AGENTBUG-SMITH: AUTOMATICALLY REPRODUCING REAL-WORLD HARNESS BUGS IN AGENTIC SYSTEMS

Anonymous authors
Paper under double-blind review

ABSTRACT

Agent harness bugs exhibit unique characteristics and remain challenging for state-of-the-art software agents to repair. Progress in this area is further hindered by existing benchmarks, which contain only a small and fixed number of executable harness bugs while requiring hundreds of human hours to construct. This work presents AGENTBUG-SMITH, an automated harness bug reproduction approach that continuously discovers and reproduces real-world harness bugs from open-source agentic systems. Across different backbone LLMs, AGENTBUG-SMITH consistently outperforms existing bug reproduction techniques designed for general software, achieving 10.67% - 27.56% higher success rates of reproducing harness bugs. By applying AGENTBUG-SMITH to open-source agentic systems in the wild, we construct LIVE-HARNESS-BENCH, a live and extensible benchmark that currently contains 200 reproducible harness bugs. We further demonstrate the utility of LIVE-HARNESS-BENCH through two downstream applications. First, we use LIVE-HARNESS-BENCH as the evaluation benchmark to systematically evaluate state-of-the-art software agents, revealing their limited capabilities in repairing real-world harness bugs. Second, we use LIVE-HARNESS-BENCH as a knowledge base of real-world harness bug fixes, from which reusable repair skills can be distilled to improve existing software agents, increasing their harness-bug repair rates by 6.32%. Together, AGENTBUG-SMITH and LIVE-HARNESS-BENCH establish a scalable foundation for continuously evaluating and improving software agents on harness bug repair, turning real-world agent failures into executable evaluation instances and reusable knowledge for harness improvement, thus contributing to the ultimate goal of recursively self-improving agents.

AGENTBUG-SMITH LIVE-HARNESS-BENCH

1 INTRODUCTION

Large Language Model (LLM) agents are rapidly emerging as a new software paradigm and have been increasingly adopted across diverse domains (Yang et al., 2024; Kannan et al., 2024; Zeng et al., 2023; Gottweis et al., 2026; Xiao et al., 2025). Agentic systems are commonly composed of backbone LLMs and their surrounding *harnesses*, where the harness serves as a software infrastructure that works with LLMs to jointly shape the agent behavior. Modern agents involve increasingly complex and large-scale harness implementations (e.g., 200K lines of code in OpenClaw (ope, 2026a)) to support diverse functionalities such as context management, security guardrails, orchestration, and tool integration. Accordingly, there have been growing research efforts on harness engineering (Li et al., 2026), demonstrating that improving the harness alone can substantially enhance agent effectiveness, even when the underlying backbone LLM remains unchanged. More recently, advances in self-evolving harnesses (Lee et al., 2026) and recursive self-improvement (Liu et al., 2026) have further elevated the harness from a static supporting layer to an evolving component of agentic systems, underscoring the critical role of harness quality in determining the agent capabilities.

The growing complexity of agent harnesses inevitably makes *harness implementation bugs* an important source of failures in agentic systems. Recent studies (Rahardja et al., 2025; Zhang et al., 2026b; Chen et al., 2026) have revealed that modern agent applications and frameworks suffer from diverse implementation bugs in harnesses. Automatically detecting and repairing such bugs is therefore becoming increasingly critical for building reliable agentic systems. However, prior

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.

work (Rahardja et al., 2025) has shown that existing software agents (Yang et al., 2024; Xia et al., 2024; Zhang et al., 2024) achieve substantially lower success rates in repairing harness bugs than general software bugs (e.g., 4.67% versus 40.67%). This difficulty stems in part from the distinct characteristics of harness bugs, which arise from agent-specific structures and behaviors, such as complex workflow orchestration and interactions with external resources (e.g., tools, services, and model providers). These characteristics introduce failure modes and repair challenges that are uncommon in traditional software systems. Taken together, automated harness bug repair represents an emerging and distinct challenge in improving agent reliability.

However, we still lack comprehensive benchmarks of harness bugs. Most benchmarks of agent failures (Cemri et al., 2025; Zhu et al., 2025) focus on failures stemming from backbone LLMs rather than harness implementation bugs. While there have been increasing effort to investigate harness implementation bugs (Zhang et al., 2026b; Chen et al., 2026), existing bug collections lack executable environments and artifacts for reproducing the agent failures. To date, the only executable benchmark of reproducible harness bugs, AgentIssue-Bench (Rahardja et al., 2025), contains only a small and fixed set of 43 harness bugs, which cannot support comprehensive or rigorous evaluation due to the limited number of bugs and the potential risk of contamination from exposure to model training data. In addition, manually reproducing harness bugs is labor-intensive and time-consuming, e.g., 150 manual hours were spent in reproducing a small number of harness bugs (Rahardja et al., 2025).

While recent approaches such as SWE-Factory (Guo et al., 2025) and SWE-bench-Live (Zhang et al., 2026a) have automated the reproduction of bugs reported in GitHub Issues, they are primarily designed for general software systems. Harness bugs in agentic systems pose unique reproduction challenges, as agent harnesses extensively interact with external resources, including LLM providers, tools, network services, and dynamically changing environments, while their failures often manifest only under specific execution states and interaction contexts. These characteristics fundamentally distinguish harness bugs from conventional software bugs and pose substantial challenges to directly applying existing automated bug reproduction techniques to agentic systems.

Technique. This work proposes AGENTBUG-SMITH, an automated harness bug reproduction approach that continuously discovers and reproduces real-world harness bugs from open-source agents. AGENTBUG-SMITH is built as a multi-agent system to streamline reproduction pipeline without any manual intervention, facilitating three phases including (i) harness bug related issues identification, (ii) agent execution environment construction, and (iii) failure-reproducing harness test generation.

Evaluation. We perform extensive evaluation to show the effectiveness of AGENTBUG-SMITH. First, based on our manual checking, AGENTBUG-SMITH achieves a high accuracy of automatically identifying agent repositories (i.e., 95% accuracy) and harness bug related issues (i.e., 92%). Second, AGENTBUG-SMITH consistently achieves higher success rate of reproducing harness bugs compared state-of-the-art bug reproduction techniques that are designed for general software (i.e., SWE-Factory and SWE-bench-Live), with 10.67% - 27.56% percentage-point improvements across all studied backbone LLMs. Third, our ablation analysis further confirms the effectiveness of both environment and test generation components in AGENTBUG-SMITH compared to baselines.

Benchmark and Downstream Application. Continuously applying AGENTBUG-SMITH to open-source agentic systems in the wild can enabling scalable construction of a benchmark of executable harness bugs, LIVE-HARNESS-BENCH. Its current release contains 200 reproducible harness bugs collected from real-world agent repositories, covering diverse harness components and spanning over wide time ranges. We further demonstrate the usage of LIVE-HARNESS-BENCH in two downstream applications. First, we use LIVE-HARNESS-BENCH as the evaluation benchmark to systematically evaluate state-of-the-art software agents, revealing their limited capabilities in repairing real-world harness bugs. Second, we use LIVE-HARNESS-BENCH as a knowledge base of real-world harness bug fixes, from which reusable repair skills can be distilled to improve existing software agents, increasing their harness-bug repair success rate by 6.32%. Together, AGENTBUG-SMITH and LIVE-HARNESS-BENCH establish a scalable foundation for continuously evaluating and improving software agents on harness bug repair, turning real-world agent failures into executable evaluation instances and reusable knowledge for harness improvement, thereby contributing to the ultimate goal of recursively self-improving agents.

