VeriBRT: Plan-Guided, Evidence-Based Automated Bug Reproduction

Anonymous Author

***Abstract*—Bug-reproducing tests (BRTs) provide executable evidence for localizing, understanding, and validating software defects. Recent large language model (LLM) agents have made substantial progress in generating BRTs directly from natural-language issue reports and source repositories. However, existing approaches largely treat bug reproduction as a sequential generation problem, repeatedly rediscovering behavioral information while uniformly using heterogeneous evidence despite its varying reliability. Consequently, behavioral information extracted early in the pipeline cannot consistently guide subsequent reasoning, and potentially valid BRTs may be discarded due to uncertain execution or LLM-based evidence. We present VERIBRT, an LLM-based framework that rethinks both behavioral representation and evidence utilization. VERIBRT represents issue-report behavior as a persistent *Two-Axis Plan*, with a *Checklist* specifying where the generated test should reach and an *Intentlist* specifying what behavior it should verify. It further introduces a reliability-aware evidence strategy and is the first approach to incorporate localization coverage into the BRT generation loop as a deterministic validity criterion, using deterministic evidence for diagnostic regeneration and uncertain evidence only for confidence-based ranking. We evaluate VERIBRT on SWT-Bench Verified. Under the same DeepSeek-V4-Flash model and evaluation protocol, VERIBRT reproduces 341 of 433 issues, outperforming the strongest baseline by 9.7 absolute percentage points. Ablation, evidence-quality, and human-agreement analyses further support the effectiveness of the proposed approach.**

***Index Terms*—LLM agents, bug-reproducing tests, SWT-Bench**

I. INTRODUCTION

Bug reproduction is a critical step in software debugging because an executable failure enables developers to localize, understand, and validate fixes for reported defects [1]–[3]. In practice, however, software failures are often reported through natural-language (NL) issue descriptions, logs, or partial execution context, which frequently lack sufficient information for reliable bug reproduction. A bug-reproducing test (BRT) bridges this gap by transforming the behavioral information contained in an issue report into executable evidence.

Automating bug reproduction has been studied for decades. Early approaches relied primarily on runtime execution artifacts, ranging from record-and-replay techniques that capture complete executions [4], [5] to symbolic and search-based methods that synthesize failures from crash stacks, field data, or search objectives [1], [2], [6]. Another line of work exploited structured bug reports and information retrieval to facilitate bug reproduction, organizing reports into expected behavior (EB), observed behavior (OB), and steps to reproduce (S2R) [7], while identifying suspicious files from report text [8], [9]. Together, these techniques improved different stages of the bug reproduction process, but they primarily focused on recovering or localizing failures rather than automatically synthesizing executable BRTs directly from issue reports.

The emergence of LLMs has shifted the focus toward repository-level BRT generation. This shift is motivated by the scarcity of developer-written BRTs: in SWE-bench repository repair tasks, *i.e.,* real GitHub issue-fixing tasks [10], a corresponding BRT almost never exists before the code is fixed [10], [11], and in Defects4J only about 4% of reports include a failing test [12]. Benchmarks such as SWT-Bench [11] have thus accelerated research on repository-level BRT generation, with approaches evolving from prompt-based generation [11] to autonomous LLM agents capable of repository exploration, planning, execution, and iterative refinement [13]–[18]. Despite their different implementations, these systems share a common objective: generating a test that satisfies the fail-to-pass ($F \rightarrow P$) criterion [11], [14], namely, failing on the buggy version and passing after the correct patch is applied.

Despite this rapid progress, reliably generating BRTs remains fundamentally challenging because existing LLM-based approaches largely treat bug reproduction as a sequential generation problem. Repository exploration, planning, test generation, and refinement are performed stage by stage, with behavioral information and execution evidence produced and consumed within individual stages. As a result, information that could guide the entire generation process is often used only transiently. Issue reports already contain much of the behavioral information required to construct a valid BRT, including where the reported fault should be exercised (S2R) and what behavior should be verified (EB/OB). Modern LLMs are capable of extracting the information from free-form issue descriptions, yet existing agents primarily consume it as *intermediate* reasoning during repository exploration and test generation rather than preserving it as reusable knowledge. Thus, every generation attempt has to repeatedly infer essentially the same behavioral intent, while later stages can no longer benefit from the reasoning performed earlier in the pipeline.

Additionally, modern LLM-based systems increasingly rely on heterogeneous evidence to evaluate generated tests, combining execution feedback with semantic reasoning. These evidence sources, however, differ fundamentally in reliability. Observable failures, such as syntax errors, compilation failures, or failing to expose the reported bug on the buggy version, provide deterministic evidence that directly indicates whether regeneration is necessary and how it should be guided. By contrast, surrogate-fix feedback and LLM-based semantic

judgments are inherently probabilistic because they depend on imperfect proxies for the unavailable gold patch and testing oracle. Existing approaches nevertheless use both types of evidence as generation-time decision signals, allowing uncertain evidence to reject, regenerate, or select candidates [12], [14], [16]. Consequently, potentially valid bug-reproducing tests may be discarded simply because the surrogate fix or semantic judgment is incorrect.

VERIBRT addresses these limitations by rethinking both how behavioral information is represented and how generation-time evidence is utilized. Rather than repeatedly rediscovering behavioral intent, VERIBRT explicitly represents the behavioral information contained in issue reports as a persistent **Two-Axis Plan**, consisting of a *Checklist* that specifies where the generated test should reach and an *Intentlist* that specifies what behavior it should verify. Instead of serving as one-off planning results, these two artifacts provide a unified semantic reference that consistently guides repository exploration, test generation, and downstream validation throughout the entire generation process. Building upon this persistent representation, VERIBRT further distinguishes generation-time evidence according to its role and reliability. In particular, it is the first approach to explicitly incorporate localization coverage into the BRT generation loop as a deterministic validity criterion, allowing coverage information to directly guide diagnostic regeneration rather than merely serving as post-hoc evaluation. Observable deterministic evidence, including localization coverage and failing to expose the reported bug on the buggy version, is used exclusively to trigger regeneration through a **Validity Filter**, whereas uncertain execution- and LLM-based evidence is reserved exclusively for confidence assessment through **Evidence-based Ranking**. By decoupling regeneration from confidence assessment, VERIBRT progressively increases confidence while preserving potentially valid bug-reproducing tests instead of prematurely discarding them.

We evaluate VERIBRT on SWT-Bench Verified [11], the standard benchmark for repository-level bug-reproducing test generation. Derived from SWE-bench Verified [10], it contains 433 real-world Python repository issues together with their corresponding buggy repositories and gold patches, enabling direct evaluation under the widely adopted $F \rightarrow P$ criterion and fair comparison with existing LLM-based BRT generation approaches. Beyond the standard $F \rightarrow P$ evaluation, we further analyze localization coverage, evidence quality, and human agreement to better understand the effectiveness of the proposed behavioral representation and evidence utilization strategies. Under the same DeepSeek-V4-Flash [19] model and evaluation protocol, VERIBRT successfully reproduces 341 issues, improving over its OpenHands backbone by 42 instances (+9.7 absolute percentage points) and consistently outperforming all same-model baselines. Ablation studies further show that both the persistent Two-Axis Plan and the proposed evidence utilization strategy contribute substantially to these gains. Finally, automatic evidence analyses together with an independent human audit provide consistent empirical support for the proposed confidence ranking, indicating that higher-ranked candidates are more likely to both reproduce the reported bug and align with the intended behavior described in the issue report.

This paper makes the following contributions:

- We propose a persistent *Two-Axis Plan* that transforms the behavioral information contained in issue reports into a reusable semantic representation, enabling the same representation to consistently guide repository exploration, test generation, and downstream validation.
- We introduce a reliability-aware evidence utilization strategy that explicitly incorporates localization coverage into the BRT generation loop as a deterministic validity criterion, while decoupling diagnostic regeneration from confidence assessment by reserving uncertain execution- and LLM-based evidence exclusively for candidate ranking.
- We conduct a comprehensive evaluation on SWT-Bench Verified, showing that VERIBRT reproduces 42 more issues (+9.7 percentage points) than the strongest baseline, and that its confidence ranking is supported by both benchmark outcomes and an independent human audit.
- We release our implementation, prompts, and full per-instance experimental results to facilitate future research on repository-level bug reproduction [1].

