BUGWITGYM: EVALUATING CODING AGENTS FOR EXECUTABLE BUG VALIDATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Bug validation asks a coding agent to produce an executable witness for a reported bug. The witness combines a concrete input with a testing harness and exposes faulty behavior during execution. Such evidence makes audit findings actionable, yet benchmark evaluation is difficult when cases reuse public historical bugs and witnesses or require manual construction. We present BUGWITGYM, an automated framework for constructing bug-validation benchmarks through bug injection. It injects bugs into test-reached paths of real projects, rebuilds each project, and retains cases exposed by a construction-time witness. Bug specifications and execution adapters allow extension to additional bug types and languages. Bug-preserving transformations vary the surrounding structure while preserving the witness behavior. Based on real-world Java projects with test suites, BUGWITGYM automatically constructs 1,300 benchmark cases. The injected patches resemble historical bug patches and are difficult for the two evaluated models to distinguish in blinded comparisons. We evaluate four coding agent frameworks in six framework/model pairings across bug types, execution contexts, and transformation depths. Witness construction remains difficult even when the bug pattern is known. Our framework, benchmark cases, and evaluation scripts are available.

1 INTRODUCTION

Agentic coding systems increasingly address repository-level software-engineering tasks (Wang et al., 2025; Yang et al., 2024; Xia et al., 2025; Antoniades et al., 2025), making reliable code auditing an important capability (Guo et al., 2025; Wang et al., 2024). In real-world AI coding scenarios, useful audit findings require concrete evidence that developers can inspect and act on, in addition to a suspicious code location (Wang et al., 2026; Zhang et al., 2025a; Zhu et al., 2025; Cheng et al., 2025; Liu et al., 2025; Simsek et al., 2026; Peng et al., 2025). This requirement matters because bug reports generated by existing AI coding agents may still contain non-negligible false positives (Zhao et al., 2026; Du et al., 2025), and repeated false alarms can quickly erode developers' trust in the tool. For this reason, a useful coding agent should generate an executable witness that exposes the reported bug (Wang et al., 2026; Gezgin et al., 2026). An executable witness combines a concrete input with a testing harness that invokes the relevant code under the required environment and exposes an observable failure such as a crash or assertion violation. Such evidence helps developers distinguish real bugs from false alarms and decide whether a report deserves attention.

Existing benchmarks provide only partial support for evaluating bug validation on new cases in real project contexts. They typically rely on curated capture-the-flag (CTF) challenges or disclosed vulnerabilities, tying evaluation to manually assembled tasks or known bugs and witnesses (Figure 1a and b). For example, NYU CTF Bench (Shao et al., 2024) and Cybench (Zhang et al., 2025b) offer executable objectives in challenge-specific environments whose construction requires curation. Benchmarks based on CVEs (Zhu et al., 2025), bug-bounty reports (Zhang et al., 2025a), and fuzzing artifacts (Wang et al., 2026) retain real project context but draw their cases from historical vulnerabilities. Reusing these public artifacts limits the supply of new evaluation targets and creates opportunities for prior exposure to their solutions.

We target a benchmark that can generate new cases automatically as models evolve while preserving the contextual reasoning required in real code auditing (Wu et al., 2025; Pu et al., 2025; Saxon et al., 2024). These goals address both the cost of case construction and the dependence on existing

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

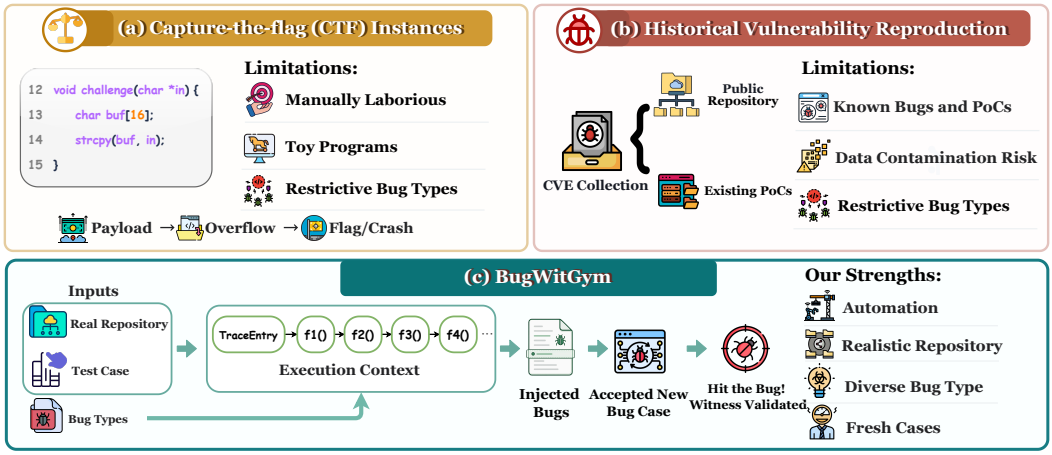

Figure 1: Comparison of benchmark-construction methodologies for executable bug validation.

solutions. Even when a static-analysis query reports a recognizable bug pattern and its location, validation requires concrete object states, call sequences, and path conditions that expose the bug through an observable failure.

We introduce BUGWITGYM, a framework that meets these goals through automated bug injection. To construct new cases, it adapts API Contract, Value Flow, and Logic bug types from the CodeQL Java query library into injection specifications (GitHub, 2026). These categories distinguish contract, data-state, and functional reasoning in witness construction (Section 3.2). To preserve real project context, it injects these bugs into production paths exercised by existing tests, rebuilds the project, and reruns the associated test. This test serves as a construction-time witness and must continue to expose the bug after transformations vary the surrounding call, data, and control structure. Its contents are removed before agent evaluation, leaving the agent to construct its own witness. Construction establishes that the bug is observable; evaluation measures whether an agent can independently recover the conditions needed to expose it.

We implement BUGWITGYM and automatically construct 1,300 benchmark cases using ten bug types in these three categories and six real-world Java projects with test suites, at a total cost of $16,500. We evaluate four agent frameworks instantiated in six framework/model pairings on their ability to construct executable witnesses for the benchmark cases. Our evaluation shows substantial remaining headroom on this task. The best-performing configuration, Codex with GPT-5.4, achieves 68.9% average validation success. Performance varies across agents and models, with a 50-percentage-point gap between the highest and lowest averages. Value Flow and Logic bugs are generally harder than API Contract bugs, and success decreases as execution contexts lengthen or transformation depth increases. The evaluation costs approximately $8,700.

This work makes three main contributions. First, we present BUGWITGYM, a systematic methodology for automatically constructing bug-validation benchmarks by injecting diverse bug types into real-world projects, producing new bug–witness pairs. Second, we instantiate this methodology using ten bug types across API Contract, Value Flow, and Logic and six real-world Java projects, yielding 1,300 execution-validated cases. Third, we evaluate coding agents on the resulting benchmark and analyze the factors that affect bug-validation performance.

2 BACKGROUND AND MOTIVATION

Bug Validation. Given a repository, a bug description, and a reported location, bug validation asks an agent to construct an executable witness. The witness combines a concrete input with a testing harness that invokes the relevant code under the required environment and exposes the bug through an exception, assertion violation, crash, or another observable failure.

This task turns a reported bug into evidence that developers can reproduce and act on. A recognizable pattern or a localized report leaves the concrete input, program state, and failure oracle to be

constructed. The agent must recover the relevant call sequence and path conditions, instantiate the required state, and expose the faulty behavior. For example, an API-contract violation may require a particular object state and call sequence before an assertion can expose the incorrect result. Evaluation therefore needs executable cases embedded in real program contexts, with a known failure that can serve as the validation oracle.

Existing Benchmarks. Table 1 compares the construction capabilities needed for this evaluation. Curated CTF tasks provide explicit objectives, but adding cases requires new challenges and associated environments (Shao et al., 2024; Zhang et al., 2025b). Historical bug-reproduction work, including LIBRO, Issue2Test, and CyberGym, retains real project context while relying on existing bug reports or disclosed vulnerabilities (Kang et al., 2023; Nashid et al., 2026; Wang et al., 2026). Its case supply is consequently tied to the availability of historical artifacts. Bug-injection methods automate construction: LAVA and EvilCoder introduce new vulnerabilities through specialized injection mechanisms, while FixReverter applies patterns learned from past fixes at new injection sites (Dolan-Gavitt et al., 2016; Pewny & Holz, 2016; Zhang et al., 2022). Their construction objectives give limited control over the surrounding program structure for comparing agents across validation difficulties.

Table 1: Comparison of bug-validation benchmark construction. **Diversity** is high for multiple bug families, medium for variants within a dominant family, and low for one narrow family. **Auto. cases** are constructed automatically; **fresh bugs** are generated rather than reused; **real projects** retain production code with its build and test environments; and **context control** varies surrounding call, data, or control structure.

Work	Diversity	Auto. cases	Fresh bugs	Real projects	Context control
CTF and security challenges					
NYU CTF Bench (2024)	High	✗	✗	✗	✗
Cybench (2025b)	High	✗	✗	✗	✗
Historical bug validation					
LIBRO (2023)	High	✗	✗	✓	✗
Issue2Test (2026)	High	✗	✗	✓	✗
CyberGym (2026)	Medium	✗	✗	✓	✗
Bug-injection methods					
LAVA (2016)	Low	✓	✓	✓	✗
EvilCoder (2016)	Low	✓	✓	✓	✗
FixReverter (2022)	Medium	✓	✓	✓	✗
BUGWITGYM (Our Work)	High	✓	✓	✓	✓

Construction Goals. We seek automatic construction of new bug cases in real projects, covering multiple bug families and retaining an executable oracle for every accepted case. Fresh cases reduce dependence on public bug–witness pairs, while controlled structural variation lets the benchmark probe how context affects validation. BUGWITGYM combines these properties by injecting bugs into test-reached code and applying transformations that preserve the observed failure. This yields new validation targets with realistic project dependencies and adjustable surrounding structure, without per-case manual bug design or witness authoring.

3 METHODOLOGY

This section introduces BUGWITGYM, a framework for constructing bug-validation benchmarks. Given a real-world project with an existing test suite and a library of bug types, BUGWITGYM injects bugs into test-reached production contexts. The associated test provides the harness and concrete input used to expose each injected bug; we call this artifact the *construction-time witness*. Its contents are removed before agent evaluation.

Figure 2 presents the BUGWITGYM workflow. Functions exercised by existing tests serve as injection contexts, providing concrete executions in which new bugs can be reached and observed. Each injection and transformation is validated by rebuilding the project and rerunning its construction-

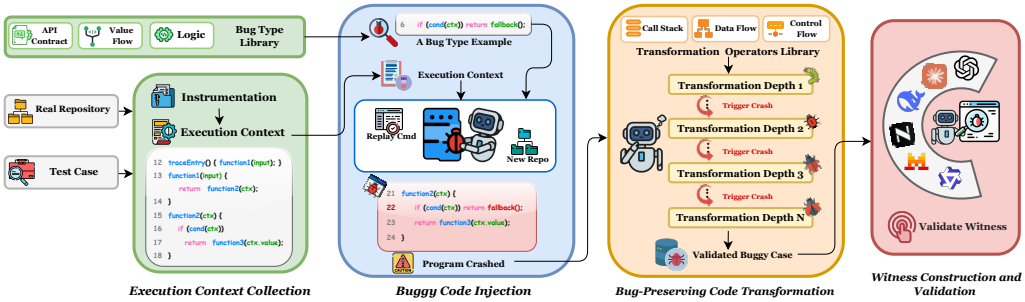

Figure 2: Overview of the BUGWITGYM construction and evaluation pipeline.

time witness. The pipeline comprises execution-context collection, buggy-code injection, and bug-preserving code transformation.

3.1 EXECUTION CONTEXT COLLECTION

Given a real-world project and its existing test suite, BUGWITGYM first identifies the functions exercised by each test case. This step provides a test-grounded search space for bug injection. To collect the executed functions during runtime, BUGWITGYM applies program instrumentation to the project. Specifically, it parses the source code of the project and inserts logging calls at the entry of each function. During test execution, these logging calls record the executed functions and their source locations. For a function f, we denote its source location as $\ell(f) = (\mathrm{name}(f), \mathrm{path}(f), \mathrm{line}(f))$, where $\mathrm{name}(f)$ is the function name, $\mathrm{path}(f)$ is the path of the source file containing f, and $\mathrm{line}(f)$ is the starting line number of f. Formally, executing a test case t yields an ordered execution trace $\tau_t = [\ell_1, \ell_2, \ldots, \ell_n]$, where each ℓ_i is an observed function location. We define the corresponding execution context as the set of unique locations in that trace, $EC_t = \{\ell_i \mid \ell_i \in \tau_t\}$. Thus, τ_t preserves order, whereas EC_t supports context membership and distinct-function counts. Repeating this process for all test cases produces an *execution context map*, $\mathcal{M}_{EC}$, which maps each test case to its execution context $\mathcal{M}_{EC}(t) = EC_t$.

The execution context map serves as the basis for selecting candidate injection locations. BUGWITGYM restricts the injection search space to functions executed by a specific test case. This design improves the likelihood that the injected bug is reachable and reduces the cost of subsequent execution-guided injection.