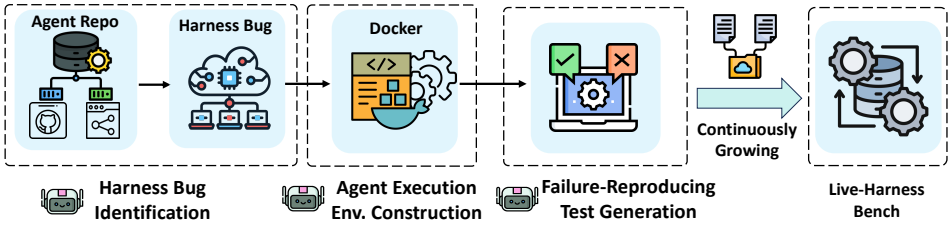

Figure 1: Overview of AGENTBUG-SMITH.

2 RELATED WORK

Failures in Agentic Systems. Substantial research efforts have been devoted to analyzing and understanding failures in agentic systems. Most existing studies or benchmarks (Zhang et al., 2025; Zhu et al., 2025; Cemri et al., 2025; Bouzenia & Pradel, 2025) focus on the runtime failures stemming from the underlying LLMs but not the failures caused by implementation bugs in the agent harness layer. While there have been recent effort in investigating the bugs in agent harnesses (Zhang et al., 2026b; Chen et al., 2026; Zhu et al., 2026; Rahardja et al., 2025; Shao et al., 2024; Xue et al., 2025), they do not provide executable environments or artifacts to trigger the failure. To date, AgentIssue-Bench is the only benchmark of executable harness bugs, but only includes a small and fixed number of bugs (i.e., 43) and is constructed with huge manual effort (i.e., 150 hours). To date, we still lack large-scale and comprehensive benchmarks of reproducible agent harness bugs.

Automated Bug Benchmark Construction. Existing techniques have automated the construction of software bug benchmarks and datasets (Yang et al., 2026; Jain et al., 2025), but primarily focus on synthesizing large numbers of artificial bugs through mutation rather than reproducing real-world bugs. Such datasets are often used as training data for model improvement. In this work, we instead focus on constructing benchmarks of real-world bugs to enable more realistic bug distribution in practice. While there has been prior work (Zhang et al., 2026a; Guo et al., 2025; Badertdinov et al., 2026; Tomassi et al., 2019; Pan et al., 2024) that automatically construct benchmarks from real-world bugs in open-source software (e.g., mostly Github issues), they are designed for general software. In contrast, this work specifically focuses on reproducing real-world bugs in agentic systems, which pose unique reproduction challenges due to the extensive interactions between agent harnesses and external resources.

3 AGENTBUG-SMITH

In this section, we present AGENTBUG-SMITH, a fully automated approach that reproduces real-world harness bugs from open-source agentic systems and thus can continuously construct a growing benchmark. Following widely-adopted construction pipelines for general-software bug benchmarks (swe, 2026; Guo et al., 2025; Zhang et al., 2026a)), we use GitHub issues as the source of real-world agent failures. Figure 1 presents the overview of AGENTBUG-SMITH, which is built as multi-agent systems covering the following three key stages. (i) *Harness Bug Identification*, an agentic pipeline that discovers high-quality GitHub repositories of agentic systems and mines issues related to harness bugs (Section 3.1); (ii) *Agent Execution Environment Construction*, an agentic pipeline that builds environment containers required for agent execution and bug reproduction (Section 3.2); and (iii) *Failure-Reproducing Harness Test Generation*, an agentic pipeline that synthesizes the harness test that can exactly trigger the issue-described agent failure (Section 3.3).

3.1 HARNESS BUG IDENTIFICATION

As the open-source agent ecosystem undergoes rapid and continuous development, static benchmarks can quickly become outdated. To capture the dynamics of this highly active community in real time, we design an automated harness bug identification pipeline to continuously discover emerging open-source agent repositories and identify GitHub issues related to agent harness bugs. Figure 2 illustrates the overall process, which consists of the following two key steps.

Agent Repository Collection. To maintain an up-to-date list of high-quality and widely used agent repositories, we incorporate a dual-stream repository discovery mechanism, including: (1) *Curated*

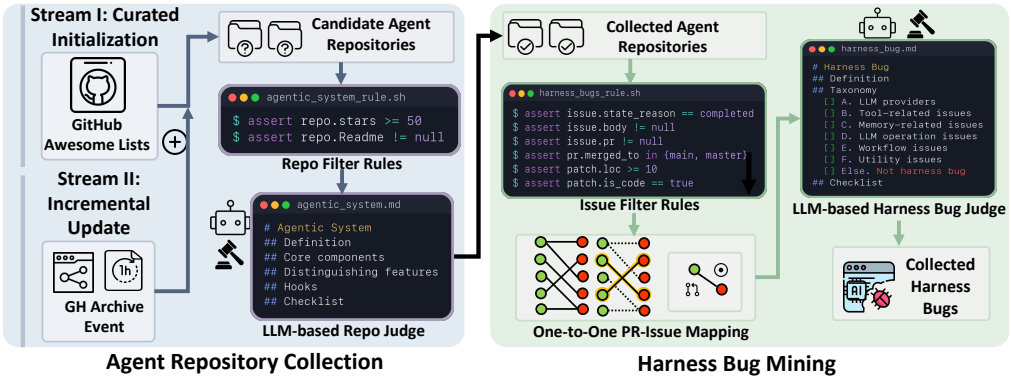

Figure 2: Harness Bug Identification in AGENTBUG-SMITH.

Initialization, which performs a one-time, large-batch initialization of the repository pool based on community-curated awesome lists of GitHub agentic systems (kyr, 2026; jen, 2026; sla, 2026; e2b, 2026; jim, 2026; roh, 2026; geo, 2026; aih, 2026; fou, 2026; shu, 2026; hyp, 2026; wan, 2026; ten, 2026; xlite dev, 2024); (2) *Incremental Stream Update*, which continuously processes near-real-time GitHub Archive event streams in small batches to discover active and emerging agent repositories based on developer interactions. The repositories identified through these two streams are merged into a unified candidate pool, allowing the initial curated collection to be continuously expanded with newly emerging projects.

To ensure the quality of selected repositories, we apply two inclusion criteria. First, repositories must demonstrate sufficient community adoption (i.e., at least 50 GitHub stars) and contain tests, providing basic support for subsequent environment construction and reproduction test generation. Detailed filtering rules are in Appendix A. Second, repositories must follow commonly adopted agent structures, involving components such as tools, orchestration, and provider integrations. Specifically, following common agent and harness components defined in prior work (Guo et al., 2026; Li et al., 2026; Liu et al., 2024; Meng et al., 2026; Wang et al., 2024a), we design an LLM-based judge to analyze each candidate repository and check whether it implements an agentic system. The detailed specification provided to the LLM judge is included in Appendix B.

Harness Bug Mining. AGENTBUG-SMITH then mines GitHub issues related to harness bugs from each agent repository. First, to ensure the quality of selected issues, we follow a set of rules (detailed in Appendix C) requiring each issue to form a valid *issue-pull request (PR) pair*, where the associated PR has been successfully merged and contains functional code changes. Moreover, as issues, PRs, and commits often exhibit many-to-many relationships, we further perform one-to-one matching (detailed in Appendix E) to identify the unique buggy and patched commits for each issue.

Second, as revealed in previous work (Rahardja et al., 2025), not all GitHub issues in agent repositories are related to harness implementation bugs. In fact, a non-trivial proportion of GitHub issues involve common bugs (e.g., utility bugs) that can also occur in general software systems. Therefore, to ensure that our constructed benchmark focuses specifically on harness bugs rather than being mixed with general software bugs, we leverage LLMs to inspect each issue and its patch, retaining an issue only if its patch occurs in a harness component. In particular, we follow the common definition of agent harness components (Guo et al., 2026; Li et al., 2026; Meng et al., 2026; Ning et al., 2026), e.g., orchestration, memory, or tools, and design a harness bug specification for the LLM judge (detailed prompt in Appendix D).