II. RELATED WORK

Traditional bug reproduction techniques can be broadly divided into two categories: dynamic approaches, which rely on runtime execution artifacts such as execution traces and crash stacks [1], [2], [4]–[6], [20], [21], and static approaches, which exploit information retrieval and textual analysis over project artifacts [8], [22]–[24]. To further improve bug reproduction accuracy, recent approaches have begun leveraging LLMs for repository-level bug reproduction.

A. Bug-Reproducing Test Generation

Recent approaches leverage LLMs to generate bug-reproducing tests using an NL issue report and a repository snapshot as input, with SWT-Bench serving as the standard benchmark for this task [11]. They explore different aspects of the generation process. Specifically, ZeroShot and ZeroShot-Plus [11] directly prompt an LLM with the issue report and repository context to generate a test in a single pass. LIBRO [25] improves generation through few-shot prompting, test post-processing, and heuristic ranking. Echo [16] and iCore [18] enhance repository exploration through graph-based and iterative retrieval, respectively. Otter [13] introduces an explicit planning stage before generation, while e-Otter++ [14] further extends this workflow with iterative refinement. Together, these approaches demonstrate the effectiveness of LLMs for repository-level bug reproduction.

Different from prior work that primarily improves retrieval, planning, or generation, VERIBRT explicitly represents the behavioral information contained in issue reports and reuses this representation throughout the generation process. By

[1] https://anonymous.4open.science/r/VeriBRT-D7E9

maintaining a persistent behavioral representation, VERIBRT provides a unified semantic reference that consistently guides the generation of bug-reproducing tests.

B. Bug-Reproducing Test Selection

The accepted criterion for a bug-reproducing test is the $F \rightarrow P$ criterion: a valid BRT should fail on the buggy version and pass after applying the corresponding gold patch [11]. Since the gold patch is unavailable during test generation, existing approaches rely on auxiliary evidence to estimate whether a generated test is likely to satisfy this criterion.

Existing work has explored two main categories of evidence. Dynamic evidence is obtained through code execution, such as whether a generated test changes from failing to passing under a surrogate fix [14], [16]. Semantic evidence is derived from LLM-based assessment of whether the generated assertions are consistent with the issue description or intended behavior [15]–[17]. These signals are incorporated into different validation and selection strategies. For example, e-Otter++ uses surrogate-fix feedback to select the final candidate [14]; AssertFlip employs an LLM-based validator to filter generated tests and trigger regeneration when necessary [12]; and Echo combines execution-based and semantic evidence during iterative refinement [16]. Together, these studies show that execution and semantic evidence are complementary for improving the reliability of generated bug-reproducing tests.

VERIBRT complements these efforts by introducing a different evidence utilization paradigm. It separates evidence according to its role: deterministic failures trigger regeneration, whereas uncertain execution- and LLM-based signals are reserved for confidence assessment and candidate ranking. This decoupling enables validation and selection to operate under different reliability requirements while fully exploiting complementary evidence sources. We provide a solid empirical motivation for this design choice in Section III and quantify its trade-off in RQ5.

III. MOTIVATION

Building on the preceding observations of existing bug-reproduction techniques, the design of VERIBRT focuses on two core pain points: how to efficiently recover structured behavioral information from free-form issues, and how to robustly use non-deterministic feedback. To this end, VERIBRT makes two key design choices: first, extracting an explicit and reusable execution plan that specifies where the bug manifests and what to assert; second, strictly separating strong evidence that can directly reject a candidate from weak evidence that only ranks candidates. These two design choices rest on two empirical premises, which this section substantiates with quantitative evidence.

The first question is whether repository-level, free-form issue descriptions still expose enough behavioral information to accurately guide the construction of the test assertion. To answer it, we ran BEE, an SVM tagger over CoreNLP features [7], on the 433 SWT-Bench Verified issues [11], labeling each sentence by the behavioral element it contains

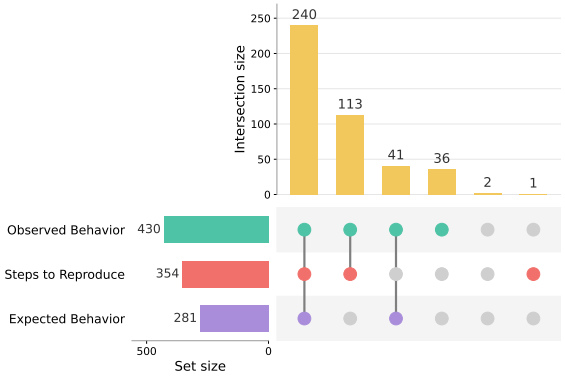

Fig. 1. Behavioral elements detected by BEE [7] across the 433 SWT-Bench Verified issues [11]

GitHub Issue: `pallets/flask-5014`

Things do not work correctly if a Blueprint is given an empty name. *It would be helpful if a `ValueError` was raised when trying to do that.*

GitHub Issue: `django/django-16642`

`FileResponse` will set the content type as `text/html`, even if the last file extension is `.Z` or `.br`.

GitHub Issue: `sympy/sympy-24066`

`SI._collect_factor_and_dimension()` cannot properly detect that the exponent is dimensionless.

Fig. 2. Examples whose expected behavior is implicit or phrased indirectly. BEE misses EB in these cases, although the intended oracle is recoverable from context.

(OB, EB, S2R). As shown in Figure 1, these free-form issues are in fact richly structured: a majority (55.4%) carry all three behavioral elements, with observed behavior (OB) present in 99.3% and steps to reproduce (S2R) in 81.8%. Most relevant here, the expected behavior (EB) that serves as the assertion target is explicitly stated in 64.9% of issues. These results support the first premise: for the vast majority of free-form issues, the behavioral intent needed to pin down the test assertion is available in the report text.

Note that the 64.9% reported by BEE represents a lower bound of EB availability, as BEE is keyed strictly to explicit phrasing and undercounts implicit context. For instance, in `flask-5014` the report states "It would be helpful if a `ValueError` was raised" (Figure 2): BEE tags this as OB and S2R but misses the EB, even though the intended post-fix contract is evident from the report. More generally, such implicit EB is usually recoverable from surrounding context, domain conventions, or common API semantics. VERIBRT therefore extracts this assertion target with an LLM rather than a feature-based tagger, on the design assumption that an LLM can perform this recovery. We examine this assumption indirectly through the human intent-alignment audit in Section VI-E.

Even with this plan of where the fault manifests and what to assert, a candidate test is only a proxy for a correct BRT [26],

which raises the second question: how far can the evidence about it be trusted? Some evidence is observable and reliable: whether the candidate is a valid, runnable test, and whether it fails on the buggy code. These are definite outcomes that can directly reject the test and trigger a diagnostic retry.

Other evidence is uncertain: whether a surrogate fix, standing in for the unavailable correct patch [14], makes the candidate pass, and whether an LLM-based intent check finds its assertion aligned with the issue. Either signal can be absent even for a test that truly reproduces the bug, since the surrogate fix may not be produced and the intent check may lack textual evidence; their absence is therefore not reliable negative evidence. Had we rejected every candidate whose surrogate fix fails to make it pass, we would have discarded 59.8% of tests that in fact reproduce the bug (Section VI-E). Because their absence is not reliable negative evidence, VERIBRT keeps these uncertain signals for ranking and never lets them reject a candidate.