3.2 BUGGY CODE INJECTION

In the second stage, BUGWITGYM utilizes the collected execution contexts to synthesize benchmark cases via buggy code injection. We systematically cataloged test-observable bug patterns from CodeQL Java query examples (GitHub, 2026) and grouped them by the program property they violate: *API Contract* for API contracts or specifications, *Value Flow* for constraints on propagated values (e.g., dereferencing null values), and *Logic* for domain-specific functional behavior. This grouping separates the contract, data-state, and functional reasoning needed to construct witnesses. To bound construction and agent-evaluation cost, we randomly selected ten eligible types while retaining examples from each category. The construction loop takes a bug-type specification, a test-reached execution context, and a replayable test supplying the input and oracle. Additional types use the same interface with a specification whose failure is observable; new ecosystems also require suitable tracing and replay adapters. In our setting, accepted bugs are required to produce observable runtime failures when exercised, which provides an automatic execution oracle for validating witnesses. The bug types used in our benchmark construction are listed in Table 5 in Appendix C.

Based on the above ingredients, the buggy code injection can be formulated as follows. Given a test case t, an execution context EC_t, and a bug type b, BUGWITGYM leverages a coding agent to inject b into the execution context EC_t of test case t. Since the functions in EC_t are exercised, the injected bug is likely to be reachable by the construction-time witness derived from test case t. After each injection attempt, BUGWITGYM rebuilds the modified project and reruns the associated

test case t. If the test does not expose the bug, BUGWITGYM provides the execution feedback to the coding agent and requests a refinement. This process continues until the bug is observed or the attempt budget is exhausted. This procedure retains cases whose injected bug is present in production code and empirically exposed by the construction-time witness. The system prompt and user prompt used in buggy code injection are provided in Appendix I.1 and Appendix I.2.

3.3 BUG-PRESERVING CODE TRANSFORMATION

Since the bug types are derived from public CodeQL examples, directly reusing these snippets may produce benchmark cases that are syntactically or structurally similar to publicly available bug types. BUGWITGYM applies bug-preserving transformations that are similar in spirit to code obfuscation. Specifically, they alter the surface representation and contextual structure of the injected bug while preserving its behavior under the same construction-time witness. The transformations reduce surface similarity to public bug examples and increase contextual complexity while preserving the injected bug.

Technically, BUGWITGYM is built on three families of transformation operators, which manipulate call stack, data flow, and control flow structures. The detailed definitions of the transformation operators are listed in Table 6 in Appendix C. Similar to buggy code injection, the transformation stage is also execution-guided so that we can validate whether the transformed code preserves the injected bug. If not, the transformation is reverted. Through this process, BUGWITGYM can derive multiple benchmark variants from the same injected bug. These variants share an underlying bug type and construction-time witness while differing in call structure, data flow, and control flow. They provide repeated validation tasks under systematically varied structural contexts. The prompt of the bug-preserving code transformation is given in Appendix I.3.

4 EVALUATION

We instantiate BUGWITGYM on six real-world Maven-based Java projects using ten observable bug types adapted from the CodeQL Java query library (GitHub, 2026). The types span API Contract, Value Flow, and Logic bugs. The benchmark contains 1,300 buggy cases across four execution-context-length categories and four transformation-depth levels, and costs approximately $16,500 to construct. Multiple cases per type provide observations across repositories and construction settings. Appendix C and Appendix D give implementation details and benchmark statistics.

4.1 EVALUATION SETUP

Bug-Validation Task. We evaluate every case under two settings that correspond to two important software-engineering scenarios. In the context-agnostic setting, denoted NoEC, the agent receives the buggy repository, the natural language description of the bug type, the buggy location, but no execution context. This setting mimics the validation of static bug-detection results, where developers may have a localized bug report but no calling context showing how the buggy location is reached. In the context-guided setting, denoted WithEC, the agent receives the same inputs together with the execution context. This setting mimics the bug-reproduction and debugging workflows, where a crash stack or recorded execution trace is available as a dynamic hint. The evaluation begins from a supplied bug report and location and measures the subsequent construction of an executable witness. In the current benchmark instantiation, execution-context length ranges from 1 to 150 distinct production-side functions, grouped into four categories: *short* ($[1, 40]$), *mid-short* ($[41, 70]$), *mid-long* ($[71, 120]$), and *long* ($[121, 150]$). Appendix I.4 and Appendix I.5 provide the prompts used for executable bug validation.

Agent Settings. We evaluate six framework/model pairings: Codex and Claude Code with their hosted models, and OpenHands and OpenCode with four open-weight models. Table 2 lists the pairings and provider release identifiers (OpenAI, 2026; Anthropic, 2025; DeepSeek-AI, 2025; Z.AI, 2025; Mistral AI, 2025; Qwen Team, 2025).

Each case allows at most three validation attempts. Each attempt uses a 1,800-second agent timeout, a 1,200-second replay-verification timeout, a 600-second optional formatting timeout, and a 4,200-second case-level timeout. For OpenHands, we set the temperature to 0 and the maximum output length to 4,096 tokens. Codex, Claude Code, and OpenCode use interface-default decoding because their evaluated interfaces do not expose directly comparable controls. Runs that exceed the case timeout, fail replay verification, or do not produce a repository test containing an executable witness are counted as failures. The evaluation costs approximately $8,700; Appendix J provides reproduction commands.

Table 2: Evaluated framework/model pairings.

Agent	Framework	Model
A1	Codex	GPT-5.4
A2	Claude Code	Claude Sonnet 4.5
A3	OpenHands	DeepSeek-V3.2
A4	OpenCode	ZAI GLM-4.7
A5	OpenHands	Devstral-2-123B
A6	OpenCode	Qwen3-Coder-480B-A35B-Instruct

4.2 RQ1: How Well Do Coding Agents Validate Bugs through Execution?

To control evaluation cost, we randomly sample 900 cases from the 1,300-case benchmark and evaluate all six agents under both NoEC and WithEC. Table 3 shows substantial performance differences. Codex (A1) and Claude Code (A2) achieve average validation success rates of 68.9% and 59.4%, while OpenHands with DeepSeek-V3.2 (A3) exceeds OpenCode with Qwen3-Coder-480B-A35B-Instruct (A6) by 29.7 percentage points. The best-performing agent still fails on 31.1% of the cases.

Table 3: Agent validation success rates.

ID	NoEC	WithEC	Average
A1	67.1%	70.7%	68.9%
A2	57.1%	61.8%	59.4%
A3	46.0%	49.8%	47.9%
A4	44.2%	47.6%	45.9%
A5	30.2%	33.8%	32.0%
A6	17.1%	19.3%	18.2%

Execution context improves validation success by 2.2–4.7 percentage points across the six agents. These gains suggest that dynamic context helps agents identify relevant functions and paths for constructing executable witnesses. However, the improvements do not change the agent ranking, and large performance gaps remain under WithEC. Even with execution context, the best-performing agent succeeds on 70.7% of cases, leaving 29.3% unsolved within the evaluation budget. The supplied context can guide code inspection, but it does not provide a complete witness.

Finding 1: Current coding agents leave substantial room for improvement in executable bug validation, including when execution context is available.

4.3 RQ2: How Do Bug Types Affect Executable Bug Validation?

To investigate the effect of bug types, we analyze a 500-case subset drawn from the full benchmark. For each bug type, the subset contains 25 short-context cases and 25 long-context cases, all constructed with the same depth-3 transformation configuration and the same transformation operator sequence, and each case is evaluated under both the NoEC and WithEC settings. As shown in Table 4, difficulty varies substantially across bug types and agents. In particular, the Codex agent framework with GPT-5.4 (A1) and the Claude Code agent framework with Claude Sonnet 4.5 (A2) are relatively weak on Logic bugs, with WithEC success rates of 70.0% and 60.0%, respectively. The open-source agents exhibit a different profile. Their performance varies sharply across bug families, and they are generally less effective on Value Flow and Logic bugs than on API Contract bugs.

The gap between industrial and open-source agents varies by bug type. In particular, OpenHands with DeepSeek-V3.2 (A3) generally outperforms the other evaluated open-source agents with open-weight models. It narrows the gap to Claude Code with Claude Sonnet 4.5 (A2) on several Logic bug types, including Guard Predicate Inversion and In-Bounds Collection Misrouting. Under WithEC, A3 matches A2 on the former and slightly exceeds it on the latter. However, this pattern does not extend to all bug types, as A3 still trails A2 on both Value Flow types. These results show that open-source agent frameworks paired with strong open-weight models can approach industrial agents on specific bug types.

Table 4: Successful bug validations across bug types. L and S denote long and short execution contexts. Each A/B cell reports successful cases under WithEC and NoEC, respectively.

Bug Type	A1		A2		A3		A4		A5		A6	
	L	S	L	S	L	S	L	S	L	S	L	S
Comparison of Identical Values	18/17	22/21	17/14	19/16	14/10	15/15	11/11	15/13	9/7	10/9	4/4	6/5
Typo in `equals`	18/15	21/21	14/13	16/18	12/13	15/13	10/10	12/11	8/7	9/10	5/3	6/5
Typo in `hashCode`	16/13	18/17	16/11	19/14	10/9	15/12	11/8	13/14	7/6	10/7	4/4	5/5
Hashed Value Without `hashCode`	14/14	19/17	15/13	16/15	10/11	15/13	9/9	14/10	8/6	10/7	4/3	5/4
Inconsistent `equals` and `hashCode`	17/15	18/20	13/12	18/17	12/9	12/13	12/9	13/11	6/6	10/9	3/4	5/4
Intra-Function Null Dereference	19/16	23/23	14/13	19/15	9/9	12/11	11/9	13/12	9/8	10/11	7/6	8/6
Collection Element Null Dereference	16/16	20/17	14/12	16/17	10/8	12/11	8/7	11/10	8/6	11/8	5/4	8/8
Guard Predicate Inversion	16/14	18/16	13/13	17/14	13/10	17/13	12/9	14/13	7/7	8/8	4/3	5/5
Skipped Aggregation Counter	16/18	23/20	15/12	21/17	12/12	15/15	12/12	17/16	8/5	10/8	4/4	5/5
In-Bounds Collection Misrouting	15/12	17/18	11/11	13/15	12/12	14/15	12/11	14/13	6/6	8/7	4/3	5/5

We further quantify the effect of execution context across bug types. Figure 3 shows the difference in success rates between WithEC and NoEC for each agent–type pair. We hold the depth-3 transformation configuration fixed across bug types. The remaining differences therefore mainly reflect bug semantics and how well each agent uses the supplied context. The gains from execution context vary across agents and bug types. The remaining difficulty lies in reasoning about program properties within this context. Agents must use the supplied information to choose concrete inputs, set up the required program state, and write checks that expose the faulty behavior. Knowing which functions are involved does not by itself determine how to trigger the bug. Different bug types require different forms of reasoning. Agents and models therefore differ in how well they can turn the same contextual information into a working test. Benchmarks need multiple bug categories and types that test program properties at different levels of detail and complexity.

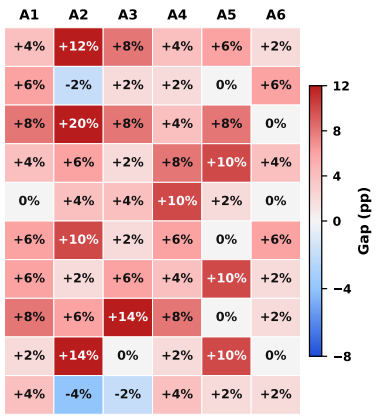

Figure 3: Success-rate gain with execution context (percentage points). Types follow Table 4.

Finding 2: Validation performance varies by bug type. Logic bugs are generally challenging, while open-source agent frameworks with open-weight models approach industrial agents on several specific bug types.

4.4 RQ3: How Does Bug-Validation Performance Vary with Execution-Context Length?

To investigate the effect of execution context length, we fix the bug type to Guard Predicate Inversion and analyze a 400-case subset drawn from the full benchmark. The subset is stratified by four execution-context-length buckets and four transformation depth settings, with 25 cases in each length-by-depth cell. For each depth setting, all cases use the same transformation-operator sequence, and each case is evaluated under both the NoEC and WithEC settings.

Figure 4a summarizes the WithEC results, and Table 7 in Appendix E reports both settings. Validation success is lower for longer execution contexts for every evaluated agent in both settings. Under WithEC, Codex with GPT-5.4 (A1) decreases from 80% on short contexts to 52% on long contexts, Claude Code with Claude Sonnet 4.5 (A2) decreases by 36 percentage points, and OpenHands with Devstral-2-123B (A5) decreases from 44% to 20%. Execution context can narrow the relevant program region, while longer contexts still require agents to reconstruct more intermediate states, setup choices, and dependency chains before exposing the bug through a valid test.

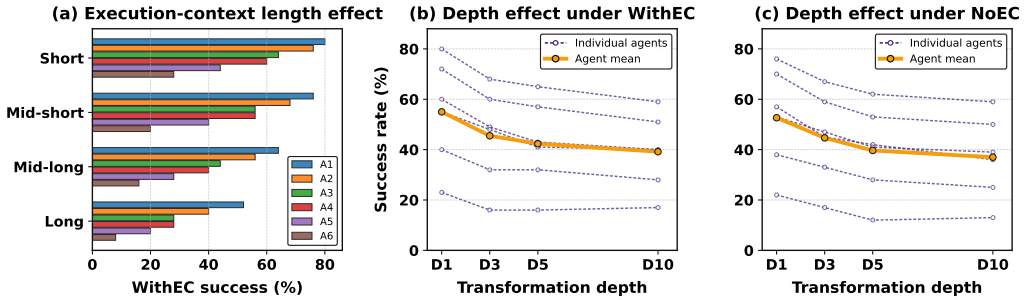

Figure 4: The effect of execution context length (a) and transformation depth (b and c).