After this stage, AGENTBUG-SMITH returns a set of candidate harness bugs, whose metadata include the issue description text, the linked PR, the buggy commit, and the patched commit.

3.2 AGENT EXECUTION ENVIRONMENT CONSTRUCTION

Compared to general software, agentic systems often depend heavily on external resources (e.g., model providers, tools, and external knowledge bases), making the setup of a proper execution environment a critical step in reproducing user-reported harness bugs. While existing techniques (Guo et al., 2025; Zhang et al., 2026a; Hu et al., 2026) can generate execution environments for gen-

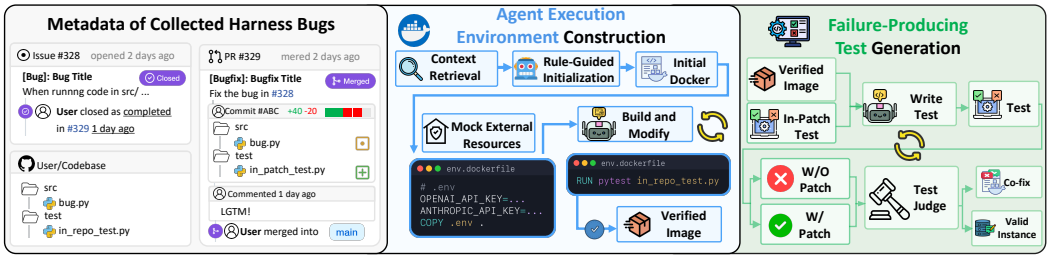

Figure 3: Environment Construction and Failure-Reproducing Test Generation.

eral software, AGENTBUG-SMITH introduces an environment generation pipeline that automatically synthesizes a Dockerfile ($\mathcal{D}$) for constructing an isolated execution environment specifically tailored to agent-dependent execution resources. In particular, to enforce strict isolation during evaluation, AGENTBUG-SMITH injects mock service environment variables at the container level, transparently rerouting external model client invocations to localized test harnesses without requiring intrusive modifications to the underlying codebase. As illustrated in Figure 3, AGENTBUG-SMITH adopts the following steps for environment construction.

Rule-Guided Initialization. Rather than relying on LLMs to generate Dockerfiles from scratch, AGENTBUG-SMITH first performs a rule-based scanning routine that parses standard package descriptors, including dependency manifests, build configurations, and CI/CD workflows, to automatically infer runtime configurations (e.g., programming language, version constraints, package manager, installation commands, and test runner). Based on the collected context, AGENTBUG-SMITH generates an initial Dockerfile $\mathcal{D}_{init}$ from an official minimal image with a single LLM API invocation, thereby incurring only minimal LLM inference cost.

Iterative Environment Refinement. The initial Dockerfile $\mathcal{D}_{init}$ can be imperfect (e.g., omitting a system library, specifying an incompatible package version, or placing the repository on an incorrect import path). AGENTBUG-SMITH therefore incorporates a tool-using agent to iteratively refine $\mathcal{D}$ based on build error messages. If the build succeeds, AGENTBUG-SMITH further samples a set of existing tests (e.g., 20) and executes them as an environment smoke test. We intentionally use a lightweight sample rather than the full test suite, as the goal at this stage is to validate basic environment executability rather than repository correctness. Running the full test suite would not only incur substantial overhead during iterative refinement, but could also introduce false-positive environment failures from buggy, flaky, or external-resource-dependent tests. Accordingly, assertion failures are excluded from the diagnostic feedback, allowing the agent to focus on environment-related failures such as collection, import, and dependency errors. The refinement loop terminates when the environment is successfully constructed or the maximum number of attempts is reached. In particular, the environment is considered successfully constructed once the image builds successfully and the tests can pass the execution.

3.3 FAILURE-REPRODUCING HARNESS TEST GENERATION

A failure-reproducing test is a test that fails on the buggy commit with the same failure reported in the corresponding issue description, while passing on the patched commit. In practice, some bug-fixing PRs contain both the patch and an *in-patch test* (IPT), where the IPT is submitted alongside the patch to validate its correctness and can therefore naturally serve as a failure-reproducing test. However, IPTs are not always available: according to our statistics, less than half (47.11%) of the issues contain IPTs. Restricting reproduction only to issues with IPTs would therefore substantially limit the scope of reproducible bugs and bias the resulting benchmark toward harness bugs for which developers explicitly provided tests. To broaden the coverage of reproducible harness bugs to issues both with and without IPTs, AGENTBUG-SMITH introduces an agentic workflow that automatically synthesizes failure-reproducing tests for a given harness bug, even when an IPT is absent.

The LLM is provided with the corresponding context (e.g., the buggy code repository, issue description, and developer-submitted patch) to generate an initial test $\mathcal{T}$ that triggers the failure described in the issue. To ensure faithful reproduction, the generated test must import and exercise the actual buggy implementation rather than replacing the target functionality with mocks. Mocking is permitted only at model-client or network boundaries to isolate test execution from nondeterministic

external networks and third-party model APIs, which is consistent with common testing practices in agentic systems (Hasan et al., 2026). In contrast, functions under test cannot be mocked, ensuring that the reproduced failure originates from the actual buggy implementation.

Joint Optimization of Test and Environment. A generated test is considered failure-reproducing only if (i) it fails on the buggy commit with the same failure reported in the issue description and (ii) it passes on the patched commit. If either condition is not satisfied, AGENTBUG-SMITH iteratively refines the test $\mathcal{T}$ based on its execution outcomes until both conditions are satisfied or the maximum number of iterations is reached. However, optimizing the test $\mathcal{T}$ alone can sometimes be insufficient, as unsuccessful reproduction may stem from misalignment between the generated test $\mathcal{T}$ and the environment Dockerfile $\mathcal{D}$. For example, a generated test may exercise an incorrect assertion while the constructed environment simultaneously imports an installed package instead of the checked-out source. To address such cases, beyond test-only optimization, AGENTBUG-SMITH optionally performs joint optimization to co-refine the test $\mathcal{T}$ and the environment Dockerfile $\mathcal{D}$. Appendix F presents an example where such joint optimization successfully brings the test into a fail-to-pass state (i.e., failing on the buggy commit while passing on the patched commit), whereas test-only optimization cannot.

4 EVALUATION

4.1 EFFECTIVENESS OF HARNESS BUG IDENTIFICATION

This section first evaluates the effectiveness of the first component (i.e., harness bug identification) in AGENTBUG-SMITH, as it serves as the foundation for all the subsequent components.

Settings. For continuous harness bug identification, we primarily use cost-effective models (i.e., GPT-4.1-mini) in AGENTBUG-SMITH, as this stage continuously processes a large volume of issues and thus requires cost-effective model inference for scalable deployment. As there is no existing technique specifically designed for harness bug identification, we compare our approach against vanilla LLM invocation as the baseline. To evaluate identification accuracy, we randomly sample 132 out of 225 GitHub issues (corresponding to a 95% confidence level and a 5% margin of error) identified as harness-bug-related by our approach and the vanilla baseline. Two annotators then independently label whether each selected repository is an agent repository and whether each selected issue is related to a harness bug. The resulting Cohen's kappa is 0.88, indicating high inter-annotator agreement (Landis & Koch, 1977). In addition, given the inherent randomness of LLM inference, we evaluate the stability of our approach by independently classifying each identified issue five times and reporting the set agreement S_{set} and normalized entropy S_{entropy} (formulas in Appendix H).

Table 1: Accuracy and Stability of Harness Bug Identification.