IV. VERIBRT

Figure 3 presents the overall architecture of VERIBRT, which is built upon the OpenHands agentic framework [27]. Given a NL issue report and the corresponding buggy repository, VERIBRT generates a bug-reproducing test through three consecutive stages: **Two-Axis Planning**, **Validity Filtering**, and **Evidence-based Ranking**. The overall design follows a simple principle: progressively increase confidence while preserving potentially valid candidates. Specifically, **Two-Axis Planning** explicitly represents the behavioral information contained in the issue report as a reusable Checklist and Intentlist, providing a persistent semantic reference that guides repository exploration, test generation, and subsequent reasoning. Guided by this plan, the agent generates a candidate BRT. Additionally, **Validity Filtering** determines whether regeneration is necessary using only observable, deterministic failures, ensuring that every retry is supported by an explicit diagnosis rather than uncertain evidence. Finally, **Evidence-based Ranking** evaluates admitted candidates using uncertain execution- and LLM-based signals solely for prioritization, never for rejection, thereby preserving potentially valid candidates while still exploiting complementary evidence. The generation process terminates once a candidate reaches the highest confidence level or the attempt budget is exhausted. The three stages are described in detail below.

A. Two-Axis Planning

A valid bug-reproducing test must satisfy two complementary requirements: it must execute the code region where the reported fault manifests and assert the behavior expected by the issue report. Existing LLM-based approaches infer these two forms of behavioral information during repository exploration and test generation, but they are primarily consumed as intermediate reasoning rather than maintained as reusable semantic artifacts. VERIBRT instead explicitly represents them before generation as a reusable two-axis behavioral plan that consistently guides the entire generation process.

Given the issue description and a repository snapshot, VERIBRT extracts two complementary artifacts only once before the first generation attempt: a **Checklist**, which specifies *where* the generated test should reach, and an **Intentlist**, which specifies *what* behavior it should verify. Together, the Checklist and Intentlist form a unified behavioral plan that serves as a *persistent* semantic reference throughout the generation process. In each generation attempt, the agent reuses this plan together with the issue description and repository context to guide repository exploration and generate a candidate bug-reproducing test. Each generated candidate is subsequently passed to the Validity Filter for deterministic verification before entering the downstream Evidence-based Ranking stage.

Checklist (Where to test). The Checklist identifies where the reported fault is most likely to manifest without suggesting implementation fixes or expected outputs. To efficiently narrow the search space over repository-level code, VERIBRT adopts the robust BM25 lexical retriever [28], following the retrieval setting of SWE-bench and SWT-Bench [10], [11]. Prior work has shown that lexical retrieval is both effective and efficient for matching long NL issue descriptions against large software repositories. Specifically, BM25 first retrieves the top-k most relevant source files, which are subsequently summarized by an LLM to produce a fault hypothesis together with a priority-ordered list of suspicious files and the target functions, methods, or classes that the generated test should reach. This hierarchical retrieval strategy substantially reduces the repository search space while preserving semantic reasoning for bug localization.

Intentlist (What to assert). Complementary to the Checklist, the Intentlist specifies the behavioral contract that distinguishes the intended behavior from the buggy implementation. Inspired by the BEE framework [7], a structured representation of behavioral information in bug reports, the Intentlist records four complementary elements: Expected Behavior (EB), Observed Behavior (OB), Steps to Reproduce (S2R), and the observation point at which the assertion should be evaluated. Importantly, the Intentlist is derived solely from the issue report rather than the current buggy implementation, since a bug-reproducing test should validate the developer's intended behavior instead of the observed faulty behavior [29], [30]. During generation, the Intentlist guides assertion construction; in later stages, it serves as the semantic reference for evaluating whether the generated assertions faithfully capture the behavior described in the issue report.

B. Validity Filtering

Not every unsuccessful generation attempt should trigger the same response. Some failures are observable and deterministic, making their causes directly diagnosable, whereas others arise from uncertain execution- or LLM-based evidence that cannot reliably distinguish valid from invalid bug-reproducing tests during generation (Section III). VERIBRT therefore adopts a *Validity Filter* that triggers regeneration only on deterministic failures while reserving uncertain evidence for the downstream ranking stage (Section IV-C). Instead of simply discarding

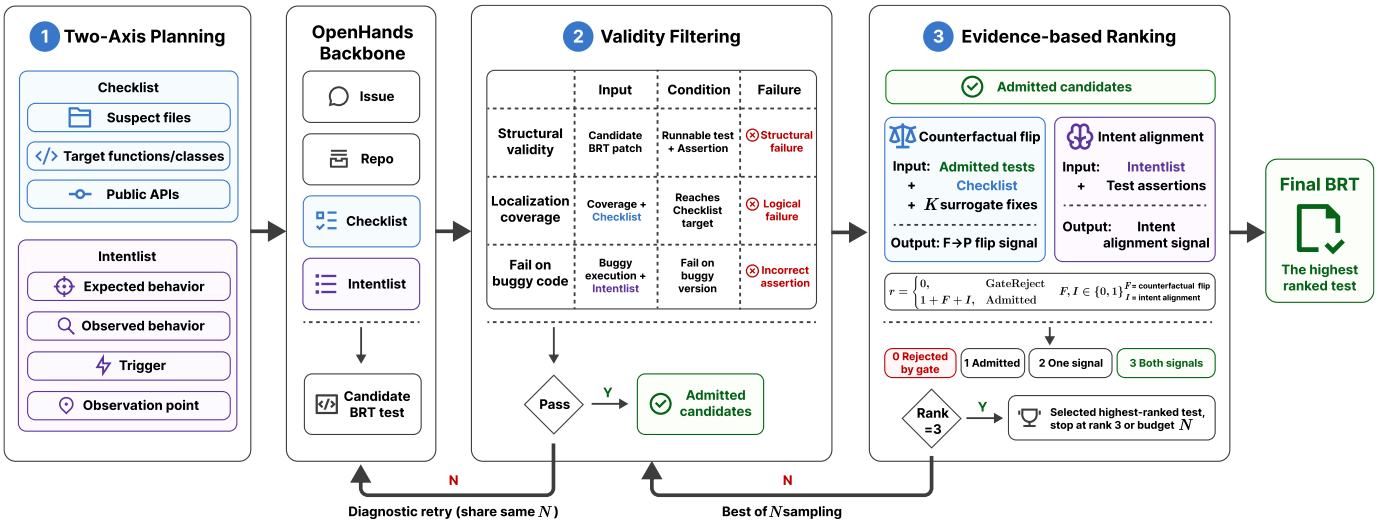

Fig. 3. Overview of the VERIBRT pipeline.

invalid candidates, the filter identifies the underlying failure cause and supplies a targeted retry prompt. It consists of an ordered cascade of three checks, where failure at any stage immediately terminates the cascade and initiates diagnostic regeneration. The failure categories follow the SWT-Bench taxonomy [31].

Structural validity. The first check ensures that the generated test is executable. Before execution, a static preflight analysis verifies that the generated test is syntactically correct, contains an explicit executable assertion, and introduces no obvious dependency or safety violations. Candidates that fail this check are rejected before execution, and the retry prompt instructs the agent to regenerate a structurally valid test.

Localization coverage. A syntactically valid test may still fail to reach the reported fault. VERIBRT therefore executes the candidate on the buggy repository, collects dynamic coverage, and intersects the executed functions with the target functions, methods, or classes specified by the Checklist. To the best of our knowledge, VERIBRT is the first approach to explicitly incorporate localization coverage into the BRT generation loop as a deterministic validity criterion. When coverage information is available but no target location is reached, the candidate is classified as a Logical Failure [31]. The corresponding retry prompt instructs the agent to reach the target functionality through an appropriate project-level entry point.

Fail on buggy code. Finally, VERIBRT verifies whether the candidate actually exposes the reported bug. A failing execution establishes the correct direction for a valid BRT. In contrast, a passing execution indicates that the generated test has not exposed the reported fault. This can occur for two reasons: (i) the test incorrectly asserts the current buggy behavior instead of the expected behavior [29]; or (ii) although the faulty code is executed, the resulting incorrect state fails to propagate to the assertion point, leaving the assertion unaffected [32]. Such candidates are classified as Incorrect Assertions [31], and the retry prompt instructs the agent to revise the assertion so that it captures the expected post-fix behavior at an appropriate observation point.