Finding 3: Bug-validation success is substantially lower for longer execution contexts, including when execution context is provided.

4.5 RQ4: How Does Bug-Validation Performance Vary with Transformation Depth?

To investigate the effect of transformation depth, we fix the bug type to Guard Predicate Inversion and regroup the same 400-case subset by four transformation depths (1, 3, 5, and 10 applied transformations). Each depth-by-length cell contains 25 validated cases, and the transformation operator sequence is fixed. Each case is evaluated under both the NoEC and WithEC settings.

Figure 4b and c show lower validation success at greater transformation depths in both settings. Under WithEC, Codex with GPT-5.4 (A1) decreases from 80% at depth 1 to 59% at depth 10, Claude Code with Claude Sonnet 4.5 (A2) decreases from 72% to 51%, and OpenHands with DeepSeek-V3.2 (A3) decreases from 60% to 40%.

Deeper transformations can introduce additional code fragments, intermediate states, and dependency relations around the injected bug. The agent must reason through the resulting implementation to construct a valid setup and oracle. Success generally decreases with depth, with small plateaus or reversals in individual cells. These comparisons characterize the evaluated transformation sequences rather than the independent contribution of each operator. Table 8 in Appendix F reports detailed statistics across transformation-depth buckets. Appendix G provides further discussion of transformation composition and injection-stage patch evolution.

Finding 4: Greater transformation depth is generally associated with lower bug-validation success and provides a controllable benchmark-construction dimension.

4.6 RQ5: How Realistic Are the Injected Bug Patches?

The value of controlled bug injection depends on how closely its code changes resemble real development errors. We assess this resemblance through blinded source discrimination, using artifact-bearing positive controls to check judge sensitivity. We sample 700 of the 1,300 injected cases, stratified by repository and bug type, and match them to 700 historical bugs by repository and edit size. GPT-5.6-Terra and Claude Sonnet 4.6 independently see only blind IDs and production-code patches in the working-to-buggy direction.

Each judge performs *single-patch judgment* three times per patch, predicting its source (4,200 judgments), and *pairwise judgment* in both left–right orientations, selecting the more realistic patch (1,400 judgments). For 100 sampled pairs, we add conspicuous code artifacts to the injected patch while preserving its bug behavior and repeat the pairwise task (200 judgments). All calls use independent contexts, giving 11,600 judgments across both models.

Pairwise accuracy is the primary equivalence endpoint; single-patch accuracy and area under the receiver operating characteristic curve (ROC AUC) are auxiliary endpoints. Chance corresponds to 0.5. We use 10,000 whole-cluster bootstrap resamples, retaining related judgments together, and

require each endpoint's complete 90% confidence interval (CI) to lie within 0.45–0.55 (Lakens, 2017; Field & Welsh, 2007). Positive controls require accuracy of at least 80% and a 95% CI above 50%. Appendix H details the protocol and diagnostics.

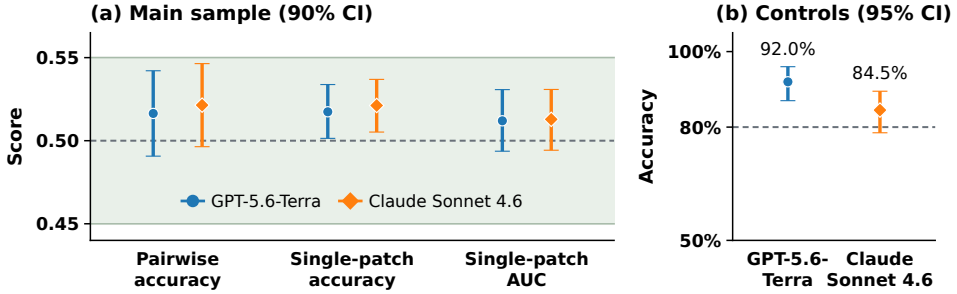

Figure 5: Blinded patch judgments. (a) Main-sample estimates and 90% cluster CIs; shading marks the 0.45–0.55 equivalence range and the dashed line marks chance. (b) Artifact-bearing controls with 95% CIs and the 80% accuracy threshold.

Figure 5 reports pairwise accuracies of 51.64% and 52.14%. Both judges meet the primary and auxiliary equivalence criteria. Their positive-control accuracies of 92.00% and 84.50% show sensitivity to conspicuous artifacts. The supplementary pairwise AUC interval for Claude Sonnet 4.6 slightly exceeds the equivalence bound; Appendix H reports this result and position preferences.

Finding 5: Injected patches resemble historical bugs in blinded comparisons, combining code-level realism with controlled construction for evaluating executable bug validation.

5 Related Work

Input Generation. Randoop, EvoSuite, and DART generate tests using execution feedback, coverage objectives, or path constraints (Pacheco et al., 2007; Fraser & Arcuri, 2011; Godefroid et al., 2005). KATCH, AFLGo, WAFLGo, CAFL, and SDFuzz direct execution using locations, patches, data constraints, or vulnerability states (Marinescu & Cadar, 2013; Böhme et al., 2017; Xiang et al., 2024; Lee et al., 2021; Li et al., 2024). Target-directed input generation shares the need to satisfy the conditions that expose a bug. BugWitGym evaluates repository cases with supplied bug descriptions and locations, asking agents to construct both the input and its execution harness.

Benchmarks. LIBRO, Issue2Test, and BRT Agent generate reproducing tests from bug reports (Kang et al., 2023; Nashid et al., 2026; Cheng et al., 2025); AnyPoC validates candidate reports, and Differential Prompting uses program pairs to generate failure-inducing tests (Zhao et al., 2026; Li et al., 2023). CyberGym (Wang et al., 2026), SEC-bench (Lee et al., 2025), SEC-bench Pro (Lee et al., 2026), and exploit generators (Simsek et al., 2026; Andersson et al., 2026; Zhang et al., 2026) evaluate vulnerability reproduction; SecCodeBench-V2 evaluates secure code generation and repair (Chen et al., 2026). Defects4J curates fault–test pairs, while LAVA, EvilCoder, and FixReverter inject bugs (Just et al., 2014; Dolan-Gavitt et al., 2016; Pewny & Holz, 2016; Zhang et al., 2022). BugWitGym creates multi-category targets with construction-time witnesses and controlled structural variants.

6 Conclusion

This paper presents BugWitGym, an automated framework for constructing benchmarks for executable bug validation. It injects bugs into real projects and retains cases with a witness that exposes the faulty behavior. Bug-preserving transformations vary the surrounding program structure, allowing validation to be studied across different contextual demands. Our evaluation shows that coding agents struggle to construct witnesses as execution contexts grow and transformations deepen. Blinded comparisons also find the injected patches difficult to distinguish from historical bugs under the evaluated protocol. BugWitGym offers a way to generate fresh validation tasks as agents evolve and study progress toward more reliable AI-assisted code auditing.

AI USE STATEMENT

We used large language models (LLMs) to assist with drafting and revising the manuscript, preparing figures, and planning experiments. Their use in benchmark construction and evaluation is described in Sections 3 and 4. The authors take full responsibility for the study design, reported results, and conclusions, as well as the accuracy of the final text and figures.

ETHICS STATEMENT

This work injects synthetic bugs into open-source repositories solely for benchmark construction; injected code stays in isolated working copies and is never released as patches to the original projects. The bug types are adapted from public examples and reflect common, previously documented defect patterns. No human subjects or personal data are involved and we see no ethical concerns.

REPRODUCIBILITY STATEMENT

The construction pipeline, agent configurations, and evaluation protocol are fully specified in the main text and appendix, including all prompts and output schemas, benchmark composition statistics, and the blinded-judgment statistical protocol. Step-by-step commands for verifying released cases, reconstructing the benchmark, and rerunning evaluation are also provided.

REFERENCES

Vivi Andersson, Sofia Bobadilla, Harald Hobbelhagen, and Martin Monperrus. PoCo: Agentic proof-of-concept exploit generation for smart contracts. *ACM Transactions on Software Engineering and Methodology*, 2026. doi: 10.1145/3816704.

Anthropic. Introducing Claude Sonnet 4.5, 2025. URL `https://www.anthropic.com/news/claude-sonnet-4-5`. Accessed September 24, 2026.

Antonis Antoniades, Albert Örwall, Kexun Zhang, Yuxi Xie, Anirudh Goyal, and William Wang. SWE-Search: Enhancing software agents with Monte Carlo tree search and iterative refinement. In *The Thirteenth International Conference on Learning Representations*, 2025.

Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2329–2344. ACM, 2017. doi: 10.1145/3133956.3134020.

Longfei Chen, Ji Zhao, Lanxiao Cui, Tong Su, Xingbo Pan, Ziyang Li, Yongxing Wu, Qijiang Cao, Qiyao Cai, Jing Zhang, Yuandong Ni, Junyao He, Zeyu Zhang, Chao Ge, Xuhuai Lu, Zeyu Gao, Yuxin Cui, Weisen Chen, Yuxuan Peng, Shengping Wang, Qi Li, Yukai Huang, Yukun Liu, Tuo Zhou, Terry Yue Zhuo, Junyang Lin, and Chao Zhang. SecCodeBench-V2 technical report. *arXiv preprint arXiv:2602.15485*, 2026.

Runxiang Cheng, Michele Tufano, Jürgen Cito, José Cambronero, Pat Rondon, Renyao Wei, Aaron Sun, and Satish Chandra. Agentic bug reproduction for effective automated program repair at Google. *arXiv preprint arXiv:2502.01821*, 2025.

DeepSeek-AI. DeepSeek-V3.2 model card, 2025. URL `https://huggingface.co/deepseek-ai/DeepSeek-V3.2`. Accessed September 24, 2026.

Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, William K. Robertson, Frederick Ulrich, and Ryan Whelan. LAVA: Large-scale automated vulnerability addition. In *IEEE Symposium on Security and Privacy, SP 2016*, pp. 110–121. IEEE Computer Society, 2016.

Xueying Du, Kai Yu, Chong Wang, Yi Zou, Wentai Deng, Zuoyu Ou, Xin Peng, Lingming Zhang, and Yiling Lou. Minimizing false positives in static bug detection via LLM-enhanced path feasibility analysis. *arXiv preprint arXiv:2506.10322*, 2025.

C. A. Field and A. H. Welsh. Bootstrapping clustered data. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 69(3):369–390, 2007. doi: 10.1111/j.1467-9868.2007.00593.x.

Gordon Fraser and Andrea Arcuri. EvoSuite: Automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, pp. 416–419. ACM, 2011. doi: 10.1145/2025113.2025179.

Derin Gezgin, Amartya Das, Shinhae Kim, Zhengdong Huang, Nevena Stojkovic, and Claire Wang. PoC-Gym: Towards more reliable LLM-assisted proof-of-concept exploit generation. *arXiv preprint arXiv:2602.04165*, 2026.

GitHub. CodeQL query help for Java and Kotlin. CodeQL documentation, 2026. URL `https://codeql.github.com/codeql-query-help/java/`. Accessed September 24, 2026.

Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 213–223. ACM, 2005. doi: 10.1145/1065010.1065036.

Jinyao Guo, Chengpeng Wang, Xiangzhe Xu, Zian Su, and Xiangyu Zhang. RepoAudit: An autonomous LLM-agent for repository-level code auditing. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 21083–21100. PMLR, 2025.

René Just, Darioush Jalali, and Michael D. Ernst. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *International Symposium on Software Testing and Analysis, ISSTA 2014*, pp. 437–440. ACM, 2014.

Sungmin Kang, Juyeon Yoon, and Shin Yoo. Large language models are few-shot testers: Exploring LLM-based general bug reproduction. In *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering*, pp. 2312–2323, 2023. doi: 10.1109/ICSE48619.2023.00194.

Daniël Lakens. Equivalence tests: A practical primer for t tests, correlations, and meta-analyses. *Social Psychological and Personality Science*, 8(4):355–362, 2017. doi: 10.1177/1948550617697177.

Gwangmu Lee, Woochul Shim, and Byoungyoung Lee. Constraint-guided directed greybox fuzzing. In *30th USENIX Security Symposium*, pp. 3559–3576. USENIX Association, 2021.

Hwiwon Lee, Ziqi Zhang, Hanxiao Lu, and Lingming Zhang. SEC-bench: Automated Benchmarking of LLM Agents on Real-World Software Security Tasks. In *Advances in Neural Information Processing Systems*, volume 38, pp. 128940–128976, 2025. doi: 10.52202/085713-3878.

Hwiwon Lee, Jiawei Liu, Dongjun Kim, Wubing Xia, Ziqi Zhang, Chunqiu Steven Xia, and Lingming Zhang. SEC-bench Pro: Can language models solve long-horizon software security tasks? *arXiv preprint arXiv:2605.26548*, 2026.

Penghui Li, Wei Meng, and Chao Zhang. SDFuzz: Target states driven directed fuzzing. In *33rd USENIX Security Symposium*, pp. 2441–2457. USENIX Association, 2024.

Tsz-On Li, Wenxi Zong, Yibo Wang, Haoye Tian, Ying Wang, Shing-Chi Cheung, and Jeff Kramer. Nuances are the key: Unlocking ChatGPT to find failure-inducing tests with differential prompting. In *38th IEEE/ACM International Conference on Automated Software Engineering*, pp. 14–26, 2023. doi: 10.1109/ASE56229.2023.00089.