Method	Stability		Accuracy	
	s_{set}	s_{entropy}	Repo. Acc.	Issue Acc.
Vanilla LLM	0.72	0.89	0.35	0.46
AGENTBUG-SMITH	0.92	0.94	0.95	0.92

Results. As shown in Table 1, AGENTBUG-SMITH achieves high accuracy in identifying both agent repositories (i.e., 95% repository-level accuracy) and harness bug related issues (i.e., 92% issue-level accuracy). Moreover, our approach demonstrates higher stability across repeated executions than vanilla LLM invocation, further supporting the reliability of the issues identified by AGENTBUG-SMITH. Overall, these results demonstrate that AGENTBUG-SMITH can accurately and consistently identify harness bug related issues in practice.

4.2 REPRODUCTION SUCCESS RATE

This section evaluates the bug reproduction success rate of AGENTBUG-SMITH and compares it with state-of-the-art bug reproduction techniques that are designed for general software systems.

Baselines and Settings. We include two state-of-the-art bug reproduction techniques, SWE-Factory (Guo et al., 2025) and SWE-bench-Live (Zhang et al., 2026a), as our baselines and directly adopt their released implementations. Following previous work (Guo et al., 2025), we evaluate all studied techniques using three different and widely used backbone LLMs,

including GPT-4.1-mini (GPT, 2025), Kimi-k2.5 (kim, 2026), and DeepSeek-v3.2 (dee, 2025), considering the cost-effectiveness required for continuous application to a large volume of GitHub issues. Since SWE-Factory and SWE-bench-Live are designed to reproduce general software bugs from GitHub issues, they do not include components for identifying harness bug related issues. Therefore, for a fair comparison, we apply all studied approaches to the same set of 225 harness bug-related issues identified by the first component of AGENTBUG-SMITH in the previous section.

Table 2: Reproduction Success Rate.

Method	Success rate	Avg. $
GPT-4.1-mini		
SWE-Factory	21/225 (9.33%)	0.09
SWE-bench-Live	6/225 (2.67%)	0.41
AGENTBUG-SMITH	45/225 (20.00%)	0.56
Kimi-k2.5		
SWE-Factory	19/225 (8.44%)	0.37
SWE-bench-Live	6/225 (2.67%)	0.66
AGENTBUG-SMITH	46/225 (20.44%)	0.91
DeepSeek-v3.2		
SWE-Factory	31/225 (13.78%)	0.31
SWE-bench-Live	1/225 (0.44%)	1.05
AGENTBUG-SMITH	63/225 (28.00%)	2.44

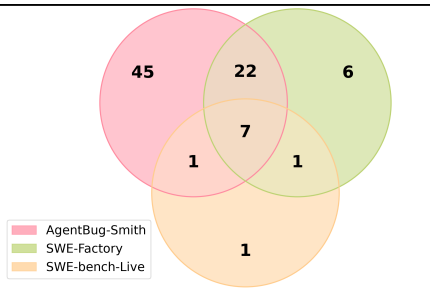

Figure 4: Overlapping Analysis.

Results. As shown in Table 2, despite the inherent difficulty of reproducing real-world harness bugs, AGENTBUG-SMITH successfully reproduces up to 28.00% of the harness bugs, substantially outperforming existing baselines (e.g., 10.67%-27.56% percentage-point improvements) at acceptably higher costs (e.g., 0.56-2.44 per issue). This result is particularly encouraging given that the best baseline can only achieve up to 13.78% reproduction success rate. Notably, such improvements are consistent across all backbone LLMs, demonstrating the generality of AGENTBUG-SMITH. We further analyze the overlap among the issues reproduced by all techniques (across all backbone LLMs) in Figure 4. Overall, AGENTBUG-SMITH successfully reproduces 75 bugs, including 45 unique bugs that cannot be reproduced by either baseline. These results further demonstrate the unique effectiveness of AGENTBUG-SMITH and highlight the necessity of developing agent-oriented bug reproduction techniques. Furthermore, to validate automatically generated tests truly reproduce the reported failure rather than merely satisfying fail-to-pass, we also perform manual inspection of all the 75 successfully-reproduced bugs, showing 74 (98.67%) of them are valid. Nevertheless, to better understand the limitations of AGENTBUG-SMITH, Appendix G further performs failure analysis of unsuccessful reproduction.

4.3 ABLATION ANALYSIS

This section performs the ablation analysis to investigate the effectiveness of the remaining two components (e.g., environment construction and test generation) in AGENTBUG-SMITH.

Effectiveness of Environment Construction. In Table 3, the column "Env. Build" compares the effectiveness of environment construction in all studied techniques. In particular, AGENTBUG-SMITH substantially outperforms both baselines across all backbone models, i.e., 10.67% - 35.11% percentage-point improvements in success rate of environment construction. These results suggest that effectively constructing execution environments for agentic systems requires explicitly accounting for their agent-specific dependencies and execution characteristics.

Effectiveness of Test Generation. In Table 3, the "#Test Gen." column reports the overall number of issues that can be successfully reproduced with failure-reproducing tests, while the "w/ IPT" and "w/o IPT" columns report the numbers of successfully reproduced issues that originally come with and without in-patch tests, respectively. Notably, AGENTBUG-SMITH is capable of generating failure-reproducing tests for issues without any in-patch tests, whereas both baselines fail to reproduce any such issues when no in-patch tests are available as references. This gap demonstrates the effectiveness of the failure-reproducing test generation component, which enables AGENTBUG-SMITH to reproduce a broader scope of harness bugs.

4.4 LIVE-HARNESS-BENCH: STATISTICS AND DOWNSTREAM APPLICATION

The evaluation above demonstrates the effectiveness of AGENTBUG-SMITH in reproducing agent harness bugs. Building on this capability, we further apply AGENTBUG-SMITH to construct LIVE-

Table 3: Effectiveness of Environment Construction and Test Generation.

Method	Env. Build	#Test Gen.	w/ IPT	wo/ IPT
GPT-4.1-mini				
SWE-Factory	36/225 (16.00%)	21	21	0
SWE-bench-Live	33/225 (14.67%)	6	6	0
AGENTBUG-SMITH	84/225 (37.33%)	45	24	21
Kimi-k2.5				
SWE-Factory	27/225 (12.00%)	19	19	0
SWE-bench-Live	29/225 (12.89%)	6	6	0
AGENTBUG-SMITH	53/225 (23.56%)	46	20	26
DeepSeek-v3.2				
SWE-Factory	44/225 (19.56%)	31	31	0
SWE-bench-Live	14/225 (6.22%)	1	1	0
AGENTBUG-SMITH	93/225 (41.33%)	63	34	29

HARNESS-BENCH, a live, large-scale benchmark of executable harness bugs. The current release of LIVE-HARNESS-BENCH contains 200 executable harness bugs and can be continuously and automatically extended with newly emerging harness bugs through AGENTBUG-SMITH.

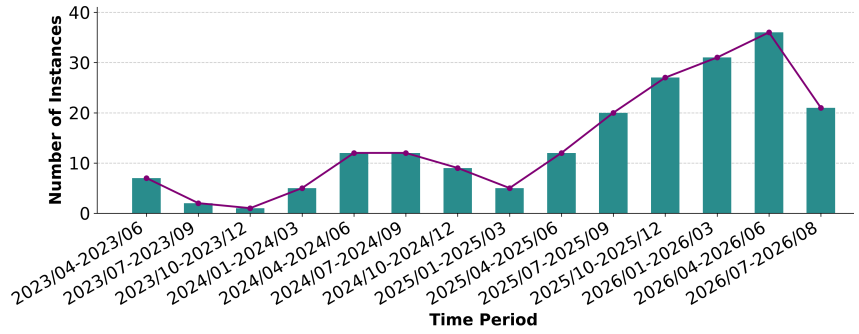

Figure 6: Temporal Distribution of Issue Creation Times in LIVE-HARNESS-BENCH.