The ordering of the three checks follows the causal progression of the PIE model [32]–[34], which states that a fault must first be executed (execution), then infect the program state (infection), and finally propagate to an observable failure (propagation). Structural validity ensures that the generated test is executable, localization coverage verifies that the fault can be reached, and failing on the buggy code confirms that the fault propagates to the observable assertion. This staged design allows each check to eliminate a distinct class of deterministic failures before proceeding to the next stage.

C. Evidence-based Ranking

Candidates admitted by the Validity Filter may still differ substantially in reproduction quality. Some expose the reported fault only incidentally, whereas others assert behavior that is correlated with the bug rather than the behavior expected by the issue report. Distinguishing between these candidates necessarily relies on uncertain evidence, which is informative but insufficiently reliable to justify regeneration (Section III). VERIBRT therefore uses uncertain evidence exclusively for candidate prioritization rather than rejection. Specifically, it combines two different signals: a dynamic signal indicating whether an independently generated surrogate fix causes a fail-to-pass flip, and a semantic signal indicating whether the generated assertions faithfully capture the expected behavior described in the issue report.

Counterfactual flip. The gold patch is unavailable during test generation, making it impossible to directly verify whether a candidate would satisfy the desired fail-to-pass behavior. VERIBRT therefore approximates this counterfactual using surrogate fixes [14]. Specifically, the agent independently generates up to K source-only fixes from the issue description, repository context, and the optional Checklist localization. To avoid information leakage [35], the fixing agent never receives the generated test or its execution results, ensuring that the synthesized fixes are independent of any particular candidate test. Each surrogate fix is then applied independently, and the

candidate test is re-executed. The counterfactual-flip signal is recorded once any surrogate fix changes the test outcome from fail to pass, at which point VERIBRT immediately terminates the surrogate-fix search. The presence of a counterfactual flip increases confidence that the generated test captures the buggy behavior, but its absence does not invalidate the candidate.

Intent alignment. Execution alone cannot determine whether a generated test verifies the behavior actually requested by the issue report. A candidate may expose the reported fault while asserting an incidental outcome, checking a secondary effect, or over-specifying behavior beyond the reported issue. To evaluate semantic correctness, VERIBRT employs an LLM judge only at this stage and confines it to a closed, quote-grounded rubric [36], [37]. The judge receives the EB and observation point extracted in the Intentlist together with the generated test code, and verifies whether (i) the assertion targets the specified observation point, (ii) the EB is traceable to the original issue report, and (iii) the generated test avoids irrelevant over-specification. The intent-alignment signal is recorded only when all three conditions are satisfied.

Unlike prior work that directly acts on uncertain execution or LLM evidence, VERIBRT converts both signals into ranking evidence, ensuring that no candidate admitted by the Validity Filter is discarded solely because uncertain evidence is absent. Formally, each admitted candidate is assigned the following rank:

$$\mathrm{rank}(V) = \begin{cases} 0, & \text{if } V \text{ is rejected by the Validity Filter;} \\ 1 + \mathbb{1}[\text{counterfactual flip}] & \\ \quad + \mathbb{1}[\text{intent alignment}], & \text{otherwise,} \end{cases}$$

where $\mathbb{1}[\cdot]$ is the indicator function. Rank 0 is reserved for rejected candidates. Admitted candidates receive ranks from 1 to 3, where rank 1 carries neither positive signal, rank 2 carries exactly one positive signal, and the maximum rank $r_{\max} = 3$ carries both.

The assigned rank drives the entire generation loop under a budget of N attempts. After each generation attempt, the candidate is first processed by the Validity Filter. Candidates rejected by the filter (rank 0) trigger a diagnostic retry using a failure-specific prompt that explicitly targets the identified failure mode. In contrast, candidates admitted by the filter (ranks 1–$r_{\max}$) enter the Evidence-based Ranking stage, where the two uncertain signals are evaluated and converted into a rank. Rather than triggering another diagnostic retry, admitted candidates follow a best-of-N resampling strategy: the prompt is reset to the original two-axis behavioral plan and task specification, and the next candidate differs only through stochastic sampling. Throughout the search, VERIBRT retains the highest-ranked candidate, replacing it only when a strictly higher rank is observed and keeping the earliest candidate in the case of ties. The generation process terminates once a candidate reaches $r_{\max}$ or the attempt budget is exhausted, at which point the retained candidate is returned as the final BRT.

V. EXPERIMENTAL DESIGN

To evaluate the effectiveness of VERIBRT for repository-level BRT generation, we investigate five research questions:

- **RQ1:** How effective is VERIBRT compared with existing prompt-based and LLM-agent approaches for BRT generation?
- **RQ2:** How do persistent behavioral representation and reliability-aware evidence utilization individually contribute to the effectiveness of VERIBRT?
- **RQ3:** How robust is VERIBRT under different generation and surrogate-fix budgets?
- **RQ4:** How does VERIBRT's performance vary across repositories, and what failure modes remain?
- **RQ5: Evidence quality.** How well do VERIBRT's evidence signals and confidence ranking reflect true bug reproduction and human judgments?

A. Benchmark

We evaluate VERIBRT and all selected baselines on the SWT-Bench Verified benchmark. SWT-Bench Verified [11] is derived from SWE-bench Verified [10], which was originally proposed for evaluating repository-level code repair. Each instance in SWE-bench Verified is constructed from a real-world NL issue report collected from one of twelve popular open-source Python repositories on GitHub and has been manually verified to be solvable. SWT-Bench Verified [11] reformulates this benchmark for bug reproduction by retaining 433 of the 500 verified instances and replacing the code repair objective with bug reproduction.

Because varying the best-of-N sampling budget and surrogate fix budget K requires repeated runs, RQ3 uses a fixed 100-instance subset uniformly sampled from SWT-Bench Verified. RQ1, RQ2, and RQ4 use the full 433-instance benchmark; in RQ5, automatic evidence analyses use all 433 instances, while the human audit and selection comparison use the same subset.

B. Models

To evaluate the generality of VERIBRT across modern LLMs, we select three recently released state-of-the-art models spanning different deployment settings and capability levels. Specifically, we include DeepSeek-V4-Flash [19] as a representative open-weight model, OpenAI's GPT-5.4-mini [38] as a cost-efficient proprietary model, and Google's Gemini-3.1-Pro [39] as a flagship proprietary model. Together, these models cover both open-weight and proprietary ecosystems while representing different capability–cost trade-offs.

C. Evaluation Metrics

We evaluate the effectiveness of BRT generation using two categories of metrics. The first category consists of the two evaluation metrics introduced by SWT-Bench [11].

***F → P* Rate.** This metric measures the fraction of SWT-Bench Verified instances for which at least one generated test fails on the buggy version and passes after applying the corresponding gold patch. It has been widely adopted in prior work [11], [13]–[18] as the primary indicator of successful bug reproduction and is also important for validating patch correctness. Since multiple tests may be generated for each

TABLE I
COMPARISON WITH BASELINES ON SWT-BENCH VERIFIED.

	DeepSeek-V4-Flash			GPT-5.4-mini			Gemini-3.1-Pro		
Approach	Instances	$F \to P$	ΔCov	Instances	$F \to P$	ΔCov	Instances	$F \to P$	ΔCov
ZeroShot [11]	30	6.9	15.8	43	9.9	13.3	55	12.7	13.2
ZeroShotPlus [11]	46	10.6	24.7	68	15.7	33.3	219	50.6	57.3
OpenHands [27]	299	69.1	66.9	148	34.2	54.9	303	70.0	66.5
AssertFlip [12]	282	65.1	69.4	189	43.6	64.9	324	74.8	**74.8**
Echo [16]	269	62.1	**71.3**	129	29.8	**67.6**	211	48.7	56.4
VERIBRT	**341**	**78.8**	65.0	**203**	**46.9**	52.4	**351**	**81.1**	67.2
	Claude 3.7 Sonnet†								
Approach	instances	$F \to P$	ΔCov	instances	$F \to P$	ΔCov	instances	$F \to P$	ΔCov
e-Otter++ [14]	263	60.7	62.5	263	60.7	62.5	263	60.7	62.5

† e-Otter++ results are leaderboard-only under Claude 3.7 Sonnet and are repeated only as a cross-model reference.

benchmark instance, an instance is counted as successful if any generated test satisfies the $F \to P$ criterion.