Kaibo Liu, Zhenpeng Chen, Yiyang Liu, Jie M Zhang, Mark Harman, Yudong Han, Yun Ma, Yihong Dong, Ge Li, and Gang Huang. LLM-powered test case generation for detecting bugs in plausible programs. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 430–440, 2025.

Paul Dan Marinescu and Cristian Cadar. KATCH: High-coverage testing of software patches. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pp. 235–245. ACM, 2013. doi: 10.1145/2491411.2491438.

Mistral AI. Devstral-2-123B-Instruct-2512 model card, 2025. URL https://huggingface.co/mistralai/Devstral-2-123B-Instruct-2512. Accessed September 24, 2026.

Noor Nashid, Islem Bouzenia, Michael Pradel, and Ali Mesbah. Issue2Test: Generating reproducing test cases from issue reports. In *Proceedings of the 48th IEEE/ACM International Conference on Software Engineering*, pp. 754–766, 2026. doi: 10.1145/3744916.3773129.

OpenAI. Introducing GPT-5.4, 2026. URL https://openai.com/index/introducing-gpt-5-4/. Accessed September 24, 2026.

Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *29th International Conference on Software Engineering*, pp. 75–84. IEEE, 2007. doi: 10.1109/ICSE.2007.37.

Wanzong Peng, Lin Ye, Xuetao Du, Hongli Zhang, Dongyang Zhan, Yunting Zhang, Yicheng Guo, and Chen Zhang. PwnGPT: Automatic exploit generation based on large language models. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2025, Vienna, Austria, July 27 - August 1, 2025*, pp. 11481–11494. Association for Computational Linguistics, 2025.

Jannik Pewny and Thorsten Holz. EvilCoder: Automated bug insertion. In *Proceedings of the 32nd Annual Conference on Computer Security Applications, ACSAC 2016*, pp. 214–225. ACM, 2016.

Sophia Xiao Pu, Sitao Cheng, Xin Eric Wang, and William Yang Wang. Dynamic evaluation for oversensitivity in LLMs. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, pp. 2337–2344, 2025.

Qwen Team. Qwen3-Coder-480B-A35B-Instruct model card, 2025. URL https://huggingface.co/Qwen/Qwen3-Coder-480B-A35B-Instruct. Accessed September 24, 2026.

Michael Saxon, Ari Holtzman, Peter West, William Yang Wang, and Naomi Saphra. Benchmarks as microscopes: A call for model metrology. In *First Conference on Language Modeling*, 2024.

Donald J. Schuirmann. A comparison of the two one-sided tests procedure and the power approach for assessing the equivalence of average bioavailability. *Journal of Pharmacokinetics and Biopharmaceutics*, 15(6):657–680, 1987. doi: 10.1007/BF01068419.

Minghao Shao, Sofija Jancheska, Meet Udeshi, Brendan Dolan-Gavitt, Haoran Xi, Kimberly Milner, Boyuan Chen, Max Yin, Siddharth Garg, Prashanth Krishnamurthy, Farshad Khorrami, Ramesh Karri, and Muhammad Shafique. NYU CTF Bench: A Scalable Open-Source Benchmark Dataset for Evaluating LLMs in Offensive Security. In *Advances in Neural Information Processing Systems*, volume 37, pp. 57472–57498, 2024. doi: 10.52202/079017-1832.

Deniz Simsek, Aryaz Eghbali, and Michael Pradel. PoCGen: Generating proof-of-concept exploits for vulnerabilities in npm packages. *Proceedings of the ACM on Software Engineering*, 3(FSE): 3887–3908, 2026. doi: 10.1145/3808178.

Chengpeng Wang, Wuqi Zhang, Zian Su, Xiangzhe Xu, Xiaoheng Xie, and Xiangyu Zhang. LLMDFA: analyzing dataflow in code with large language models. In *Advances in Neural Information Processing Systems*, volume 37, pp. 131545–131574, 2024. doi: 10.52202/079017-4181.

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

Zhun Wang, Tianneng Shi, Jingxuan He, Matthew Cai, Jialin Zhang, and Dawn Song. CyberGym: Evaluating AI Agents' Real-World Cybersecurity Capabilities at Scale. In *The Fourteenth International Conference on Learning Representations*, 2026.

Xiaobao Wu, Liangming Pan, Yuxi Xie, Ruiwen Zhou, Shuai Zhao, Yubo Ma, Mingzhe Du, Rui Mao, Anh Tuan Luu, and William Yang Wang. AntiLeakBench: Preventing data contamination by automatically constructing benchmarks with updated real-world knowledge. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2025, Vienna, Austria, July 27 - August 1, 2025*, pp. 18403–18419. Association for Computational Linguistics, 2025.

Chunqiu Steven Xia, Zhe Wang, Yan Yang, Yuxiang Wei, and Lingming Zhang. Live-SWE-agent: Can software engineering agents self-evolve on the fly? *arXiv preprint arXiv:2511.13646*, 2025.

Yi Xiang, Xuhong Zhang, Peiyu Liu, Shouling Ji, Hong Liang, Jiacheng Xu, and Wenhai Wang. Critical code guided directed greybox fuzzing for commits. In *33rd USENIX Security Symposium*, pp. 2459–2474. USENIX Association, 2024.

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

Z.AI. GLM-4.7 model card, 2025. URL `https://huggingface.co/zai-org/GLM-4.7`. Accessed September 24, 2026.

Andy K. Zhang, Joey Ji, Celeste Menders, Riya Dulepet, Thomas Qin, Ron Y. Wang, Junrong Wu, Kyleen Liao, Jiliang Li, Jinghan Hu, Sara Hong, Nardos Demilew, Shivatmica Murgai, Jason Khiem Tran, Nishka Kacheria, Ethan Jun-shen Ho, Denis Liu, Lauren McLane, Olivia Beyer Bruvik, Dai-Rong Han, Seungwoo Kim, Akhil Vyas, Cuiyuanxiu Chen, Ryan Li, Weiran Xu, Jonathan Z. Ye, Prerit Choudhary, Siddharth M. Bhatia, Vikram Sivashankar, Yuxuan Bao, Dawn Song, Dan Boneh, Daniel E. Ho, and Percy Liang. BountyBench: Dollar Impact of AI Agent Attackers and Defenders on Real-World Cybersecurity Systems. In *Advances in Neural Information Processing Systems*, volume 38, 2025a. doi: 10.52202/085713-5725.

Andy K. Zhang, Neil Perry, Riya Dulepet, Joey Ji, Celeste Menders, Justin Lin, Eliot Jones, Gashon Hussein, Samantha Liu, Donovan Jasper, Pura Peetathawatchai, Ari Glenn, Vikram Sivashankar, Daniel Zamoshchin, Leo Glikbarg, Derek Askaryar, Haoxiang Yang, Aolin Zhang, Rishi Alluri, Nathan Tran, Rinnara Sangpisit, Kenny Oseleononmen, Dan Boneh, Daniel Ho, and Percy Liang. Cybench: A Framework for Evaluating Cybersecurity Capabilities and Risks of Language Models. In *The Thirteenth International Conference on Learning Representations*, 2025b. URL `https://openreview.net/forum?id=tc90LV0yRL`.

Jingwen Zhang, Yuhong Nan, Kaiwen Ning, Mingxi Ye, Wei Li, Yuming Xiao, Yuming Feng, Weizhe Zhang, and Zibin Zheng. V2E: Validating smart contract vulnerabilities through profit-driven exploit generation and execution. *Proceedings of the ACM on Software Engineering*, 3 (FSE):2259–2281, 2026. doi: 10.1145/3808108.

Zenong Zhang, Zach Patterson, Michael Hicks, and Shiyi Wei. FIXREVERTER: A realistic bug injection methodology for benchmarking fuzz testing. In *31st USENIX Security Symposium (USENIX Security 22)*, pp. 3699–3715. USENIX Association, 2022.

Zijie Zhao, Chenyuan Yang, Weidong Wang, Yihan Yang, Ziqi Zhang, and Lingming Zhang. AnyPoC: Universal proof-of-concept test generation for scalable LLM-based bug detection. *arXiv preprint arXiv:2604.11950*, 2026.

Yuxuan Zhu, Antony Kellermann, Dylan Bowman, Philip Li, Akul Gupta, Adarsh Danda, Richard Fang, Conner Jensen, Eric Ihli, Jason Benn, Jet Geronimo, Avi Dhir, Sudhit Rao, Kaicheng Yu, Twm Stone, and Daniel Kang. CVE-Bench: A Benchmark for AI Agents' Ability to Exploit Real-World Web Application Vulnerabilities. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 79850–79867. PMLR, 2025.

A LIMITATIONS

The current implementation covers ten observable bug types in Maven-based Java projects with replayable tests. Additional bug specifications and language-specific tracing and replay adapters would extend this coverage. Context-length and transformation-depth comparisons use a fixed bug type to control the target semantics; broader combinations would test how these factors interact across types. The blinded patch study characterizes source discrimination under the two reported judges and presentation protocol. Broader model coverage and developer assessments could examine other aspects of realism, including how injected bugs resemble those encountered during software maintenance.

B BROADER IMPACT

This work advances the evaluation of AI coding agents on executable bug validation in code-auditing scenarios. The injected bugs are isolated benchmark artifacts and are not changes proposed for deployed software. The framework can support research on evidence-grounded code auditing and on coding agents that validate reported bugs through execution.

Bug-validation performance varies with execution context length, transformation depth, and bug type across the evaluated agents. These dimensions support construction of cases with different reasoning requirements and difficulty levels.

The current implementation focuses on Maven-based Java projects and bug types derived from CodeQL Java queries. Applying the workflow to another ecosystem requires a runnable test environment, observable validation signals, and language-specific adapters. The benchmark methodology can support further study of bug validation for AI-assisted software engineering and security analysis.

C IMPLEMENTATION DETAILS

Project and Bug Type Preparation. We instantiate BUGWITGYM on six real-world Java projects spanning distributed middleware, data serialization, and multipurpose utility libraries. These projects provide injection sites across varied execution contexts. All selected projects use Maven as their build system, which allows an individual test case to be replayed with a localized command such as:

```
./mvnw -q -pl <module> -Dtest=<test-class> test.
```

We exclude tests that depend on external services, network access, nondeterministic timing, or excessive runtime. Each experiment starts from a clean repository template, and all injection, transformation, and evaluation runs are performed in isolated working copies. The current implementation targets Maven-based Java projects. Adapting the workflow to another language or build system requires isolated execution of a designated test and corresponding tracing, transformation, and validation adapters.

For bug type selection, we require the injected bug to produce an observable failure, such as an assertion failure or a runtime exception. Types whose effects are difficult to observe deterministically, such as pure performance regressions, are outside the scope of this benchmark. Among the eligible patterns surveyed from CodeQL Java query examples, we randomly selected ten types across the three categories to bound construction and evaluation cost. Representative types include Guard Predicate Inversion, Intra-Function Null Dereference, and Comparison of Identical Values. Table 5 lists all ten types organized by category.

Execution Context Collection. For each selected test case, we collect the ordered production-side execution path exercised by the test and summarize it as its execution context. To study how execution-context length affects bug validation, we stratify the collected execution contexts into length-based buckets. Here, execution context length refers to the number of distinct production-side functions reached during the test run, as recorded in the execution log τ_t. Specifically, we use four buckets: *short*, *mid-short*, *mid-long*, and *long*, with execution context lengths of $[1, 40]$, $[41, 70]$, $[71, 120]$, and $[121, 150]$ production functions, respectively. Shorter contexts correspond

Table 5: The ten bug types used in the benchmark construction.

Category	Type	Bug-type description
API Contract	Comparison of Identical Values	A condition compares a value with itself, producing a constant result that silently disables validation or equality logic.
	Typo in `equals`	A function overriding `equals(Object)` has the wrong name or signature, so Java keeps using reference equality.
	Typo in `hashCode`	A function intended to override `hashCode()` has the wrong name or signature, so hashed collections use identity hashing.
	Hashed Value Without `hashCode`	A class overrides `equals(Object)` but not `hashCode()`, so logically equal keys may hash to different buckets.
	Inconsistent `equals` and `hashCode`	`equals` and `hashCode` use different fields or unstable state, breaking lookup in `HashMap` or `HashSet`.
Value Flow	Intra-Function Null Dereference	A value that was checked for null is later overwritten, moved outside the guard, or dereferenced on an unguarded path.
	Collection Element Null Dereference	Maintenance or eviction leaves a null element in a list, map, or cache, and a later lookup or iteration assumes it is non-null.
Logic	Guard Predicate Inversion	A guard condition is flipped or weakened, sending valid inputs to an error or skip path, or letting invalid inputs continue.
	Skipped Aggregation Counter	A branch still processes an item but skips the counter or total update, so returned metrics undercount the work performed.
	In-Bounds Collection Misrouting	A loop writes an element to a legal but wrong index or key, leaving the collection complete but logically misordered.

to more constrained execution paths. Longer contexts capture broader execution paths that may traverse multiple production functions, classes, or modules.