Statistics of LIVE-HARNESS-BENCH. Table 4 summarizes the complexity of the harness bugs in LIVE-HARNESS-BENCH in terms of the scale of the corresponding code repositories and the developer-submitted gold patches. Overall, the reproduced bugs involve large codebase and non-trivial code changes, highlighting the substantial effort required to navigate the repositories, localize the faults, and implement the corresponding fixes. Figure 5 further shows the distribution of bugs in LIVE-HARNESS-BENCH across different harness components. In particular, LIVE-HARNESS-BENCH provides diverse coverage of all critical agent harness components, for example, with 31.5% of the bugs involving tool registries and action interfaces and 30.5% involving context and memory management. Moreover, Figure 6 presents the temporal distribution of the reproduced bugs, demonstrating consistent coverage across time. Enabled by AGENTBUG-SMITH, LIVE-HARNESS-BENCH can also incorporate recent harness bugs, including those reported as recently as August 2026. Going forward, periodically applying AGENTBUG-SMITH to continuously extend LIVE-HARNESS-BENCH with newly reported bugs can help mitigate concerns about contamination from model training data.

Table 4: Benchmark Statistics.

Level	#Item	Average	Median
Repo.	LoC	123k	83k
	Files	479	343
Patch	Files	3.8	3.0
	Hunks	19.8	12.0
	Lines	202.9	102.5

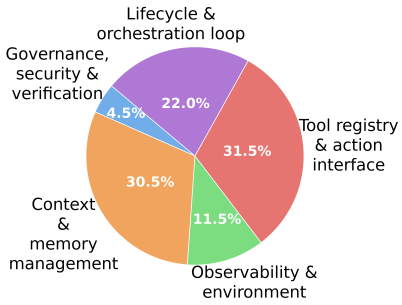

Figure 5: Bug Distribution across Harness Components.

Downstream Application I: Evaluation Benchmark. One straightforward application of LIVE-HARNESS-BENCH is to serve as a benchmark for systematically evaluating the capabilities of

existing software agents in fixing harness bugs. Table 5 presents the performance of three widely used software agents, mini-SWE-agent (min, 2026), OpenHands (ope, 2026b), and AutoCodeRover (Zhang et al., 2024), with GPT-4.1-mini on LIVE-HARNESS-BENCH. The "plausibly resolved" column refers to cases where the generated patch passes the failure-reproducing tests. Following the previous work (Rahardja et al., 2025) and the common practice in program repair (Xia & Zhang, 2024), we further manually inspect whether the plausible patches are semantically equivalent to the developer-submitted gold patch, which is presented in the "correctly resolved" column. Overall, state-of-the-art software agents exhibit limited resolution rates on LIVE-HARNESS-BENCH (e.g., at most 9.00%), especially compared with their resolution rates reported on general software bugs, e.g., 40.67% on SWE-Bench Verified (swe, 2025). These results highlight the challenges of fixing real-world harness bugs and the importance of constructing reproducible harness bug benchmarks for rigorous evaluation. Detailed analysis of the resolution rates across harness components is in Appendix J.

Table 5: Effectiveness of Software Agents on LIVE-HARNESS-BENCH.

Agent	Plausibly resolved%	Correctly resolved%	Localization %		Avg. $Cost
			File-level	Function-level	
mini-SWE-agent	19.50	9.00	75.12	68.66	0.38
OpenHands	17.50	8.50	81.00	66.00	0.03
AutoCodeRover	6.00	3.50	37.19	31.66	0.04

Downstream Application II: Enhancement. Given the limited capabilities of existing software agents in fixing harness bugs (as shown in Application I), another application of LIVE-HARNESS-BENCH is to serve as a knowledge base for enhancing existing software agents on harness bug repair. To validate this potential, we split LIVE-HARNESS-BENCH into training and test sets. To avoid overfitting and data leakage, we strictly ensure that harness bugs from the same agent repository do not appear in both the training and test sets. As a result, the training set contains 121 harness bugs, while the remaining 79 harness bugs form the test set. Specifically, we implement a basic skill distillation pipeline that iteratively generates and refines a textual skill (`SKILL.md`) over the training instances. We then apply the distilled skill to an existing software agent, mini-SWE-agent, which achieves the best effectiveness in our evaluation in Application I. The detailed skill distillation design is in Appendix I. As shown in Table 6, the distilled skill improves the correct resolution rate of the off-the-shelf software agent by 6.32%, while achieving 31.81% function-level localization accuracy. Appendix K presents an example illustrating how the distilled skill enables mini-SWE-agent to fix a harness bug that it cannot resolve without the distilled skill. Overall, given the continuously growing nature of LIVE-HARNESS-BENCH, it can serve as a scalable data foundation for continuously improving software agents on harness bug repair. More broadly, as harness improvement constitutes an important building block toward recursively self-improving agents, LIVE-HARNESS-BENCH provides a foundation for advancing this capability through continuously accumulated real-world harness failures and fixes.

Table 6: Effectiveness of Distilled Skills.

Setting	Plausibly resolved%	Correctly resolved%	Localization %	
			File-level	Function-level
W/o skill	35.44	1.27	79.32	34.60
W/ skill	41.77	7.59	92.31	66.41

5 CONCLUSION

This work presents AGENTBUG-SMITH, an automated harness bug reproduction approach that continuously discovers and reproduces real-world harness bugs from open-source agentic systems. AGENTBUG-SMITH substantially outperforms existing bug reproduction techniques designed for general software. Building on AGENTBUG-SMITH, we automatically construct LIVE-HARNESS-BENCH, a live and extensible benchmark that currently contains 200 reproducible harness bugs and can be continuously expanded with newly emerging bugs. Our two downstream application experiments further demonstrate that LIVE-HARNESS-BENCH can serve as both a rigorous evaluation benchmark and a reusable knowledge base for evaluating and improving existing software agents in fixing harness bugs, thereby contributing to the broader goal of recursively self-improving agents.

AI USE STATEMENT

In this work, our approach to harness bug reproduction integrates LLMs. LLMs are used as judges for repository and issue filtering, as well as for Dockerfile and test generation, as described in our methodology. We also used generative AI tools to polish the manuscript. We did not use generative AI tools for research ideation or execution or to draft sections of the paper. Generating synthetic datasets and proving mathematical claims are not applicable to this work. We reviewed all AI-assisted content and conducted rigorous evaluations to assess the effectiveness of our approach.

REPRODUCIBILITY STATEMENT

To support reproducibility and facilitate future research, we release our data and code in AGENTBUG-SMITH. Our benchmark is available in LIVE-HARNESS-BENCH.

REFERENCES

Gpt-4.1-mini, 2025. https://openai.com/index/gpt-4-1/.

Deepseek-v3.2, 2025. https://api-docs.deepseek.com/news/news251201/.

Swe-bench verified, 2025. https://www.swebench.com/verified.html.

Awesome chinese llm, 2026. https://github.com/AiHubCN/Awesome-Chinese-LLM.

Awesome ai agents, 2026. https://github.com/e2b-dev/awesome-ai-agents.

Awesome-foundation-agents, 2026. https://github.com/FoundationAgents/awesome-foundation-agents.

Awesome ai in finance, 2026. https://github.com/georgezouq/awesome-ai-in-finance.

Awesome llm-powered agent, 2026. https://github.com/hyp1231/awesome-llm-powered-agent.

Awesome-ai-agents, 2026. https://github.com/Jenqyang/Awesome-AI-Agents.

Awesome ai agents: Tools, resources, and projects, 2026. https://github.com/jim-schwoebel/awesome_ai_agents.

Kimi-k2.5, 2026. https://www.kimi.ai/ai-models/kimi-k2-5/.

