mutations?" arXiv preprint arXiv:2505.10443, 2025.
- [38] J. Cao, W. Zhang, and S.-C. Cheung, "Concerned with data contamination? Assessing countermeasures in code language model," arXiv preprint arXiv:2403.16898, 2024.
- [39] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque, "PIT: A practical mutation testing tool for Java (demo)," in *Proceedings of the 25th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2016, pp. 449–452.
- [40] PIT, "Mutation operators," Technical documentation, accessed: 2026-09-08. [Online]. Available: <https://pitest.org/quickstart/mutators/>
- [41] A. V. Kamiński, L. Palechor, C.-P. Bezemer, and A. Hindle, "PySStuBs: Characterizing single-statement bugs in popular open-source Python projects," in *Proceedings of the 18th IEEE/ACM International Conference on Mining Software Repositories (MSR)*. IEEE, 2021, pp. 520–524.
- [42] N. Maveli, A. Vergari, and S. B. Cohen, "What can large language models capture about code functional equivalence?" in *Findings of the Association for Computational Linguistics: NAACL 2025*. Association for Computational Linguistics, 2025, pp. 6880–6918.
- [43] Y. Zhang, Y. Xie, S. Li, K. Liu, C. Wang, Z. Jia, X. Huang, J. Song, C. Luo, Z. Zheng, R. Xu, Y. Liu, S. Zheng, and X. Liao, "Unseen horizons: Unveiling the real capability of LLM code generation beyond the familiar," in *Proceedings of the 47th IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE, 2025, pp. 604–615.
- [44] T. J. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, 1976.
- [45] M. Mizrahi, G. Kaplan, D. Malkin, R. Dror, D. Shahaf, and G. Stanovsky, "State of what art? A call for multi-prompt LLM evaluation," *Transactions of the Association for Computational Linguistics*, vol. 12, pp. 933–949, 2024.
- [46] M. Sclar, Y. Choi, Y. Tsvetkov, and A. Suhr, "Quantifying language models' sensitivity to spurious features in prompt design or: How I learned to start worrying about prompt formatting," in *Proceedings of the 12th International Conference on Learning Representations (ICLR)*, 2024, pp. 25 055–25 083.
- [47] A. Korn, L. Zaruchas, C. Arora, A. Metzger, S. Smolka, F. Wang, and A. Vogelsang, "Reporting LLM prompting in automated software engineering: A guideline based on current practices and expectations," in *Proceedings of the 3rd IEEE/ACM International Conference on AI Foundation Models and Software Engineering (FORGE)*. ACM, 2026, pp. 184–194.
- [48] S. Baites, F. Angermeier, C. Arora, M. Muñoz Barón, C. Chen, L. Böhme, F. Calefato, N. Ernst, D. Falesi, B. Fitzgerald, D. Fucci, J. He, C. Treude, M. Kalinowski, S. Lambiase, D. Russo, M. Lungu, C. Martinez Montes, L. Prechelt, P. Ralph, R. van Tonder, and S. Wagner, "Guidelines for empirical studies in software engineering involving large language models," arXiv preprint arXiv:2508.15503, 2025.
- [49] DeepInfra, "Qwen3.8-27B pricing," Model page, accessed: 2026-10-07. [Online]. Available: <https://deepinfra.com/Qwen/Qwen3.8-27B>
- [50] A. C. Davison and D. V. Hinkley, *Bootstrap Methods and their Application*. Cambridge University Press, 1997.
- [51] E. Miller, "Adding error bars to evals: A statistical approach to language model evaluations," arXiv preprint arXiv:2411.00640, 2024.
- [52] A. C. Cameron and D. L. Miller, "A practitioner's guide to cluster-robust inference," *Journal of Human Resources*, vol. 50, no. 2, pp. 317–372, 2015.
- [53] J. G. MacKinnon and M. D. Webb, "Wild bootstrap inference for wildly different cluster sizes," *Journal of Applied Econometrics*, vol. 32, no. 2, pp. 233–254, 2017.
- [54] J. G. MacKinnon, M. Ø. Nielsen, and M. D. Webb, "Cluster-robust inference: A guide to empirical practice," *Journal of Econometrics*, vol. 232, no. 2, pp. 272–299, 2023.
- [55] M. Bertrand, E. Duflo, and S. Mullainathan, "How much should we trust differences-in-differences estimates?" *The Quarterly Journal of Economics*, vol. 119, no. 1, pp. 249–275, 2004.
- [56] A. Wei, J. Cao, R. Li, H. Chen, Y. Zhang, Z. Wang, Y. Liu, T. S. F. X. Teixeira, D. Yang, K. Wang, and A. Aiken, "EquiBench: Benchmarking large language models' reasoning about program semantics via equivalence checking," in *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, 2025, pp. 33 868–33 881.
- [57] N. Oza, I. Govil, P. Gupta, D. Khandelwal, D. Garg, and P. Singla, "LLMs are brittle to simple code transformations: Introducing CETBench – a benchmark for code-equivalence checking," in *Findings of the Association for Computational Linguistics: ACL 2026*. Association for Computational Linguistics, 2026, pp. 41 653–41 685.
- [58] Z. Wu and C. Nita-Rotaru, "ALIBI: Adaptive agentic attacks on LLM-based vulnerability detectors via adversarial code comments," *Computing Research Repository*, vol. abs/2607.24964, 2026.
- [59] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing?" in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. ACM, 2014, pp. 654–665.