Buggy Code Injection. We perform bug injection with Claude Code backed by Claude Sonnet 4.5. For each candidate case, the injection process is allowed up to three attempts. Before every attempt, the repository is restored to a clean workspace so that failed attempts do not accumulate state. A candidate is archived only if it satisfies three sequential validation checks. First, it must introduce a genuine production-code diff; cases with no change, formatting-only edits, or modifications confined to test or build files are rejected. Second, the diff must be structurally consistent with the intended bug type; we use lightweight syntactic checks to filter out obvious mismatches, such as injecting a null-pointer bug when the requested type is an equality inconsistency. Third, the replay command must pass on the clean baseline and expose the injected bug on the modified repository; cases that fail to compile, fail for unrelated infrastructure reasons, or do not expose the intended bug are discarded.

Bug-Preserving Transformation. After obtaining a valid injected case, we apply bug-preserving transformations, also using Claude Code backed by Claude Sonnet 4.5. We allow one attempt per transformation. To diversify the structural embedding of injected bugs, we define ten transformation operators across three families. Table 6 lists all ten operators. Each transformation prompt instructs the agent to rewrite the region containing the injected bug while preserving the construction-time witness. We apply 1, 3, 5, or 10 operators sequentially after injection. Each step rebuilds the project and reruns the replay command. A transformed case is retained only when the project remains valid and the same witness still exposes the injected bug. The transformations therefore vary program structure without repairing the bug.

Table 6: The ten bug-preserving transformation operators.

Family	Operator	Effect
Call Stack	Helper Call Insertion	Adds an internal helper or wrapper call between the original caller and the bug site, increasing call-chain depth.
	Call-Site Rerouting	Replaces a direct internal call with a forwarding function, so the same logic is reached through an extra dispatch step.
	Extract-and-Apply Split	Splits one computation into an extraction helper and an application helper, forcing the bug to cross function boundaries.
Data Flow	Parameter Bundling	Packs related arguments or locals into an internal object that is passed through helper calls instead of using separate values.
	Local State Bundling	Moves related local variables into a context object, so later code reads state through fields rather than direct locals.
	Intermediate State Passing	Stores intermediate values in a state holder and reads them later across helper calls, lengthening the value flow path.
	Cached Value Access	Routes reads of a derived value through a cached accessor instead of direct recomputation or direct field access.
	Nested Data Access	Replaces direct value access with access through a nested container, wrapper, map entry, or list element.
	Pipeline Stage Split	Breaks a multi-step function into ordered internal stages, making the bug appear inside a staged processing pipeline.
Control Flow	Rare Input Path Conditioning	Restricts the bug path to rare but deterministic input conditions, preserving normal behavior for common inputs.

D Detailed Benchmark Statistics

The constructed benchmark contains 1,300 archived buggy cases from six real-world Java projects. Each case stores a buggy repository snapshot, the execution context associated with the selected test, the applied bug type, the transformation sequence, and the construction-time witness used for validation. Figure 6 summarizes the benchmark composition. At the case level, the benchmark contains 201 execution contexts: 56 short, 45 mid-short, 45 mid-long, and 55 long contexts.

The benchmark covers all ten bug types in Table 5. Guard Predicate Inversion appears most frequently, with 688 cases, because it is used as the anchor type for systematically varying transformation depth and execution context length. The remaining nine types each appear in 68 cases, yielding balanced coverage over the other Logic, Value Flow, and API Contract bugs. At the category level, the benchmark therefore contains 824 Logic cases, 136 Value Flow cases, and 340 API Contract cases. Across all cases, the ten transformation operators in Table 6 appear 4,406 times in total. The most frequent operators are Call-Site Rerouting (1,028 occurrences), Helper Call Insertion (1,026), Local State Bundling (801), and Rare Input Path Conditioning (609), followed by Intermediate State Passing (218), Parameter Bundling (158), Cached Value Access (151), Nested Data Access (146), Pipeline Stage Split (141), and Extract-and-Apply Split (128).

Approximately 63% of attempted candidates pass the construction pipeline. The estimated cost includes failed attempts and successful bug injection, transformation, execution-context collection, replay validation, and construction-time witness validation. The end-to-end construction cost is approximately $16,500.

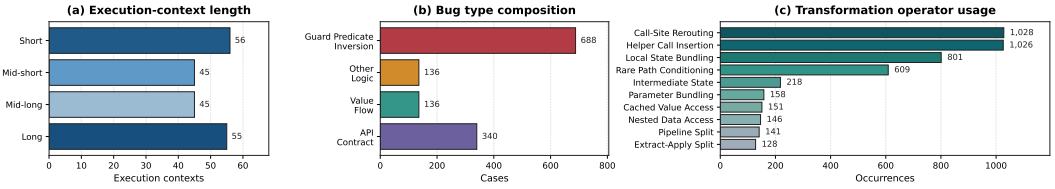

Figure 6: The distributions of execution context length, different categories of bug types, and the occurrence of transformation operators.

Table 7: Bug-validation success rates across execution context length buckets. Subscript annotations give percentage-point differences from the short bucket.

Agent ID	Short		Mid-Short		Mid-Long		Long	
	NoEC	WithEC	NoEC	WithEC	NoEC	WithEC	NoEC	WithEC
A1	80%	80%	72% $_{\downarrow 8pp}$	76% $_{\downarrow 4pp}$	64% $_{\downarrow 16pp}$	64% $_{\downarrow 16pp}$	48% $_{\downarrow 32pp}$	52% $_{\downarrow 28pp}$
A2	72%	76%	64% $_{\downarrow 8pp}$	68% $_{\downarrow 8pp}$	56% $_{\downarrow 16pp}$	56% $_{\downarrow 20pp}$	40% $_{\downarrow 32pp}$	40% $_{\downarrow 36pp}$
A3	60%	64%	52% $_{\downarrow 8pp}$	56% $_{\downarrow 8pp}$	40% $_{\downarrow 20pp}$	44% $_{\downarrow 20pp}$	28% $_{\downarrow 32pp}$	28% $_{\downarrow 36pp}$
A4	60%	60%	52% $_{\downarrow 8pp}$	56% $_{\downarrow 4pp}$	40% $_{\downarrow 20pp}$	40% $_{\downarrow 20pp}$	28% $_{\downarrow 32pp}$	28% $_{\downarrow 32pp}$
A5	44%	44%	36% $_{\downarrow 8pp}$	40% $_{\downarrow 4pp}$	28% $_{\downarrow 16pp}$	28% $_{\downarrow 16pp}$	16% $_{\downarrow 28pp}$	20% $_{\downarrow 24pp}$
A6	24%	28%	20% $_{\downarrow 4pp}$	20% $_{\downarrow 8pp}$	12% $_{\downarrow 12pp}$	16% $_{\downarrow 12pp}$	8% $_{\downarrow 16pp}$	8% $_{\downarrow 20pp}$

E Extended Analysis of Execution Context Effects

This appendix expands RQ3 using the same 400-case controlled subset of Guard Predicate Inversion used in the main text. The subset contains four execution context length buckets and four transformation depth buckets, with 25 validated cases for each bucket combination. Table 7 groups these cases by execution context length and aggregates over transformation depths.

Table 7 gives a finer-grained view of how execution-context length affects validation. Success decreases with longer contexts for every agent under both NoEC and WithEC. Under WithEC, A1 drops from 80% to 52%, A2 from 76% to 40%, A3 from 64% to 28%, and A4 from 60% to 28% between the short and long buckets.

Execution-context guidance improves several buckets, while success still declines as contexts grow. Under WithEC, A1 loses 28 percentage points and A2 loses 36 percentage points from short to long contexts. Longer contexts require agents to select relevant setup operations, preserve intermediate states, and construct an oracle that exposes the bug.

The decline is often progressive across buckets. Longer execution contexts expand the program behavior that a witness must reconstruct before the bug becomes observable.

F Extended Analysis of Transformation Depth Effects

This appendix expands RQ4 using the same 400-case controlled subset. The subset contains four transformation depth buckets and four execution context length buckets, with 25 validated cases for each bucket combination. Table 8 groups these cases by transformation depth and aggregates over execution context lengths.

Table 8 expands the transformation-depth analysis. Under WithEC, success from depth 1 to depth 10 decreases from 80% to 59% for A1, 72% to 51% for A2, 60% to 40% for A3, and 55% to 40% for A4. Transformation depth therefore provides a practical dimension for adjusting bug-validation difficulty.

Table 8: Bug-validation success rates across transformation-depth buckets. Subscript annotations give percentage-point differences from depth 1.

Agent ID	Depth 1		Depth 3		Depth 5		Depth 10	
	NoEC	WithEC	NoEC	WithEC	NoEC	WithEC	NoEC	WithEC
A1	76%	80%	67% $_{\downarrow 9pp}$	68% $_{\downarrow 12pp}$	62% $_{\downarrow 14pp}$	65% $_{\downarrow 15pp}$	59% $_{\downarrow 17pp}$	59% $_{\downarrow 21pp}$
A2	70%	72%	59% $_{\downarrow 11pp}$	60% $_{\downarrow 12pp}$	53% $_{\downarrow 17pp}$	57% $_{\downarrow 15pp}$	50% $_{\downarrow 20pp}$	51% $_{\downarrow 21pp}$
A3	57%	60%	45% $_{\downarrow 12pp}$	49% $_{\downarrow 11pp}$	42% $_{\downarrow 15pp}$	43% $_{\downarrow 17pp}$	36% $_{\downarrow 21pp}$	40% $_{\downarrow 20pp}$
A4	53%	55%	47% $_{\downarrow 6pp}$	48% $_{\downarrow 7pp}$	41% $_{\downarrow 12pp}$	41% $_{\downarrow 14pp}$	39% $_{\downarrow 14pp}$	40% $_{\downarrow 15pp}$
A5	38%	40%	33% $_{\downarrow 5pp}$	32% $_{\downarrow 8pp}$	28% $_{\downarrow 10pp}$	32% $_{\downarrow 8pp}$	25% $_{\downarrow 13pp}$	28% $_{\downarrow 12pp}$
A6	22%	23%	17% $_{\downarrow 5pp}$	16% $_{\downarrow 7pp}$	12% $_{\downarrow 10pp}$	16% $_{\downarrow 7pp}$	13% $_{\downarrow 9pp}$	17% $_{\downarrow 6pp}$

The effect of transformation depth differs from the effect of execution context length. Execution context length increases the amount of triggering behavior that the test must reconstruct, whereas transformation depth changes how the injected bug is structurally embedded in the code. As more transformations are applied, the same underlying bug may be surrounded by additional code fragments, modified control structure, or altered data dependencies. Consequently, the agent must reason through a transformed implementation before it can synthesize a valid setup and oracle. This makes transformation depth a construction-side complexity factor that complements execution context length.

Transformation depth yields an overall difficulty trend with small plateaus and reversals in individual cells. For example, under WithEC, OpenHands with Devstral-2-123B (A5) remains at 32% from depth 3 to depth 5, and OpenCode with Qwen3-Coder-480B-A35B-Instruct (A6) increases from 16% at depth 5 to 17% at depth 10. Transformations are compositional: each additional step can affect code shape, control flow, and data flow in different ways. Transformation count thus provides a coarse-grained difficulty control; finer-grained metrics would account for the structural and semantic effects of individual operators.

G Additional Analysis of Transformation Dimension Effects

Transformation composition. The transformation families change different aspects of the code surrounding an injected bug. Call-stack operators introduce helper boundaries or additional dispatch steps. Data-flow operators change how values are stored, accessed, and passed between functions, while control-flow operators change the conditions under which the buggy path is reached. These changes can affect the program state and call sequence that an agent must reconstruct before a test exposes the bug.

A transformed case may contain multiple operators, and later steps can reshape code introduced by earlier ones. The depth comparisons in RQ4 therefore concern the evaluated transformation sequences rather than the independent contribution of each operator. Grouping cases by operator occurrence alone does not separate these contributions. We examine the resulting patches below to describe how their size, structure, and retained content vary with transformation depth.

Injection-stage patch evolution. We analyze the injection-stage artifacts to examine how transformation depth changes the structure of the generated buggy patches. This analysis uses the same 400-case transformation-depth subset as RQ4, with one fixed bug type, four transformation-depth buckets, and 100 successfully validated injected cases per bucket. Across these cases, we analyze 1,900 step-level transformation patches and the final buggy patch retained for each case. The analysis complements the validation results by describing the code changes produced during construction.

Figure 7 and Table 9 show that greater transformation depth is associated with larger final patches and more hunks, without substantially spreading edits across more files. The mean final patch size grows from 65.0 non-empty diff lines at depth 1 to 243.7 at depth 10, while the mean hunk count grows from 2.33 to 6.09. In contrast, the mean number of changed files remains nearly flat, from

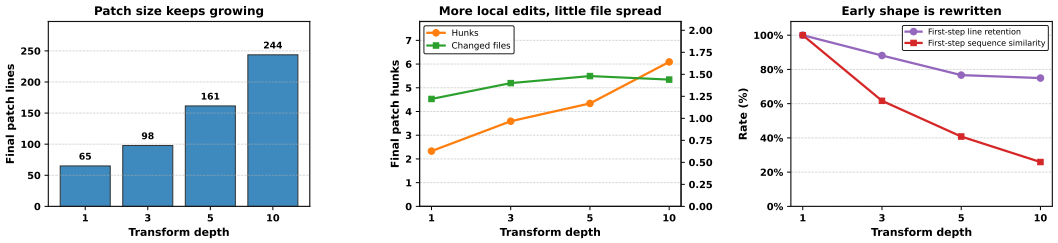

Figure 7: Injection-stage patch evolution by transformation depth. Final patch size and hunk count grow with depth, while the number of changed files remains nearly flat.