Awesome agents, 2026. https://github.com/kyrolabs/awesome-agents.

mini-swe-agent, 2026. https://github.com/swe-agent/mini-swe-agent.

openclaw, 2026a. https://github.com/openclaw/openclaw.

openhands, 2026b. https://github.com/OpenHands/openhands.

Awesome ai apps, 2026. https://github.com/rohitg00/awesome-ai-apps.

Awesome llm apps, 2026. https://github.com/shubhamsaboo/awesome-llm-apps.

Awesome ai agents, 2026. https://github.com/slavakurilyak/awesome-ai-agents.

Swe leaderboards, 2026. https://www.swebench.com.

Awesome llmops, 2026. https://github.com/tensorchord/Awesome-LLMOps.

Awesome llm resources, 2026. https://github.com/WangRongsheng/awesome-LLM-resources.

Ibragim Badertdinov, Alexander Golubev, Maksim Nekrashevich, Anton Shevtsov, Simon Karasik, Andrei Andriushchenko, Maria Trofimova, Daria Litvintseva, and Boris Yangel. Swe-rebench: An automated pipeline for task collection and decontaminated evaluation of software engineering agents. *Advances in Neural Information Processing Systems*, 38, 2026.

Islem Bouzenia and Michael Pradel. Understanding software engineering agents: A study of thought-action-result trajectories. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 2846–2857. IEEE, 2025.

Mert Cemri, Melissa Z. Pan, Shuyi Yang, Lakshya A. Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya G. Parameswaran, Dan Klein, Kannan Ramchandran, Matei A. Zaharia, Joseph E. Gonzalez, and Ion Stoica. Why do multi-agent LLM systems fail? In *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2025, NeurIPS 2025, San Diego, CA, USA, December 2-7, 2025 / Mexico City, Mexico, November 30 - December 5, 2025*, 2025.

Jingyi Chen, Songqiang Chen, Hengcheng Zhu, Jialun Cao, Jiasi Shen, and Shing-Chi Cheung. Understanding agent-reactive bugs at the model-harness boundary: An empirical study of llm agent issue reports. *arXiv preprint arXiv:2607.15684*, 2026.

Juraj Gottweis, Wei-Hung Weng, Alexander Daryin, Tao Tu, Petar Sirkovic, Artiom Myaskovsky, Grzegorz Glowaty, Felix Weissenberger, Alessio Orlandi, Dan Popovici, Anil Palepu, Keran Rong, Ryutaro Tanno, Khaled Saab, Fan Zhang, Jacob Blum, Andrew Carroll, Kavita Kulkarni, Nenad Tomašev, Dina Zverinski, Ivor Rendulic, Elahe Vedadi, Florian Hasler, Luka Rimanic, Marina Boia, Ivan Budiselic, Ben Feinstein, Mathias Bellaiche, Tom Sheffer, Jan Freyberg, Jeremy Ratcliff, Ottavia Bertolli, Katherine Chou, Avinatan Hassidim, Burak Gokturk, Amin Vahdat, Yuan Guan, Vikram Dhillon, Eeshit Dhaval Vaishnav, Byron Lee, Tiago R. D. Costa, José R. Penadés, Gary Peltz, Yossi Matias, James Manyika, Demis Hassabis, Yunhan Xu, Pushmeet Kohli, Annalisa Pawlosky, Alan Karthikesalingam, and Vivek Natarajan. Accelerating scientific discovery with co-scientist. *Nature*, 655(8122):487–496, May 2026. ISSN 1476-4687. doi: 10.1038/s41586-026-10644-y. URL http://dx.doi.org/10.1038/s41586-026-10644-y.

Jianyuan Guo, Zhiwei Hao, Chengcheng Wang, Cheng Fan, Tingzhang Luo, Hongguang Li, Ying Gao, Hefei Mei, Jiankun Peng, Rongjian Xu, Minjing Dong, Han Im Wu, Mengyu Zheng, Kai Han, Shiqi Wang, Chang Xu, and Yunhe Wang. From question answering to task completion: A survey on agent system and harness design. *arXiv preprint arXiv:2606.20683*, 2026.

Lianghong Guo, Yanlin Wang, Caihua Li, Wei Tao, Pengyu Yang, Jiachi Chen, Haoyu Song, Duyu Tang, and Zibin Zheng. Swe-factory: Your automated factory for issue resolution training data and evaluation benchmarks. *arXiv preprint arXiv:2506.10954*, 2025.

Mohammed Mehedi Hasan, Hao Li, Emad Fallahzadeh, Gopi Krishnan Rajbahadur, Bram Adams, and Ahmed E Hassan. An empirical study of testing practices in open source ai agent frameworks and agentic applications. *Empirical Software Engineering*, 31(5):124, 2026.

Ruida Hu, Chao Peng, Junjielong Xu, and Cuiyun Gao. Repo2run: Automated building executable environment for code repository at scale. *Advances in Neural Information Processing Systems*, 38:32679–32718, 2026.

Naman Jain, Jaskirat Singh, Manish Shetty, Liang Zheng, Koushik Sen, and Ion Stoica. R2e-gym: Procedural environments and hybrid verifiers for scaling open-weights swe agents. *arXiv preprint arXiv:2504.07164*, 2025.

Shyam Sundar Kannan, Vishnunandan L. N. Venkatesh, and Byung-Cheol Min. SMART-LLM: smart multi-agent robot task planning using large language models. In *IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2024, Abu Dhabi, United Arab Emirates, October 14-18, 2024*, pp. 12140–12147. IEEE, 2024. doi: 10.1109/IROS58592.2024.10802322. URL https://doi.org/10.1109/IROS58592.2024.10802322.

J. Richard Landis and Gary G. Koch. The measurement of observer agreement for categorical data. *Biometrics*, 33(1):159–174, 1977. doi: 10.2307/2529310.

Yoonho Lee, Roshen Nair, Qizheng Zhang, Kangwook Lee, Omar Khattab, and Chelsea Finn. Meta-harness: End-to-end optimization of model harnesses. *arXiv preprint arXiv:2603.28052*, 2026.

Junjie Li, Xi Xiao, Yunbei Zhang, Chen Liu, Lin Zhao, Xiaoying Liao, Yingrui Ji, Janet Wang, Yingqiang Ge, Weijie Xu, Xi Fang, Xiang Xu, Tianchen Zhao, Youngeun Kim, Jihun Hamm, Tianyang Wang, and Chandan Reddy. Agent harness engineering: A survey, 2026. URL https://openreview.net/pdf?id=eONq7FdiHa.

Junwei Liu, Kaixin Wang, Yixuan Chen, Xin Peng, Zhenpeng Chen, Lingming Zhang, and Yiling Lou. Large language model-based agents for software engineering: A survey. *CoRR*, abs/2409.02977, 2024. doi: 10.48550/ARXIV.2409.02977. URL https://doi.org/10.48550/arXiv.2409.02977.

Shuaiqi Liu, Zhengkai Lin, Yuxiang Zhang, Yuanyi Ren, Yue Wu, Yongbin Li, Zheng Wang, Zhihang Fu, and Jieping Ye. The path to recursive self-improving agents: Foundation, framework, and future directions. 2026.

Qianyu Meng, Yanan Wang, Liyi Chen, Wei Wu, Yihang Li, Wenyuan Jiang, Qimeng Wang, Chengqiang Lu, Yan Gao, Yi Wu, and Yao Hu. Agent harness for large language model agents: A survey. 2026. doi: 10.20944/preprints202604.0428.v3. URL https://www.preprints.org/manuscript/202604.0428/v3.