ΔCov. This metric measures the extent to which the generated tests cover the code lines modified by the gold patch. Specifically, it is computed as the fraction of changed lines (added, deleted, or modified) that are covered by the generated tests. However, covering the changed lines alone does not guarantee successful bug reproduction: a test may execute the modified code without asserting the behavior that distinguishes the buggy version from the fixed one [11]. Consequently, a higher ΔCov does not necessarily imply a higher $F \to P$ rate. Together, ΔCov and the $F \to P$ rate provide complementary perspectives on bug reproduction performance, enabling a more fine-grained evaluation [11].

The second category explains why the Evidence-based Ranking module uses positive evidence to rank candidate tests rather than to reject them. To this end, we evaluate how well each of the two positive-evidence signals, counterfactual flip (after applying a surrogate fix, the candidate turns from failing to passing) and intent alignment (an LLM judges whether the candidate matches the intent described in the Intentlist), indicates true reproduction. We treat each signal as binary: on a given candidate test, it either fires or it does not. Here, true reproduction means the candidate satisfies $F \to P$ under the gold patch; this label is provided by the benchmark, is available only at evaluation time, and is independent of our evidence. We then compute the following metrics for each signal.

Firing rate. The fraction of candidate tests on which the signal fires, showing how common the signal is.

Precision. Among the candidate tests on which the signal fires, the fraction that are true reproductions, revealing how trustworthy "firing implies true reproduction" is.

Recall. Among all truly reproducing candidate tests, the fraction on which the signal fires, showing how many true reproductions the signal covers.

Discard Cost. Equal to $1 -$ Recall: the fraction of true reproductions we would wrongly discard if we rejected every candidate where the signal does not fire. It quantifies the cost of discarding such candidates, which is why Evidence-based Ranking only ranks candidate tests rather than rejecting them.

For RQ5, we additionally report a human intent-match rate on the fixed 100-instance subset, where two authors independently check whether the delivered test exercises the behavior described in the issue report and asserts the intended post-fix outcome; this audit is used only to validate the confidence ranking and is not used by VERIBRT.

D. Baselines

We prioritize baselines with strong reported performance on SWT-Bench while covering the main LLM-based BRT generation families. Other LLM-based methods in Section II are discussed to position the research landscape and are not exhaustively repeated in the main table. Following the experimental setup of prior work [11], [13]–[18], we compare VERIBRT against both prompt-only and LLM-agent baselines. We first consider the two prompt-only baselines introduced by SWT-Bench [11]: (i) **ZeroShot**, which prompts the LLM once with the issue description and a BM25-retrieved subset of the repository [28], asking it to generate a single test as a unified diff without agentic interaction or execution feedback; and (ii) **ZeroShotPlus**, which follows the same single-pass procedure but adopts SWT-Bench's custom diff format, designed specifically for LLMs and more tolerant of minor formatting errors. We further compare against four representative LLM-agent baselines: **OpenHands** [27], a general-purpose software engineering agent that also serves as the backbone of VERIBRT; **AssertFlip** [12], which first generates a passing test on the buggy version and then flips its assertions to produce a failing test; **Echo** [16], which combines graph-augmented repository retrieval with execution-feedback refinement to generate a single test per instance; and **e-Otter++** [14], which follows a localize-generate-refine pipeline with heterogeneous prompting and execution-based candidate selection.

We directly run all publicly available open-source baselines. Since OpenHands is designed for general-purpose SE tasks, we adapt only their task instructions for bug reproduction while keeping their underlying implementations unchanged. The OpenHands baseline therefore uses the same backbone model

and task instructions as VERIBRT but excludes all of our proposed modules, making it the starting point for the ablation study (Section VI-B). As e-Otter++ has not been publicly released, its results are taken directly from the SWT-Bench leaderboard under its official Claude 3.7 Sonnet configuration.

E. Configuration

Unless otherwise noted, VERIBRT's best-of-N sampling budget (the maximum number of candidate tests generated per instance) and surrogate fix budget (the maximum number of surrogate fixes used to probe a flip per candidate) both default to $N = K = 3$. Our budget sensitivity study (Section VI-C) shows that the $F \to P$ rate of both budgets saturates before reaching this default; setting the default past the saturation point safely preserves reproduction effectiveness while keeping generation and fixing cost low.

VI. EVALUATION RESULTS

A. RQ1: Overall effectiveness

Table I reports the performance of VERIBRT and the baselines on SWT-Bench Verified. Under DeepSeek-V4-Flash, VERIBRT achieves the highest $F \to P$ rate among same-model baselines, reproducing 341 of 433 instances and improving over its OpenHands backbone by 42 instances (+9.7 pp) and AssertFlip by 59 instances (+13.7 pp). The same-model comparison shows the same trend under GPT-5.4-mini and Gemini-3.1-Pro, where VERIBRT also achieves the highest $F \to P$ rate among methods using the same model. Since e-Otter++ is not publicly released, we report its official Claude 3.7 Sonnet leaderboard result only as a cross-model reference rather than a same-model comparison. Although we attempted to reproduce e-Otter++ with DeepSeek-V4-Flash following its official documentation, the reproduced performance was substantially lower than the reported results; therefore, we report only the official leaderboard results.

RQ2 decomposes the DeepSeek-V4-Flash gain over OpenHands across the three added components: Two-Axis Planning, the Validity Filter, and Evidence-based Ranking. VERIBRT's advantage is not accompanied by broader coverage of gold-patch-changed lines. Under DeepSeek-V4-Flash, its ΔCov is lower than Echo (65.0% vs. 71.3%) and AssertFlip (65.0% vs. 69.4%), even though its $F \to P$ rate is higher. This contrast is consistent with SWT-Bench's observation that executing changed code is insufficient when the generated assertion does not expose the reported bug [11]. RQ2–RQ5 therefore examine where the gains come from, whether they depend on budget, which failures remain, and how candidates are selected.

B. RQ2: Design principles

Table II reports a cumulative ablation under DeepSeek-V4-Flash. We add components in execution order rather than using leave-one-out ablation because downstream stages reuse the two-axis plan: the Validity Filter uses the Checklist for localization coverage, while Evidence-based Ranking uses the Intentlist for intent alignment and the Checklist for surrogate fixes. The cumulative ablation shows incremental gains as the three components are added in pipeline order. Two-Axis Planning adds 11 reproductions by making execution and assertion targets explicit; the Validity Filter adds 15 by retrying candidates with deterministic failures; and Evidence-based Ranking adds 16 by selecting admitted candidates with stronger counterfactual-flip and intent-alignment evidence. ΔCov does not increase monotonically, suggesting that the gain comes from better bug-reproduction behavior rather than broader coverage of gold-patch lines.

TABLE II
CUMULATIVE ABLATION ON SWT-BENCH VERIFIED.

	DeepSeek-V4-Flash		
Configuration	Instances	$F \to P$	ΔCov
OpenHands baseline	299	69.1	66.9
+ Two-Axis Plan	310	71.6	**70.6**
+ Two-Axis Plan + Validity Filter	325	75.1	64.8
+ Two-Axis Plan + Validity Filter + Evidence-based Ranking	**341**	**78.8**	65.0

TABLE III
$F \to P$ RATE PER REPOSITORY ON SWT-BENCH VERIFIED.