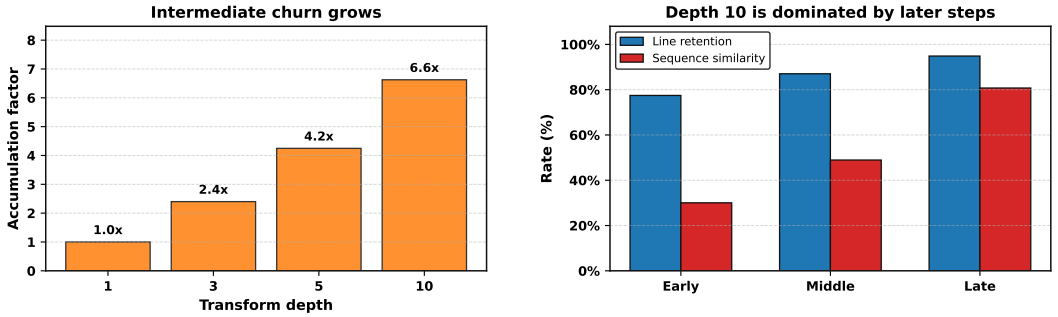

Figure 8: Intermediate edit churn and step-position effects. At depth 10, later transformation steps have higher line retention and sequence similarity to the final buggy patch than earlier steps.

1.22 to 1.44. The additional edits therefore remain concentrated within a small number of files, even as the final patch becomes larger and more fragmented.

Transformation depth also changes how earlier edits appear in the final buggy patch. First-step added-line retention remains relatively high, dropping from 100.0% at depth 1 to 74.9% at depth 10. However, first-step sequence similarity drops much more sharply, from 100.0% to 25.9%. Early transformation content often remains partly present even when its original arrangement changes. This pattern is consistent with later operators wrapping, splitting, moving, or rewriting earlier edits. The line-accumulation factor rises from 1.00× at depth 1 to 6.63× at depth 10, indicating that intermediate patches contain substantially more changed-line content in total than the final retained patch.

Figure 8 further shows that later steps are more closely reflected in the final patch at depth 10. Early steps retain 77.5% of added lines on average, with 30.0% sequence similarity, while late steps reach 94.9% line retention and 80.7% sequence similarity. These observations are consistent with later transformations retaining parts of earlier edits while reshaping their structure. They also show why transformation depth should not be interpreted as a simple accumulation of unchanged edits.

Together, these results provide a structural interpretation of the difficulty trend in RQ4. Deeper transformation sequences produce larger, more fragmented patches, while later steps change the arrangement of earlier edits. These changes may increase the code and state relationships that an agent must reason through when constructing an executable witness. The patch metrics describe the evaluated sequences for the fixed bug type; they do not by themselves establish which structural changes cause validation failures or isolate the effect of an individual operator.

H Blinded Patch Realism Study

H.1 Samples and Judgment Protocol

We sample 700 of the 1,300 injected cases, stratified by repository and bug type. Historical controls are matched by repository and edit size, also considering bug category, changed files, and hunks.

Table 9: Injection-stage patch-evolution metrics by transformation depth. Line accumulation is the ratio between cumulative step-patch lines and final-patch lines. Retention and sequence similarity are measured from the first transformation step to the final buggy patch.

Depth	Runs	Final lines	Hunks	Changed files	Line accum.	First-step retention	First-step similarity
1	100	65.0	2.33	1.22	1.00×	100.0%	100.0%
3	100	97.8	3.59	1.40	2.40×	88.1%	61.7%
5	100	161.5	4.34	1.48	4.25×	76.7%	40.8%
10	100	243.7	6.09	1.44	6.63×	74.9%	25.9%

This matching aims to reduce differences in repository style and patch size that could otherwise guide source judgments. Historical bug records establish provenance; pure refactorings, feature additions, and formatting changes are excluded. All production-code diffs use the working-to-buggy direction, reversing historical fixes where needed. We standardize formatting and context length, show blind IDs, and withhold commit metadata and source labels. In-patch comments, helpers, and structure remain visible so that judgments can still reflect code-level signs of artificial edits.

Judge A is GPT-5.6-Terra with high reasoning effort; judge B is Claude Sonnet 4.6. Both judges receive identical instructions and materials in randomized order. Each judgment uses a fresh context, with no shared responses or source retrieval.

Each patch receives three single-patch judgments, each producing an injected probability and a brief rationale. Probabilities of at least 0.5 yield the injected label. Each pair receives two judgments with positions swapped, each producing a left-more-realistic probability and a rationale. Probabilities of at least 0.5 select the left patch, and selecting the historical patch is scored as correct. Each model makes 4,200 single-patch and 1,400 main pairwise judgments.

Positive Controls. Before judging, we sample 100 pairs and add unnecessary helpers, redundant wrappers, or unnatural comments to the injected patches, retaining their historical counterparts. Construction checks confirm that the bug behavior is preserved. The added artifacts contain no source labels, and controls are not selected based on judge success. These controls test whether the judges can recognize conspicuous artificial edits under the same blinded protocol. Evaluating both orientations adds 200 judgments per model, giving 5,800 judgments per model and 11,600 overall. The sensitivity check requires accuracy $\geq 80\%$ and a 95% CI lower bound $> 50\%$.

H.2 Statistical Analysis

Pairwise accuracy is the primary endpoint. For the two correctness indicators $y_{i1}, y_{i2} \in \{0, 1\}$ of pair i, we compute $s_i = (y_{i1} + y_{i2})/2$ and average these scores over pairs. Single-patch accuracy and AUC are auxiliary endpoints, with injected as the positive class. Supplementary pairwise AUC uses the left-more-realistic probability and real-on-the-left as the positive class.

The equivalence hypothesis is $0.45 < \theta < 0.55$, against $\theta \leq 0.45$ or $\theta \geq 0.55$. This study-specific margin allows deviations from chance of less than five percentage points. Under the two one-sided testing principle ($\alpha = 0.05$), each judge's complete 90% CI must lie strictly within these bounds (Lakens, 2017; Schuirmann, 1987). An interval that merely includes chance is not sufficient to establish equivalence. Approximate intervals use 10,000 whole-cluster bootstrap resamples (Field & Welsh, 2007), grouping patch repetitions, pair orientations, and cases sharing an original injection or historical control. This procedure keeps related judgments together rather than treating repeated calls as independent observations. Positive controls use 95% CIs, and the judges are analyzed separately.

Table 10: Detailed source-discrimination results. Accuracy CIs are percentages.

Task	Judge	Correct / Total	Accuracy	90% CI	AUC	90% CI
Single	A	2,173 / 4,200	51.74%	50.14–53.38	0.51198	0.49367–0.53075
Single	B	2,189 / 4,200	52.12%	50.52–53.69	0.51286	0.49426–0.53084
Pairwise	A	723 / 1,400	51.64%	49.07–54.21	0.51681	0.48703–0.54633
Pairwise	B	730 / 1,400	52.14%	49.64–54.64	0.52411	0.49517–0.55252

Table 11: Artifact-bearing positive controls. Both judges meet the prespecified sensitivity check.

Judge	Correct / Total	Incorrect / Total	Accuracy	95% CI	Check
A	184 / 200	16 / 200	92.00%	87.00%–96.00%	Pass
B	169 / 200	31 / 200	84.50%	78.50%–89.50%	Pass

Tables 10 and 11 show that both judges meet the primary and auxiliary equivalence criteria and pass the sensitivity check. The contrast between the main sample and positive controls suggests that the judges can detect conspicuous artifacts but have limited ability to distinguish the matched injected and historical patches. Judge B's supplementary pairwise AUC interval reaches 0.55252, slightly above the upper equivalence bound. Equivalence is therefore not established for this additional metric. These findings concern source discrimination by the two evaluated judges under this protocol; they do not establish equivalence between the broader distributions of injected and historical bugs.

H.3 Preference and Stability Diagnostics

In Figure 9, *split* means different underlying patch choices after swapping positions, not an explicit tie. For real-both, split, and injected-both counts R, S, I, correct judgments total $2R + S$. These totals are 723/730 for the main sample and 184/169 for controls (A/B).

Table 12: Judgment stability, position preferences, and correctness counts (1,400 patches per judge).

Diagnostic	Judge A	Judge B
Single-patch repeat agreement	60.62%	58.67%
Patches with non-unanimous single judgments	59.07%	62.00%
Main orientation-choice agreement	68.43%	64.29%
Main LEFT choice rate	52.93%	54.71%
Main LEFT choices / judgments	741 / 1,400	766 / 1,400
Main split pairs choosing LEFT twice	131	158
Patches with 0 / 1 correct single judgments	277 / 369	247 / 402
Patches with 2 / 3 correct single judgments	458 / 296	466 / 285

Repeat agreement averages the three pairwise comparisons of each patch's single-patch judgments. Orientation agreement measures whether a judge selects the same underlying patch after positions are swapped. More than half the patches receive non-unanimous single-patch labels, showing substantial variation across repeated judgments. Table 12 also shows a left-choice tendency, especially for judge B. Evaluating both orientations balances the position of the historical patch, but does not establish position invariance. The aggregate equivalence results should therefore not be interpreted as stable decisions on every individual patch.

For the per-patch counts $N_0, \ldots, N_3$ in Table 12, correct judgments equal $N_1 + 2N_2 + 3N_3$. Repeat agreement equals $[3(N_0 + N_3) + N_1 + N_2]/4{,}200$, since unanimous labels agree in all three comparisons and non-unanimous labels agree in only one.

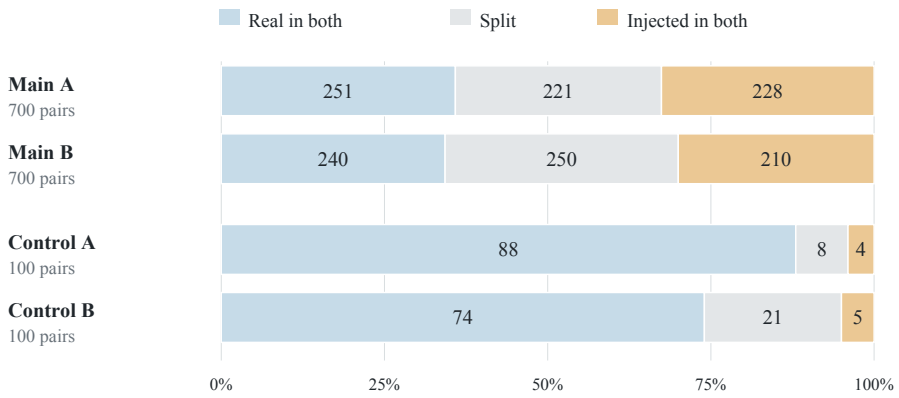

Figure 9: Preferences across both orientations. Numbers give pair counts; segment widths show percentages. Main groups contain 700 pairs and control groups contain 100. "Split" denotes inconsistent choices, not an explicit tie.

I PROMPT DESIGN

We report the prompt structures used in BUGWITGYM. The implementation uses layered prompts: a reusable system prompt specifies global behavioral constraints, while a dynamically assembled task payload injects case-specific fields such as the bug type, bug type specification, transformation operator, execution context, replay command, candidate anchors, and file scope.

For readability, repository-local paths and long execution context lists are shortened in this appendix, but the task goals, constraints, input fields, and required output formats follow the actual pipeline.

Table 13: Layered prompt structures used by BUGWITGYM during benchmark construction and executable bug validation.

Prompt Structure	**Main Inputs**	**Required Output**
Bug Injection System Prompt	Global injection rules, replay-command discipline, bounded edit–run–revise loop, diff checks, and success criteria	Single-line `INJECT_REPORT_JSON` with operational evidence fields
Bug Injection Task Payload	Bug type, bug type specification, required and forbidden elements, execution context, candidate anchors, trigger hints, and file scope	Same `INJECT_REPORT_JSON` prefix with type-site rationale fields
Bug-Preserving Transformation Task Payload	Transformation operator specification, applicability requirements, preservation constraints, execution context, and candidate anchors	Single-line `TRANSFORM_REPORT_JSON`
Bug Validation System Prompt	Test-only edit constraints, Java test-case materialization rules, executable-witness objective, and evaluation-attempt policy	Single-line `TRIGGER_REPORT_JSON` fallback schema
Bug Validation Task Payload	Buggy repository metadata, case ID, bug type, applied transformations, bug-related files, replay command, and optional execution context	Single-line `TRIGGER_REPORT_JSON` with detailed witness-construction fields

I.1 Bug Injection System Prompt

Bug Injection System Prompt

Mission. Introduce a real, localized bug into production code so that the externally provided replay command fails for the intended bug type. The repository must end with a non-empty semantic git diff, and the replay command must fail because of the injected bug.

Core rules.

1. Produce a non-empty semantic production-code diff.
2. Run the replay command exactly as provided, without changing the module, flags, test target, or command structure.
3. Do not modify tests unless explicitly allowed by the task.
4. Do not modify build files, dependency files, CI files, or repository configuration.
5. Do not change public or protected API signatures.
6. Keep the edit localized, realistic, and consistent with the repository style.
7. Do not claim success if the git diff is empty or if the observed failure is unrelated to the injected bug.

Construction-side execution budget. During one bug-injection session, the implementation permits at most three runs of the provided replay command. If the replay command still passes, the agent must revise the production-code edit and rerun the same command within this budget. This construction-side replay budget is separate from the bug-validation attempt budget used in the evaluation.