Xuying Ning, Katherine Tieu, Dongqi Fu, Tianxin Wei, Zihao Li, Yuanchen Bei, Jiaru Zou, Mengting Ai, Zhining Liu, Ting-Wei Li, et al. Code as agent harness. *arXiv preprint arXiv:2605.18747*, 2026.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. Training software engineering agents and verifiers with swe-gym. *arXiv preprint arXiv:2412.21139*, 2024.

Alfin Wijaya Rahardja, Junwei Liu, Weitong Chen, Zhenpeng Chen, and Yiling Lou. Can agent fix agent issues? In D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen (eds.), *Advances in Neural Information Processing Systems*, volume 38, Main Conference, pp. 49002–49028. Curran Associates, Inc., 2025. doi: 10.52202/085713-1638. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/45f0ab2d264a057113492b35c5ef47f2-Paper-Conference.pdf.

Yuchen Shao, Yuheng Huang, Jiawei Shen, Lei Ma, Ting Su, and Chengcheng Wan. Are llms correctly integrated into software systems? *arXiv preprint arXiv:2407.05138*, 2024.

David A Tomassi, Naji Dmeiri, Yichen Wang, Antara Bhowmick, Yen-Chuan Liu, Premkumar T Devanbu, Bogdan Vasilescu, and Cindy Rubio-González. Bugswarm: Mining and continuously growing a dataset of reproducible failures and fixes. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 339–349. IEEE, 2019.

Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, Wayne Xin Zhao, Zhewei Wei, and Jirong Wen. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6), March 2024a. ISSN 2095-2236. doi: 10.1007/s11704-024-40231-1. URL http://dx.doi.org/10.1007/s11704-024-40231-1.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent workflow memory. *arXiv preprint arXiv:2409.07429*, 2024b.

Chunqiu Steven Xia and Lingming Zhang. Automated program repair via conversation: Fixing 162 out of 337 bugs for $0.42 each using chatgpt. In Maria Christakis and Michael Pradel (eds.), *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2024, Vienna, Austria, September 16-20, 2024*, pp. 819–831. ACM, 2024. doi: 10.1145/3650212.3680323. URL https://doi.org/10.1145/3650212.3680323.

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

Ying Xiao, Jie Huang, Ruijuan He, Jing Xiao, Mohammad Reza Mousavi, Yepang Liu, Kezhi Li, Zhenpeng Chen, and Jie M. Zhang. Fairmedqa: Benchmarking bias in large language models for medical question answering. *CoRR*, abs/2505.19562, 2025. doi: 10.48550/ARXIV.2505.19562. URL https://doi.org/10.48550/arXiv.2505.19562.

liyucheng09 etc xlite dev. Awesome-llm-inference: A curated list of awesome llm inference papers with codes, 2024. URL https://github.com/xlite-dev/Awesome-LLM-Inference. Open-source software available at https://github.com/xlite-dev/Awesome-LLM-Inference.

Ziluo Xue, Yanjie Zhao, Shenao Wang, Kai Chen, and Haoyu Wang. A characterization study of bugs in llm agent workflow orchestration frameworks. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 3369–3380. IEEE, 2025.

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

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

Fanlong Zeng, Wensheng Gan, Yongheng Wang, Ning Liu, and Philip S. Yu. Large language models for robotics: A survey. *CoRR*, abs/2311.07226, 2023. doi: 10.48550/ARXIV.2311.07226. URL https://doi.org/10.48550/arXiv.2311.07226.

Linghao Zhang, Shilin He, Chaoyun Zhang, Yu Kang, Bowen Li, Chengxing Xie, Junhao Wang, Maoquan Wang, Yufan Huang, Shengyu Fu, et al. Swe-bench goes live! *Advances in Neural Information Processing Systems*, 38, 2026a.

Shaokun Zhang, Ming Yin, Jieyu Zhang, Jiale Liu, Zhiguang Han, Jingyang Zhang, Beibin Li, Chi Wang, Huazheng Wang, Yiran Chen, and Qingyun Wu. Which agent causes task failures and when? on automated failure attribution of LLM multi-agent systems. In *Forty-second International Conference on Machine Learning, ICML 2025, Vancouver, BC, Canada, July 13-19, 2025*, volume 267 of *Proceedings of Machine Learning Research*. PMLR / OpenReview.net, 2025.

Xiaowen Zhang, Hannuo Zhang, and Shin Hwei Tan. Understanding bugs in modern agentic frameworks: A study of symptoms, root causes, and triggering conditions. *arXiv preprint arXiv:2604.08906*, 2026b.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. Autocoderover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 1592–1604, 2024.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. Expel: LLM agents are experiential learners. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pp. 19632–19642, 2024. doi: 10.1609/aaai.v38i17.29936.

Kunlun Zhu, Zijia Liu, Bingxuan Li, Muxin Tian, Yingxuan Yang, Jiaxun Zhang, Pengrui Han, Qipeng Xie, Fuyang Cui, Weijia Zhang, Xiaoteng Ma, Xiaodong Yu, Gowtham Ramesh, Jialian Wu, Zicheng Liu, Pan Lu, James Zou, and Jiaxuan You. Where LLM agents fail and how they can learn from failures. *CoRR*, abs/2509.25370, 2025.

Xinxue Zhu, Jiacong Wu, Xiaoyu Zhang, Tianlin Li, Yanzhou Mu, Juan Zhai, Chao Shen, Chunrong Fang, and Yang Liu. Bugs in modern llm agent frameworks: An empirical study. In *Proceedings of the 34th ACM International Conference on the Foundations of Software Engineering*, pp. 1588–1592, 2026.

A Repository Selection Rules

Dual-stream collection

Stream	Rule
Static	Query GitHub for curated awesome-lists (`awesome-agent`, `awesome-llm`, `awesome-ai-agents`, ...) and expand hits with the keyword family.
Incremental	Ingest GHArchive hourly files, extract `repo.name`, apply the same family.
Merge	Deduplicate the two streams.

Keyword family	Examples
Agent identity	agent, assistant, chatbot
LLM / providers	LLM, OpenAI, Anthropic, model names
Tools and memory	tool calling, memory, RAG, vector
Control	planner, orchestration, multi-agent
Prompting	prompt, ReAct, few-shot
Frameworks	CrewAI, AutoGen, LangChain, MetaGPT, AutoGPT

We seed from community-curated awesome-lists of agentic systems and expand them with a finite keyword family. The family can later be extended by mining frequent terms from repositories that survive verification.

Event	Counts as active
`PushEvent`	code was pushed
`PullRequestEvent`, `IssuesEvent`	a PR or issue moved
`WatchEvent`, `ForkEvent`	a star or fork
`CreateEvent`, `ReleaseEvent`	a ref or release
`MemberEvent`, `PublicEvent`, `DeleteEvent`	membership, visibility, or deletion

We ingest GHArchive hourly files. A repository is *active* if it appears in recent events above (pushes, pull requests, issues, stars, forks, releases). Activity is not equated with code change: a star and a push both indicate that the repository is currently visible, which is the signal we want for emerging systems.

Filtering gates

Gate	Keep if
Stars	≥ 50
README	parseable
In-repo tests	at least one path matches the detector below (issue-agnostic; a hard gate)

B Agentic System specification

This is the document given to the repository judge.

Definition. An *Agentic System* is a codebase that implements or contains an LLM-driven agent. Its operational logic is driven by a large language model and exhibits some of autonomous planning, memory management, environmental perception, and tool use. If a repository satisfies some of the components below, it should be considered an Agentic System.

Core components.

Component	What it covers
LLM "brain"	task decomposition, scheduling, conversational or action memory
Perception	user input, events, files, sensors, or other environment signals
Action / tooling	search, shell, databases, third-party APIs, plugins
Orchestration	single-agent loop or multi-agent collaboration
Provider integration	SDKs, API keys, or model configuration
Runtime artifacts	optional: Docker, prompts, tests, example scenarios

Distinguishing features.