Repository	Instances	VERIBRT	e-Otter++	Echo	AssertFlip
astropy	17	70.6	58.8	47.1	**82.4**
django	216	**83.8**	60.2	64.8	62.0
matplotlib	32	**81.3**	75.0	59.4	65.6
mwaskom	2	**50.0**	0.0	0.0	0.0
pallets	1	**100.0**	**100.0**	**100.0**	0.0
psf	4	25.0	50.0	**75.0**	25.0
pydata	15	**86.7**	66.7	80.0	**86.7**
pylint-dev	6	50.0	**66.7**	0.0	33.3
pytest-dev	15	73.3	60.0	80.0	**86.7**
scikit-learn	24	**95.8**	83.3	79.2	83.3
sphinx-doc	28	**46.4**	17.9	32.1	28.6
sympy	73	**76.7**	65.8	63.0	**76.7**
Total instances	433	**341**	263	269	282

C. RQ3: Robustness

To test whether the RQ1–RQ2 gains mainly come from larger search budgets, we vary the best-of-N sampling budget N and the surrogate fix budget K on the fixed 100-instance subset. Figure 4 shows that increasing K from 1 to 10 only raises the flip signal's Firing rate from 30.0% to 34.0%, while Precision stays within 88%–90%; both metrics saturate by the default $K = 3$. With $K = 3$, Figure 5 shows that increasing N from 1 to 2 improves the $F \to P$ rate from 73.0% to 76.0%, after which the rate stays flat through $N = 5$. These trends suggest that the RQ1–RQ2 gains are not simply a product of larger search budgets; the default $N = K = 3$ lies within the saturated region.

D. RQ4: Success and failure analysis.

Table III reports the per-repository $F \to P$ rate on SWT-Bench Verified. VERIBRT achieves the best overall result (78.8%, 341/433), but its performance varies substantially across repositories. Among repositories with at least 15 instances, its rate ranges from 46.4% on *sphinx-doc* to 95.8% on

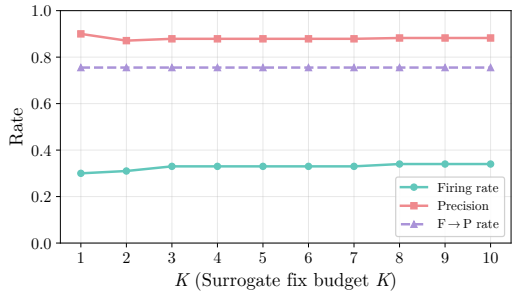

Fig. 4. Effect of the surrogate fix budget ($K = 1$ to 10, on the 100-instance subset).

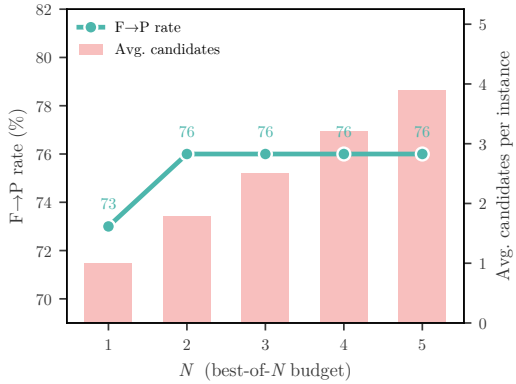

Fig. 5. Effect of the best-of-N budget (on the 100-instance subset).

scikit-learn. This spread suggests that repository-level characteristics and issue styles affect how easily the Checklist target and Intentlist observation point can be recovered. Among repositories with at least 15 instances, VERIBRT obtains the highest or tied-highest rate on six of eight repositories. Its strongest repositories include *scikit-learn* (95.8%), *pydata* (86.7%), and *django* (83.8%). These repositories appear to contain many issues with concrete API behavior or directly observable outputs. By contrast, *sphinx-doc* remains the lowest large repository (46.4%), suggesting a harder mode for output-oriented bugs whose expected behavior is expressed through rendered text, diagnostics, or formatting effects rather than a stable Intentlist observation point.

Figure 6 further characterizes the 92 unresolved instances under DeepSeek-V4-Flash by terminal failure mode. AssertionError dominates (59.8%, 55/92), whereas CoverageError accounts for only 4.3% (4/92), indicating that most residual failures come from wrong assertions rather than missing the Checklist target. Import and dependency failures account for another 14.1% (13/92), reflecting candidates that cannot be made harness-compatible within the budget.

Overall, RQ4 suggests that VERIBRT's main residual bottleneck is assertion correctness rather than target reachability, motivating the positive-evidence signals evaluated in RQ5.

E. RQ5: Evidence Quality

A valid candidate is not necessarily a true bug reproduction, so Evidence-based Ranking needs signals that are reliable when present but unsafe for rejection when absent. RQ5 therefore evaluates whether counterfactual flip and intent alignment

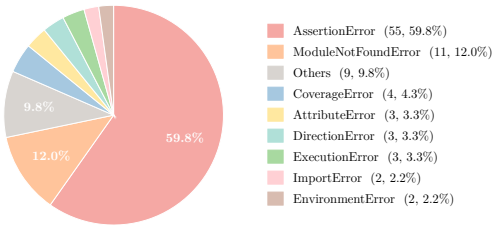

Fig. 6. Error-type distribution of the 92 unresolved instances.

TABLE IV
RELIABILITY OF THE TWO POSITIVE-EVIDENCE SIGNALS.

Signal	Firing rate	Precision	Recall	Discard Cost
counterfactual flip	35.2	90.1	40.2	59.8
intent alignment	87.5	80.4	89.1	10.9

have high Precision for ranking and high Discard Cost when used as rejection criteria (Table IV), and whether the resulting ranks align with automatic and human judgments (Table V).

Table IV shows that the two signals are complementary. Counterfactual flip is precise but sparse: its Precision is 90.1%, but its Firing rate is only 35.2% and its Discard Cost reaches 59.8%. Intent alignment fires more broadly (87.5%) and covers most true reproductions (Recall 89.1%), but with lower Precision (80.4%). Thus, both signals are useful for ranking when present, while their absence is too costly to use for rejection. Table V reports the rank distribution among the 400 candidates admitted by the Validity Filter; the remaining 33 instances are rejected before ranking. The automatic $F \rightarrow P$ rate rises monotonically from rank 1 to rank 3 (41.7% $\rightarrow$ 75.2% $\rightarrow$ 90.1%), showing that candidates with more positive evidence are more likely to reproduce the bug. The human audit on the fixed 100-instance subset shows the same trend: Human intent-match rises from 75.0% at rank 1 to 93.1% at rank 2 and 96.7% at rank 3. Because the audit is independent of Evidence-based Ranking, this agreement suggests that the confidence tiers reflect semantic bug reproduction rather than only the automatic verdict. Moreover, we also find that evidence-based ranking achieves the highest delivery rate on the 100-instance subset: 76.0%, compared with 73.0% for the first valid candidate and 74.0% for random selection.

VII. DISCUSSION

A. System-Specific Variations in Problem-Solving Capabilities

While $F \rightarrow P$ is widely used as the primary metric in bug reproduction benchmarks, it does not fully characterize the unique capabilities of different systems [11]. Figure 7 shows that BRT generation systems remain complementary. Across VERIBRT, Echo, AssertFlip, and the official e-Otter++ leaderboard result, the union covers 390 of 433 instances (90.1%), far above the best single-system result of 341 instances (78.8%). VERIBRT uniquely solves 26 instances, while Echo, AssertFlip, and e-Otter++ uniquely solve 11, 11, and 6 instances, respectively. This suggests that future BRT generators may benefit from combining complementary

TABLE V
CALIBRATION OF THE CONFIDENCE RANKING: UNDER THE AUTOMATIC VERDICT (400) AND THE HUMAN AUDIT (100).

	Automatic (400)		Human audit (100)	
Score	Instances	$F \to P$	Instances	Human intent-match
3	142	90.1	30	96.7
2	246	75.2	58	93.1
1	12	41.7	12	75.0

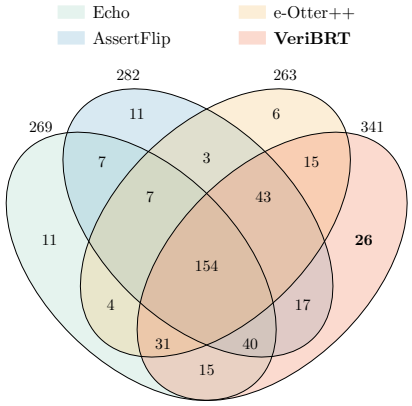

Fig. 7. Overlap of resolved instances among different systems.

generation strategies rather than only optimizing a single pipeline.