Required final output. The system prompt requires exactly one final single-line JSON object prefixed by `INJECT_REPORT_JSON`. The operational evidence fields include: `ok`, `git_changed_files`, `repro_cmd`, `repro_exit_code`, `observed_exception`, `call_chain`, and `notes`.

I.2 Bug Injection Task Payload

Bug Injection Task Payload

Task-specific inputs.

- Bug type and bug type ID.
- Bug type name, category, scope, and language.
- Bug type description.
- Required elements that must appear in the patch.
- Forbidden elements that must not appear in the patch.
- Expected failure signal and trigger-condition hints.
- Execution context summary collected from the target test.
- Candidate anchors with trace-relevant methods, files, and candidate edit sites.
- Allowed or preferred production-code file scope.

Bug type contract. The injected bug must match the provided bug type specification rather than merely resemble it. Required elements must be visible in the actual patch, and forbidden elements must not appear in the final patch. The chosen site should be trace-relevant and should allow the provided replay command to trigger the injected bug after validation.

Task-level reporting fields. The task payload uses the same `INJECT_REPORT_JSON` prefix and adds bug-type-site rationale fields: `ok`, `type_id`, `target_files`, `target_symbols`, `selected_anchor`, `why_this_site`, `expected_trigger_path`, `stealth_rationale`, `bug_mechanism`, `preserved_prior_structure`, and `possible_risks`. The implementation field `stealth_rationale` records the localization and plausibility of the edit.

I.3 Bug-Preserving Transformation Task Payload

Bug-Preserving Transformation Task Payload

The transformation prompt is a dynamically assembled task payload generated from an internal transformation operator specification. The payload uses implementation-level fields such as `id`, `name`, `intent`, `description`, `applicability`, `forbidden_elements`, `must_preserve`, and `post_conditions`. Each internal operator is mapped to a paper-facing transformation operator and family. For example, `t_rare_profile_gate_v1` is reported as Rare Input Path Conditioning,

`t_call_stack_deepen_v1` is reported as Helper Call Insertion, and `t_private_adapter_layer_v1` is reported as a call-site rerouting / private-adapter transformation.

Task. Apply one bug-preserving transformation step after successful bug injection. The transformation should alter the surrounding production-code structure while preserving compilation and preserving the replay behavior that exposes the injected bug. It is bug-preserving in the benchmark-construction sense: it preserves the relevant trigger path and bug-inducing behavior, but it is not required to preserve the full behavior of the original unmodified program.

Inputs.

- Internal transformation ID and operator name.
- Paper-facing operator term and transformation family.
- Operator intent and description.
- Applicability requirements.
- Forbidden edits.
- Must-preserve constraints, including compilation success, public API signatures, behavior for non-trigger inputs when applicable, and replay-command triggerability.
- Post-conditions expected from the operator.
- Execution context summary and candidate anchors.

Edit boundaries.

1. Do not modify tests unless explicitly allowed by the operator.
2. Do not change public or protected API signatures.
3. Do not perform broad refactors, renames, or formatting sweeps.
4. Keep the transformation localized to trace-relevant production code.
5. Preserve the injected bug-triggering behavior under the same replay command.

Required final output. The payload requires exactly one final single-line JSON object prefixed by `TRANSFORM_REPORT_JSON`. The JSON object contains: `ok`, `transform_id`, `changed_files`, `edit_sites`, `what_was_added_or_wrapped`, `what_was_preserved`, `what_was_replaced_or_overwritten`, `compatibility_with_previous_steps`, `expected_effect_on_stealth`, and `expected_effect_on_trigger_depth`. The field `expected_effect_on_stealth` is the implementation field name for the expected surface-form and review-plausibility effect of the transformation.

I.4 Bug Validation System Prompt

Bug Validation System Prompt

Mission. Create a Java test that exposes an already-injected production bug under the provided Maven replay command. The agent's job is not to fix code, improve the repository, or write a generic regression test. The only goal is to materialize a test that exposes the injected bug.

Evaluation attempt budget. For each benchmark case, the evaluation protocol allows at most three bug-validation attempts. Each attempt must follow the same test-only constraints and use the provided replay command as the validation target. A run succeeds only if the generated witness test compiles, the replay command exposes the injected bug, and no production file or build configuration is modified.

Hard constraints.

1. Never modify production code.
2. Never modify build files, dependency files, CI files, configuration files, or repository metadata.
3. Only create or edit Java test files under `src/test/java`.
4. Prefer rewriting the official test file at the provided path and with the provided package and class name.
5. The witness test must materialize in the repository as a real `.java` file before verification.
6. The test must be designed to trigger the injected bug, not to validate normal behavior.
7. Do not delete or modify unrelated files.

Required final output. The system prompt requires a final single-line `TRIGGER_REPORT_JSON`. The fallback schema includes: `ok`, `test_strategy`, `target_bug_path_hypothesis`, `why_trigger_should_fail`, and `materialized_test_files`.

I.5 Bug Validation Task Payload

Bug Validation Task Payload

Task-specific inputs.

- Repository name.
- Evaluation group or context condition.
- Injected case run ID.
- Bug type ID and bug type.
- Applied bug-preserving transformations.
- Bug-related production files and location metadata.
- Injection-stage verification exit code.
- Maven replay command used for validation.
- Natural-language bug summary.
- Target test path and target test class when available.

Execution context handling. The two evaluation settings are NoEC and WithEC. In WithEC, the task payload includes the execution context summary collected during benchmark construction. In NoEC, this execution context block is omitted. Neither setting provides the construction-time witness, hidden validation oracle, construction-time solution code, or permission to modify production code.

Required final output. The task payload asks for a detailed final `TRIGGER_REPORT_JSON` schema with: `summary`, `suspected_bug_mechanism`, `inspected_production_files`, `inspected_test_files`, `selected_test_path`, `selected_test_class`, `trigger_strategy`, `expected_failure_signal`, `materialized_test_files`, and `notes`.

I.6 End-to-End Case Prompt Example

This example shows a concrete case-level prompt flow assembled by the pipeline. The case uses repository `dubbo`, bug type Guard Predicate Inversion, and a depth-3 transformation chain. Long execution context lists are shortened for page fit, but the fields and constraints shown below follow the actual prompt structure.

Case Metadata

```
run_id: inject__20260315T064448Z
repository: dubbo
trace_group: short
trace_id: org.apache.dubbo.common.url.URLParamTest
bug_type: Guard Predicate Inversion
type_id: logic_predicate_inversion_in_guard
transform_depth: 3
transform_ids:
  - t_rare_profile_gate_v1
  - t_call_stack_deepen_v1
  - t_private_adapter_layer_v1
paper_facing_transform_terms:
  - Rare Input Path Conditioning
  - Helper Call Insertion
  - Call-Site Rerouting / Private Adapter Layer
official_test_path: dubbo-common/src/test/java/org/apache/dubbo/common/url/URLParamTest.java
official_test_class: org.apache.dubbo.common.url.URLParamTest
replay_command: ./mvnw -q -pl dubbo-common -Dtest=org.apache.dubbo.common.url.URLParamTest test
context_setting_for_executable_validation: WithEC
evaluation_attempt_budget: at most 3 bug-validation attempts per benchmark case
```

Bug Injection Prompt

```
SYSTEM:
You are operating inside a Git repository in an automated bug-injection pipeline.

MISSION:
Introduce a real, localized bug into production code so that the externally provided reproduction
     command fails for the intended bug type. Your work is only valid if the repository ends the
     session with a non-empty semantic git diff and the provided reproduction command, run verbatim,
     fails because of the injected bug.
```

```
NON-NEGOTIABLE RULES:
- You must produce a non-empty semantic git diff in the current repository state before finalizing.
- Semantic git diff means a real logic-changing edit, not whitespace-only, formatting-only, comment-
    only, import-only, rename-only, or mechanically trivial changes.
- You must run the reproduction command exactly as provided in the task, verbatim.
- Do not invent, replace, shorten, simplify, or substitute any command, module, flag, path, or test
    target.
- Do not modify test code unless the task explicitly allows it.
- Do not change public or protected API signatures.
- Do not claim success based on a hypothetical edit.
- Do not output a final success JSON if git diff is empty.
- Do not output a final success JSON if the reproduction command in your JSON differs from the
    externally provided command.

CONSTRUCTION-SIDE EXECUTION BUDGET:
- Up to 3 total runs of the provided reproduction command in the same injection session.
- If the reproduction command passes, make another production-code edit and rerun the same command
    verbatim within this budget.

TASK PAYLOAD:
You are performing controlled BUG INJECTION into production code for research benchmarking.
Goal: introduce a subtle, realistic bug consistent with the specified bug type, while keeping changes
    localized and review-plausible.

TYPE:
- id: logic_predicate_inversion_in_guard
- name: Predicate inversion in critical guard
- category: Logic
- scope: intra_method_branch_condition
- language: java

Description:
A boolean predicate used to guard important logic is inverted or altered in a subtle way. The code
    still compiles, branches look plausible, but the semantics flip which path handles which inputs,
     causing assertions and invariants to fail.

Required elements:
- if_or_else_block_guarding_non_trivial_logic
- condition_based_on_collection_size_or_status_flag
- tests_that_distinguish_between_valid_and_invalid_input_cases

Forbidden elements:
- conditions_that_are_constant_true_or_false_after_modification
- completely_removing_the_guard_and_inlining_the_body
- branches_with_no_observable_effect

Runtime effect hints:
- exception_type_hint: none
- typical_failure_signal: assertion failure or invalid return value

Trigger condition hints:
- test_shape_hint: tests feed different valid/invalid input categories and assert different behavior
- input_properties_hint: at least one test relies on the original guard semantics

TRACE CONTEXT SUMMARY:
{
  "trace_id": "org.apache.dubbo.common.url.URLParamTest",
  "test_name": "org.apache.dubbo.common.url.URLParamTest",
  "methods_count": 91
}

CANDIDATE ANCHORS:
- method: URLParamTest
  file: dubbo-common/src/test/java/org/apache/dubbo/common/url/URLParamTest.java
  role: target test
- method: hasMethodParameter
  file: dubbo-common/src/main/java/org/apache/dubbo/common/url/component/URLParam.java
  role: trace-relevant guard candidate
- method: getMethodParameter
  file: dubbo-common/src/main/java/org/apache/dubbo/common/url/component/URLParam.java
  role: trace-relevant lookup candidate
- method: getMethodParameterStrict
  file: dubbo-common/src/main/java/org/apache/dubbo/common/url/component/URLParam.java
  role: trace-relevant lookup candidate
- method: initMethodParameters
  file: dubbo-common/src/main/java/org/apache/dubbo/common/url/component/URLParam.java
  role: method-parameter initialization path
- method: size
```

```
  file: dubbo-common/src/main/java/org/apache/dubbo/common/url/component/URLParam.java
  role: status/collection-size related helper
- method: isEmpty
  file: dubbo-common/src/main/java/org/apache/dubbo/common/url/component/URLParam.java
  role: status/collection-size related helper

DELIVERABLE:
1. Implement the bug strictly according to the bug type contract.
2. Keep the patch minimal and consistent with repository style.
3. Run the provided reproduction command verbatim.
4. Provide evidence in the final JSON.

REQUIRED FINAL OUTPUT:
Append exactly one final line with this prefix:
INJECT_REPORT_JSON: { ... }

Required field groups:
- operational evidence: ok, git_changed_files, repro_cmd, repro_exit_code, observed_exception,
      call_chain, notes
- type-site rationale: type_id, target_files, target_symbols, selected_anchor, why_this_site,
      expected_trigger_path, stealth_rationale, bug_mechanism, preserved_prior_structure,
      possible_risks
```

Bug Injection Report

```
INJECT_REPORT_JSON:
{
  "ok": true,
  "type_id": "logic_predicate_inversion_in_guard",
  "target_files": [
    "dubbo-common/src/main/java/org/apache/dubbo/common/url/component/URLParam.java"
  ],
  "target_symbols": [
    "getAnyMethodParameter"
  ],
  "selected_anchor": "getAnyMethodParameter",
  "why_this_site": "The method is reached by URLParamTest and contains a critical guard over method-
      specific parameter lookup.",
  "expected_trigger_path": "URLParamTest -> URLParam.getAnyMethodParameter -> inverted guard returns
      null instead of the expected method parameter.",
  "stealth_rationale": "The edit is a localized predicate substitution inside an existing defensive
      check.",
  "bug_mechanism": "A guard condition is inverted from a non-empty map check to an empty map check,
      swapping valid and invalid lookup paths.",
  "preserved_prior_structure": "Method signatures, surrounding branch structure, and other production
      logic remain unchanged.",
  "possible_risks": "The bug is most visible through method-parameter tests that expect lookup results
       from non-empty maps."
}
```

Bug-Preserving Transformation Prompt

```
TASK:
Apply one bug-preserving transformation step after successful bug injection.
The transformation must keep the repository compilable and preserve the replay behavior that exposes
      the injected bug.

TRANSFORM SPEC 1:
- id: t_rare_profile_gate_v1
- paper_facing_operator: Rare Input Path Conditioning
- family: Control Flow
- name: Gate misroute behind rare profile
- description: restrict the misroute to a deterministic but low-probability input profile
- applicability: at_least_one_mid_level_helper_candidate
- forbidden_edits: changes_to_public_API_signatures, obvious_dead_branches_such_as_if_false
- must_preserve: compilation_success, public_api_signatures, replay_command_triggerability
- post_conditions: rare deterministic gate on the bug-related path

TRANSFORM SPEC 2:
- id: t_call_stack_deepen_v1
- paper_facing_operator: Helper Call Insertion
- family: Call Stack
- name: Deepen call stack with a natural helper extraction
```

```
- description: introduce a small helper method or lightweight wrapper to deepen the call stack
- applicability: at_least_one_mid_level_helper_candidate
- forbidden_edits: changes_to_public_API_signatures, broad refactors, formatting sweeps
- must_preserve: compilation_success, public_api_signatures, behavior_for_non_trigger_inputs
- post_conditions: additional_helper_in_call_chain

TRANSFORM SPEC 3:
- id: t_private_adapter_layer_v1
- paper_facing_operator: Call-Site Rerouting / Private Adapter Layer
- family: Call Stack
- name: Add a private adapter layer and reroute one call site
- description: introduce a private adapter method and reroute at least one internal call site through
    it
- applicability: at_least_one_internal_call_site_can_be_rerouted
- forbidden_edits: changes_to_public_API_signatures, obvious_dead_branches_such_as_if_false
- must_preserve: compilation_success, public_api_signatures, behavior_for_non_trigger_inputs
- post_conditions: additional_adapter_in_call_chain

TRACE CONTEXT SUMMARY:
{
  "trace_id": "org.apache.dubbo.common.url.URLParamTest",
  "test_name": "org.apache.dubbo.common.url.URLParamTest",
  "methods_count": 91
}

CANDIDATE ANCHORS:
- URLParam.hasMethodParameter
- URLParam.getMethodParameter
- URLParam.getMethodParameterStrict
- URLParam.initMethodParameters
- URLParam.size
- URLParam.isEmpty

EDIT BOUNDARIES:
- Do not modify tests unless explicitly allowed by the transform specification.
- Do not change public or protected API signatures.
- Avoid broad refactors, renames, or formatting sweeps.
- Keep diffs localized to trace-relevant production code.
- Preserve replay-command triggerability.

REQUIRED FINAL OUTPUT:
Append exactly one final line with this prefix:
TRANSFORM_REPORT_JSON: { ... }

Required keys:
ok, transform_id, changed_files, edit_sites, what_was_added_or_wrapped, what_was_preserved,
     what_was_replaced_or_overwritten, compatibility_with_previous_steps, expected_effect_on_stealth,
      expected_effect_on_trigger_depth
```

Bug-Preserving Transformation Report

```
TRANSFORM_REPORT_JSON:
{
  "ok": true,
  "transform_id": "t_rare_profile_gate_v1 + t_call_stack_deepen_v1 + t_private_adapter_layer_v1",
  "changed_files": [
    "dubbo-common/src/main/java/org/apache/dubbo/common/url/component/URLParam.java"
  ],
  "edit_sites": [
    "URLParam.getMethodParameter",
    "URLParam.hasMethodParameter",
    "URLParam.keyIndexToOffset",
    "URLParam.computeValueOffset",
    "URLParam.resolveParameterWithEncoding"
  ],
  "what_was_added_or_wrapped": "Added private helper and adapter methods around parameter lookup and
     offset resolution.",
  "what_was_preserved": "Public APIs, compilation, and replay-command triggerability.",
  "what_was_replaced_or_overwritten": "Direct parameter and offset resolution call sites were rerouted
      through private helpers.",
  "compatibility_with_previous_steps": "The transformation chain preserves the injected guard-
     inversion behavior.",
  "expected_effect_on_stealth": "The resulting structure resembles localized helper extraction and
     profile-aware lookup logic.",
  "expected_effect_on_trigger_depth": "The execution path now includes additional helper and adapter
     calls before the wrong value is observed."
```

```
}
```

Bug Validation Prompt

```
SYSTEM:
You are an autonomous code agent operating inside a Java repository.

You are running an executable bug-validation task for an already-injected production bug.
Your job is not to fix code.
Your job is not to improve the repository.
Your job is not to write a generally reasonable test.
Your only goal is to create a new Java test case that exposes the already-injected bug under the
     provided Maven verification command.

EVALUATION ATTEMPT BUDGET:
For each benchmark case, the evaluation protocol allows at most three bug-validation attempts.
Each attempt must obey the same test-only constraints and use the provided replay command as the
     validation target.

HARD CONSTRAINTS:
1. Never modify production code under src/main/java or any non-test source root.
2. Never modify build files, pom.xml files, Gradle files, CI files, config files, or dependency files.
3. Only create or edit Java test files under src/test/java.
4. Prefer rewriting the official testcase file at the same path and with the same package and class
     name when that information is provided.
5. The testcase must materialize in the repository as a real .java file before verification.
6. The testcase must be designed to trigger the injected bug, not to validate normal behavior.
7. Do not output file blocks for an external parser. Directly write files into the repository
     workspace.
8. Keep the testcase minimal but bug-targeted.
9. Do not delete or modify unrelated files.

TASK PAYLOAD:
Generate a new Java testcase that triggers the already-injected bug.

Success criteria:
- A real Java testcase file exists in the repository under src/test/java.
- The testcase uses the correct package declaration.
- The testcase compiles.
- The provided verification command fails because the injected bug is triggered.
- The failure is a semantic test failure, not a compile error and not a harness failure.
- No production file or build configuration file is modified.

Repository: dubbo
Context setting: WithEC
Implementation group: B
Injected case run_id: inject__20260315T064448Z
Type ID: logic_predicate_inversion_in_guard
Bug type: Guard Predicate Inversion
Applied transforms: t_rare_profile_gate_v1, t_call_stack_deepen_v1, t_private_adapter_layer_v1
Bug-related production files:
- dubbo-common/src/main/java/org/apache/dubbo/common/url/component/URLParam.java
Official test path: dubbo-common/src/test/java/org/apache/dubbo/common/url/URLParamTest.java
Official test class: org.apache.dubbo.common.url.URLParamTest
Verification command: ./mvnw -q -pl dubbo-common -Dtest=org.apache.dubbo.common.url.URLParamTest test
Injection-stage verify rc: 1

Type summary:
A boolean guard condition in production code was inverted, causing guarded URL-parameter lookup logic
     to execute under the wrong condition and leading to semantically incorrect behavior without
     changing public APIs.

Execution context block:
The target test reaches URLParam construction and method-specific parameter lookup.
Trace-relevant production methods include URLParam.hasMethodParameter, URLParam.getMethodParameter,
     URLParam.getMethodParameterStrict, URLParam.initMethodParameters, URLParam.size, and URLParam.
     isEmpty.
This execution context block identifies the path used during benchmark construction; it does not
     include the reference-trigger implementation or hidden validation oracle.

What you must do:
- Read the bug context carefully.
- Inspect the relevant production and test files.
- Rewrite or create the testcase directly at the official test path.
- Keep the package name and class name correct.
- Exercise the suspected buggy path through URLParam method-specific parameter lookup.
```

```
- Write an oracle that fails under the injected bug.
- Run the provided verification command if needed.
- Leave no production-code or build-configuration changes.

Required final output:
At the very end, print exactly one single-line JSON object prefixed by:
TRIGGER_REPORT_JSON:

The JSON object must have this schema:
{
  "summary": "...",
  "suspected_bug_mechanism": "...",
  "inspected_production_files": ["..."],
  "inspected_test_files": ["..."],
  "selected_test_path": "...",
  "selected_test_class": "...",
  "trigger_strategy": ["...", "...", "..."],
  "expected_failure_signal": "...",
  "materialized_test_files": ["..."],
  "notes": "..."
}
```

Bug Validation Report

```
TRIGGER_REPORT_JSON:
{
  "summary": "Generate a URLParam test that exercises method-specific parameter lookup after the
     transformed guard inversion.",
  "suspected_bug_mechanism": "A guard inversion and transformed lookup path cause method-specific
     parameter retrieval to return an incorrect value or miss a valid method entry.",
  "inspected_production_files": [
    "dubbo-common/src/main/java/org/apache/dubbo/common/url/component/URLParam.java"
  ],
  "inspected_test_files": [
    "dubbo-common/src/test/java/org/apache/dubbo/common/url/URLParamTest.java"
  ],
  "selected_test_path": "dubbo-common/src/test/java/org/apache/dubbo/common/url/URLParamTest.java",
  "selected_test_class": "org.apache.dubbo.common.url.URLParamTest",
  "trigger_strategy": [
    "Construct URLParam with multiple parameters so that method-specific lookup and transformed helper
       paths are exercised.",
    "Call hasMethodParameter and getMethodParameter on methods that should be present.",
    "Assert the expected valid lookup result so that the injected guard inversion becomes observable."
  ],
  "expected_failure_signal": "Assertion failure: expected method-specific parameter to be present or
     equal to the expected value, but the transformed buggy path returns an incorrect result.",
  "materialized_test_files": [
    "dubbo-common/src/test/java/org/apache/dubbo/common/url/URLParamTest.java"
  ],
  "notes": "The test is written only under src/test/java and does not modify production code or build
     configuration."
}
```

J REPRODUCIBILITY INSTRUCTIONS

We report how to verify released BUGWITGYM cases and rerun the construction and evaluation pipeline. Paths in the anonymous release may differ from the placeholders below, but the required inputs and outputs are unchanged.

J.1 ENVIRONMENT REQUIREMENTS

BUGWITGYM requires a Unix-like shell, Python 3, a Java runtime compatible with the target projects, and Maven or Maven-wrapper support. Agent-based construction and evaluation additionally require the corresponding model-backend credentials. Because hosted agents may change over time, reproducibility centers on archived case metadata, deterministic replay validation, repository-diff audits, and final success decisions.

J.2 Verifying a Released Case

Each released case contains `case.json`, `bug.patch`, `buggy_repo/`, and an optional `execution_context.json` used for the WithEC setting. To verify one case, inspect `case.json`, enter the buggy repository snapshot, and run the recorded replay command verbatim:

```
cd <case>/buggy_repo
<recorded Maven replay command>
```

The case is valid only if the command fails because of the injected bug, rather than a missing dependency, compilation error, infrastructure failure, or unrelated test failure.

J.3 Reconstructing Benchmark Cases

Given clean repository templates, construction can be reproduced by collecting execution contexts, building an injection manifest, and running the paper-aligned construction wrapper:

```
python3 bugwitgym-trace-prep/scripts/collect_maven_execution_context.py \
  --repo-template <benchmark-root>/<repo> \
  --repo-name <repo> \
  --official-test-class <test-class> \
  --official-test-path <test-path> \
  --workspace-root <out-root>/workspaces/trace \
  --output-dir <out-root>/trace_assets/<trace-id>

python3 bugwitgym-inject/scripts/build_inject_manifest_from_traces.py \
  --traces <out-root>/trace_assets/<trace-id>/selected_traces.json \
  --commands <out-root>/trace_assets/<trace-id>/selected_commands.json \
  --output <out-root>/manifests/inject.jsonl \
  --repo-template-root <benchmark-root> \
  --type-ids <bug-type-id> \
  --transform-ids <transform-id-1> ...

python3 bugwitgym-inject/scripts/run_paper_aligned_inject_batch.py \
  --project-root <project-root> \
  --manifest <out-root>/manifests/inject.jsonl \
  --workspace-root <out-root>/workspaces/inject \
  --out-root <out-root>/runs/inject \
  --dataset-root <out-root>/dataset \
  --backend-mode <backend> \
  --model-id <model-id>
```

The wrapper restores a clean workspace before each attempt, enforces the construction attempt budget, reruns the construction-time witness after transformations, and rejects cases without a production-code diff or with test/build-file modifications.

A successful construction run can be converted to the released-case format with:

```
python3 bugwitgym-full-workflow/scripts/package_review_case.py \
  --case-summary <out-root>/runs/inject/cases/<case-key>/case_summary.json \
  --case-id <case-id> \
  --output-root <out-root>/release/cases \
  --repo-id <repo> \
  --type-category <category>
```

J.4 Running Bug-Validation Evaluation

Evaluation starts from packaged cases. The runner prepares a clean workspace, overlays the buggy production-code state, removes or rewrites the official target test, invokes the agent, and reruns the recorded replay command. First convert released cases to an evaluation manifest:

```
python3 bugwitgym-trigger/scripts/release_to_trigger_manifest.py \
  --release-root <out-root>/release \
```

```
  --repo-template-root <benchmark-root> \
  --output-jsonl <out-root>/manifests/trigger.jsonl
```

Then run the two context settings:

```
python3 bugwitgym-trigger/scripts/run_paper_aligned_trigger_batch.py \
  --project-root <project-root> \
  --manifest <out-root>/manifests/trigger.jsonl \
  --out-root <out-root>/runs/trigger_<agent> \
  --groups A,B \
  --backend-mode <backend> \
  --agent-runner-script <runner-script> \
  --model-id <model-id>
```

Group `A` corresponds to NoEC, and group `B` corresponds to WithEC. The only intended difference is whether the archived execution-context summary is included. Neither setting exposes the construction-time witness, hidden validation oracle, construction-time solution, or permission to modify production code.

The runner records prompts, agent outputs, generated tests, replay logs, diff audits, failure classifications, and success decisions. A validation attempt succeeds only when it materializes a Java test under the allowed test root, the replay command fails with a semantic test failure exposing the injected bug, and no production or build files are modified.