B. Comparison Across Different Difficulty Categories

SWT-Bench Verified divides all bugs into different difficulty levels based on the developers' estimated time to fix the bugs. We examined the performance of the method on those bugs considered difficult to fix. In software engineering practice, automated and high-quality bug reproduction is highly valuable, as existing studies have shown that reproduction significantly helps resolve bugs [40]. Table VI displays the total number of instances successfully reproduced by each method across different difficulty levels. The results indicate that VERIBRT demonstrates a strong edge across all difficulty intervals. It solves the most problems in the <15 min and 15 min–1 hour ranges. Even in the complex bug category with an estimated time as high as 1–4 hours, VERIBRT's advantage further expands, successfully reproducing 28 out of 36 high-difficulty problems, reaching a success rate of 77.8%. These findings indicate that VERIBRT is not only effective on simple bugs but also performs well when dealing with more complex problems.

VIII. THREATS TO VALIDITY

A. Internal Validity

We reduce implementation and environment bias by running all open baselines with the same SWT-Bench harness, model settings, and task inputs. For e-Otter++, whose implementation is not public, we use the official SWT-Bench leaderboard result and treat it as a cross-model reference. Our cumulative ablation does not isolate fully independent components, but this choice follows the pipeline dependency: downstream stages reuse the Checklist and Intentlist produced by Two-Axis Planning.

TABLE VI
COMPARISON OF SYSTEMS ACROSS DIFFICULTY CATEGORIES ON SWT-BENCH VERIFIED.

Approach	<15 min fix (172 total)	15 min–1 hour (225 total)	1–4 hours (36 total)
AssertFlip	128	140	14
Echo	115	136	18
e-Otter++	127	123	13
VERIBRT	**141**	**172**	**28**

For the human audit, two authors independently labeled the selected tests and resolved disagreements through discussion.

B. External Validity

We evaluate VERIBRT on SWT-Bench Verified, which covers repository-level Python problems from widely used projects, but it does not represent all types of bugs or all codebases. Although the methodology of VERIBRT is conceptually language-agnostic, extending it to other languages still requires rewriting the test runner and adjusting prompt templates, and the generalizability remains to be empirically validated. Second, as with other LLM-based open benchmark evaluations, the pre-training corpus of the model may contain problems, patches, or tests. We cannot fully control this. This is a limitation of all prior works using LLMs [13]–[17], [25]. We mitigate the fairness of the comparison by controlling variables, ensuring that both VERIBRT and all baselines share exactly the same model backend. The only exception is e-Otter++, whose results are taken directly from the official leaderboard under its Claude 3.7 Sonnet configuration and are included strictly as a cross-model reference. Fault localization is another critical factor. In this study, localization is not our research focus, and our primary results utilize the localization of BM25 [28]. In actual industrial projects, however, fluctuations in localization accuracy may have a certain impact on the reproduction success rate of VERIBRT.

IX. CONCLUSION

In this paper, we introduced VERIBRT, an LLM-agent framework for generating bug-reproducing tests from NL issue reports. VERIBRT extracts a two-axis behavioral plan, uses deterministic validity checks to trigger diagnostic retries, and reserves uncertain execution- and intent-based evidence for ranking admitted candidates rather than rejecting them. On SWT-Bench Verified, VERIBRT reproduces 341 of 433 instances under DeepSeek-V4-Flash, improving over its OpenHands backbone by 42 instances (+9.7 pp), and remains the strongest same-model method under GPT-5.4-mini and Gemini-3.1-Pro. The ablation and ranking analyses show that the gains come from making execution and assertion targets explicit and from using evidence that is reliable when present but too incomplete to serve as rejection criteria.

DATA AVAILABILITY

To support reproducibility, all related scripts and data are available in our anonymous repository: https://anonymous.4open.science/r/VeriBRT-D7E9.

REFERENCES

[1] W. Jin and A. Orso, "Bugredux: Reproducing field failures for in-house debugging," in *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, M. Glinz, G. C. Murphy, and M. Pezzè, Eds. IEEE Computer Society, 2012, pp. 474–484. [Online]. Available: https://doi.org/10.1109/ICSE.2012.6227168

[2] N. Chen and S. Kim, "STAR: stack trace based automatic crash reproduction via symbolic execution," *IEEE Trans. Software Eng.*, vol. 41, no. 2, pp. 198–220, 2015. [Online]. Available: https://doi.org/10.1109/TSE.2014.2363469

[3] A. C. Hora and G. Fraser, "Understanding bug-reproducing tests: A first empirical study," *CoRR*, vol. abs/2602.02965, 2026. [Online]. Available: https://doi.org/10.48550/arXiv.2602.02965

[4] R. O'Callahan, C. Jones, N. Froyd, K. Huey, A. Noll, and N. Partush, "Engineering record and replay for deployability: Extended technical report," *CoRR*, vol. abs/1705.05937, 2017. [Online]. Available: http://arxiv.org/abs/1705.05937

[5] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu, "PRES: probabilistic replay with execution sketching on multiprocessors," in *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*, J. N. Matthews and T. E. Anderson, Eds. ACM, 2009, pp. 177–192. [Online]. Available: https://doi.org/10.1145/1629575.1629593

[6] M. Soltani, A. Panichella, and A. van Deursen, "A guided genetic algorithm for automated crash reproduction," in *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, S. Uchitel, A. Orso, and M. P. Robillard, Eds. IEEE / ACM, 2017, pp. 209–220. [Online]. Available: https://doi.org/10.1109/ICSE.2017.27

[7] Y. Song and O. Chaparro, "BEE: a tool for structuring and analyzing bug reports," in *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, P. Devanbu, M. B. Cohen, and T. Zimmermann, Eds. ACM, 2020, pp. 1551–1555. [Online]. Available: https://doi.org/10.1145/3368089.3417928

[8] J. Zhou, H. Zhang, and D. Lo, "Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports," in *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, M. Glinz, G. C. Murphy, and M. Pezzè, Eds. IEEE Computer Society, 2012, pp. 14–24. [Online]. Available: https://doi.org/10.1109/ICSE.2012.6227210

[9] T. Zimmermann, R. Premraj, N. Bettenburg, S. Just, A. Schröter, and C. Weiss, "What makes a good bug report?" *IEEE Trans. Software Eng.*, vol. 36, no. 5, pp. 618–643, 2010. [Online]. Available: https://doi.org/10.1109/TSE.2010.63

[10] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. R. Narasimhan, "Swe-bench: Can language models resolve real-world github issues?" in *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024. [Online]. Available: https://openreview.net/forum?id=VTF8yNQM66

[11] N. Mündler, M. N. Müller, J. He, and M. T. Vechev, "Swt-bench: Testing and validating real-world bug-fixes with code agents," in *Advances in Neural Information Processing Systems 37: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, A. Globersons, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. M. Tomczak, and C. Zhang, Eds., 2024. [Online]. Available: http://papers.nips.cc/paper_files/paper/2024/hash/94f093b41fc2666376fb1f667fe282f3-Abstract-Conference.html

[12] L. Khatib, N. S. Mathews, and M. Nagappan, "Assertflip: Reproducing bugs via inversion of llm-generated passing tests," *CoRR*, vol. abs/2507.17542, 2025. [Online]. Available: https://doi.org/10.48550/arXiv.2507.17542

[13] T. Ahmed, J. Ganhotra, R. Pan, A. Shinnar, S. Sinha, and M. Hirzel, "Otter: Generating tests from issues to validate SWE patches," in *Forty-second International Conference on Machine Learning, ICML 2025, Vancouver, BC, Canada, July 13-19, 2025*, ser. Proceedings of Machine Learning Research, A. Singh, M. Fazel, D. Hsu, S. Lacoste-Julien, F. Berkenkamp, T. Maharaj, K. Wagstaff, and J. Zhu, Eds., vol. 267. PMLR / OpenReview.net, 2025. [Online]. Available: https://proceedings.mlr.press/v267/ahmed25b.html

[14] T. Ahmed, J. Ganhotra, A. Shinnar, and M. Hirzel, "Heterogeneous prompting and execution feedback for swe issue test generation and selection," 2026. [Online]. Available: https://arxiv.org/abs/2508.06365

[15] X. Wang, P. Gao, X. Meng, C. Peng, R. Hu, Y. Lin, and C. Gao, "AEGIS: an agent-based framework for general bug reproduction from issue descriptions," *CoRR*, vol. abs/2411.18015, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2411.18015

[16] Z. Fei, Y. Pan, F. Sarro, J. Ge, M. Liu, V. Ng, and H. Ye, "Echo: Graph-enhanced retrieval and execution feedback for issue reproduction test generation," *CoRR*, vol. abs/2603.07326, 2026. [Online]. Available: https://doi.org/10.48550/arXiv.2603.07326

[17] N. Nashid, I. Bouzenia, M. Pradel, and A. Mesbah, "Issue2test: Generating reproducing test cases from issue reports," *CoRR*, vol. abs/2503.16320, 2025. [Online]. Available: https://doi.org/10.48550/arXiv.2503.16320

[18] J. Wang, J. Cao, and Z. Liu, "icore: An iterative correlation-aware retriever for bug reproduction test generation," *CoRR*, vol. abs/2604.19224, 2026. [Online]. Available: https://doi.org/10.48550/arXiv.2604.19224

[19] DeepSeek-AI *et al.*, "Deepseek-v4: Towards highly efficient million-token context intelligence," 2026. [Online]. Available: https://arxiv.org/abs/2606.19348

[20] Y. Zhao, T. Yu, T. Su, Y. Liu, W. Zheng, J. Zhang, and W. G. J. Halfond, "Recdroid: automatically reproducing android application crashes from bug reports," in *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, J. M. Atlee, T. Bultan, and J. Whittle, Eds. IEEE / ACM, 2019, pp. 128–139. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00030

[21] K. Kitsios, M. Castelluccio, and A. Bacchelli, "Automated generation of issue-reproducing tests by combining llms and search-based testing," in *40th IEEE/ACM International Conference on Automated Software Engineering, ASE 2025, Seoul, Korea, Republic of, November 16-20, 2025*. IEEE, 2025, pp. 1982–1994. [Online]. Available: https://doi.org/10.1109/ASE63991.2025.00165

[22] M. Fazzini, M. Prammer, M. d'Amorim, and A. Orso, "Automatically translating bug reports into test cases for mobile apps," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, F. Tip and E. Bodden, Eds. ACM, 2018, pp. 141–152. [Online]. Available: https://doi.org/10.1145/3213846.3213869

[23] O. Chaparro, J. Lu, F. Zampetti, L. Moreno, M. D. Penta, A. Marcus, G. Bavota, and V. Ng, "Detecting missing information in bug descriptions," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, E. Bodden, W. Schäfer, A. van Deursen, and A. Zisman, Eds. ACM, 2017, pp. 396–407. [Online]. Available: https://doi.org/10.1145/3106237.3106285

[24] O. Chaparro, C. Bernal-Cárdenas, J. Lu, K. Moran, A. Marcus, M. D. Penta, D. Poshyvanyk, and V. Ng, "Assessing the quality of the steps to reproduce in bug reports," in *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, M. Dumas, D. Pfahl, S. Apel, and A. Russo, Eds. ACM, 2019, pp. 86–96. [Online]. Available: https://doi.org/10.1145/3338906.3338947

[25] S. Kang, J. Yoon, and S. Yoo, "Large language models are few-shot testers: Exploring llm-based general bug reproduction," in *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2023, pp. 2312–2323. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00194

[26] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE Trans. Software Eng.*, vol. 41, no. 5, pp. 507–525, 2015. [Online]. Available: https://doi.org/10.1109/TSE.2014.2372785

[27] X. Wang, B. Li, Y. Song, F. F. Xu, X. Tang, M. Zhuge, J. Pan, Y. Song, B. Li, J. Singh, H. H. Tran, F. Li, R. Ma, M. Zheng, B. Qian, Y. Shao, N. Muennighoff, Y. Zhang, B. Hui, J. Lin, and et al., "Openhands: An open platform for AI software developers as generalist agents," in *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net, 2025. [Online]. Available: https://openreview.net/forum?id=OJd3ayDDoF

[28] S. E. Robertson and H. Zaragoza, "The probabilistic relevance framework: BM25 and beyond," *Found. Trends Inf. Retr.*, vol. 3, no. 4, pp. 333–389, 2009. [Online]. Available: https://doi.org/10.1561/1500000019

[29] M. Konstantinou, R. Degiovanni, and M. Papadakis, "Do llms generate test oracles that capture the actual or the expected program behaviour?" *CoRR*, vol. abs/2410.21136, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2410.21136

[30] A. Bodicoat, G. Jahangirova, and V. Terragni, "Understanding llm-driven test oracle generation," *CoRR*, vol. abs/2601.05542, 2026. [Online]. Available: https://doi.org/10.48550/arXiv.2601.05542

[31] D. Agarwal *et al.*, "A taxonomy of failing bug reproduction tests on SWT-Bench," OpenReview, 2026, under review. Accessed: 2026-05-29. [Online]. Available: https://openreview.net/forum?id=OLHKkIidVs

[32] J. M. Voas, "PIE: A dynamic failure-based technique," *IEEE Trans. Software Eng.*, vol. 18, no. 8, pp. 717–727, 1992. [Online]. Available: https://doi.org/10.1109/32.153381

[33] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Trans. Software Eng.*, vol. 37, no. 5, pp. 649–678, 2011. [Online]. Available: https://doi.org/10.1109/TSE.2010.62

[34] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. L. Traon, and M. Harman, "Chapter six - mutation testing advances: An analysis and survey," *Adv. Comput.*, vol. 112, pp. 275–378, 2019. [Online]. Available: https://doi.org/10.1016/bs.adcom.2018.03.015

[35] I. Badertdinov, A. Golubev, M. Nekrashevich, A. Shevtsov, S. Karasik, A. Andriushchenko, M. Trofimova, D. Litvintseva, and B. Yangel, "Swe-rebench: An automated pipeline for task collection and decontaminated evaluation of software engineering agents," *CoRR*, vol. abs/2505.20411, 2025. [Online]. Available: https://doi.org/10.48550/arXiv.2505.20411

[36] L. Zheng, W. Chiang, Y. Sheng, S. Zhuang, Z. Wu, Y. Zhuang, Z. Lin, Z. Li, D. Li, E. P. Xing, H. Zhang, J. E. Gonzalez, and I. Stoica, "Judging llm-as-a-judge with mt-bench and chatbot arena," in *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, Eds., 2023. [Online]. Available: http://papers.nips.cc/paper_files/paper/2023/hash/91f18a1287b398d378ef22505bf41832-Abstract-Datasets_and_Benchmarks.html

[37] Z. Zhao, A. Esmaeili, and F. H. Fard, "Bias in the loop: Auditing llm-as-a-judge for software engineering," *CoRR*, vol. abs/2604.16790, 2026. [Online]. Available: https://doi.org/10.48550/arXiv.2604.16790

[38] OpenAI, "Introducing GPT-5.4 mini and nano," https://openai.com/index/introducing-gpt-5-4-mini-and-nano/, 2026, accessed: 2026-06-09.

[39] Google DeepMind, "Gemini 3.1 pro model card," Google DeepMind, Tech. Rep., February 2026, online; accessed July 1, 2026. [Online]. Available: https://storage.googleapis.com/deepmind-media/Model-Cards/Gemini-3-1-Pro-Model-Card.pdf

[40] M. Beller, N. Spruit, D. Spinellis, and A. Zaidman, "On the dichotomy of debugging behavior among programmers," in *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, M. Chaudron, I. Crnkovic, M. Chechik, and M. Harman, Eds. ACM, 2018, pp. 572–583. [Online]. Available: https://doi.org/10.1145/3180155.3180175