Feature	Why it differs from conventional software
Nondeterminism	identical inputs may yield different outputs
External dependence	providers, tools, and resources change frequently
Cross-component failures	faults span LLM calls, memory, tools, and workflow
Prompt / context	prompt libraries and context length are first-class
Workflow orientation	loops and state checks may hang or run infinitely

Hooks. A repository is likely an Agentic System if it contains:

#	Hook	Typical evidence
1	LLM provider	`openai`, `anthropic`, or a custom wrapper
2	Prompts	`prompt/`, `templates/`, `prompts/`
3	Memory	vector DB, session store, history module
4	Tools	`tools/`, `plugins/`, `tool_wrappers/`
5	Control loop	`planner`, `scheduler`, agent loop
6	Model config	model name, token limit, API key, context length
7	Language	README mentions of agent, planner, assistant loop, tool invocation

Checklist. Three or more "yes": *likely*. Five or more: *highly likely*. The judge emits `owner/repo` and drops documentation collections, tutorials, paper lists, and standalone tool libraries.

ID	Question	If yes
AS1	README or code mentions agent, planner, tool invocation, memory, or LLM?	likely
AS2	Depends on an LLM-provider SDK?	likely
AS3	Prompt / template directory, or prompt-management code?	likely
AS4	Memory / session / vector store?	likely
AS5	Wrappers or calls to external tools or plugins?	likely
AS6	Orchestration or an agent loop (planner / scheduler)?	strongly

C Issue Selection Rules

Keep if

Criterion	Rule
Closure	`state_reason` = `completed`; drop `not_planned` and `null`
Description	non-empty body after stripping whitespace
Linked fix	at least one PR (local git if cloned, else the GitHub API)
Merge target	some linked PR is merged into `main` or `master`; drop `develop` / `release`
Patch size	added + deleted lines ≥ 20
In-patch tests	recorded with the same detector as in-repo tests, never required

Drop if the patch is exclusively

Kind	Patterns
Documentation	`docs/`, README, `*.md`, `*.rst`
Lock / generated	`package-lock.json`, `yarn.lock`, `poetry.lock`, `Pipfile.lock`, `Cargo.lock`, `go.sum`, `*.pb.go`, `dist/`, `build/`
Vendored	`vendor/`, `third_party/`, `node_modules/`
Binary / assets	images, archives, PDF, audio/video, `.onnx`, `.pt`, `.pth`, `.h5`, `.pickle`
Formatting	≥ 20 changed lines of which $< 5\%$ have non-whitespace content

Stored fields

Field	Role
`repo`, timestamps	which Agentic System, when crawled
issue meta	number, title, url, body, labels
linked PR	number, merge flag, base branch
`base_sha`	buggy snapshot (pre-merge mainline)
`head_sha`	patched snapshot (PR tip)
`patch`	unified diff; may be missing
in-repo tests	`existing_test_paths`
in-patch tests	`test_paths_in_patch`; optional
`ai_judgment`	Harness Bug decision and raw response

D HARNESS BUG SPECIFICATION

This is the document given to the issue judge.

Definition. A *Harness Bug* is a user-reported problem (bug report or feature request) in an Agentic System that concerns agent-specific execution machinery: LLM-provider integration, tool invocation, memory, LLM operation, workflows, and utilities.

Taxonomy. Six categories and twenty sub-categories. Category F is listed so that utility issues—failures that also arise in conventional software—can be excluded.

ID	Sub-category	Typical failure
A. Incompatibility with LLM providers		
A.1	Incompatible dependencies	missing or misused provider SDKs (e.g., OpenAI, LiteLLM)
A.2	Unsupported models	cannot bind popular models (GPT-4, Claude, DeepSeek, . . .)
A.3	Incompatible parameters	unexpected or missing provider arguments
B. Tool-related issues		
B.1	Tool dependencies	missing libraries or binaries needed to register or run tools
B.2	Tool configuration	retriever / embedder / retrieval-mode misconfiguration
B.3	Tool implementation	bugs in tool or RAG logic, including helpers
B.4	Misused interfaces	bad arguments, serialization, or LLM–tool binding
C. Memory-related issues		
C.1	Initialization	DB / workspace reset failures, inconsistent state
C.2	Content errors	bad message attributes, serialization, non-primitive types
C.3	Memory dependencies	broken internal or external memory-stack modules
D. LLM operation issues		
D.1	Model access	wrong binding or missing credentials
D.2	Token usage	max tokens, pricing, or token-accounting failures
D.3	Output handlers	empty, malformed, or refusal responses mishandled
D.4	Model dependencies	missing tokenization or provider-client libraries
D.5	Context length	overflow or incorrect length accounting
D.6	Prompts	missing, stale, or poorly managed prompts
E. Workflow issues		
E.1	Scheduling / loops	hangs, infinite loops, skipped steps
F. Utility issues		
F.1	Implementation	UI, Docker, logging, unrelated imports
F.2	Dependencies	non-agent libraries or internal circular imports
F.3	Configuration	I/O paths, encoding, IPs, telemetry

Checklist. Accepted if closed with a developer patch, at least one row is yes, and it is *not* a utility-only issue.

ID	Question	If yes
HB1	Mentions an LLM provider, model name, SDK, or API key?	include
HB2	Mentions prompt content, templates, or prompt management?	include
HB3	Reports memory symptoms (missing history, corrupt storage, init failures)?	include
HB4	Involves tool invocation, parameters, configuration, or implementation?	include
HB5	Describes workflow anomalies (hangs, loops, repeated actions)?	include
HB6	Patch changes LLM calls, memory, tool wrappers, prompts, or orchestration?	include

E ONE-TO-ONE ISSUE–PR–COMMIT MATCHING

Issues, PRs, and commits are many-to-many and would otherwise duplicate the same fix. For each PR p we collapse the set of k commits into a squash commit c_p^s, build a bipartite graph $G = (\mathcal{P}, \mathcal{C}^s, E)$, and keep only one-to-one edges

$$\mathcal{C}(p) = \bigcup_{j=1}^{k} \{c_j\} \longrightarrow c_p^s \qquad E^\star = \big\{(p, c^s) \in E \;\big|\; \deg_G(p) = \deg_G(c^s) = 1\big\}. \tag{1}$$

With issue–PR incidence E_{IP}, each instance is $(i, p, c^s) \in \mathcal{D}$ where $(p, c^s) \in E^\star$ and $(i, p) \in E_{IP}$. The squash commit is the patched snapshot; the PR base commit is the buggy snapshot.

F EXAMPLE OF JOINT OPTIMIZATION

AgentScope issue #1297 (`https://github.com/agentscope-ai/agentscope/issues/1297`) reports that a disconnected Studio hook raises a `ConnectionError` after retrying and crashes the agent. Its developer patch logs the failure and returns, but it contains no IPT. This instance exposes why independent artifact refinement can stall when both the import path and the test oracle are misaligned.

Observed trace for AgentScope issue #1297

Point in pipeline	Observed evidence
Before co-fix	All 25 attempts across five epochs to optimizing the test still remained `f2f` (i.e., the test fails on both buggy and patched commits). In the final attempt, the patched run still loaded the hook from `site-packages`, while the test patched a logger location that did not correspond to the implementation's observable behavior.
Full-generation co-fix	With no IPT to restore, one response changed the container to an editable install with `/app/src` first on `PYTHONPATH`, and rewrote the test to simulate a failed `requests.post` call and assert that the real agent reply completes after the expected retries.
Re-verification	The same command returned exit code 1 on the buggy snapshot and returned exit code 0 after applying the patch, yielding the first `f2p` (i.e., failed on the buggy commit and passing on the patched commit) outcome for the instance.

This trace demonstrates the role of full-generation co-fix after test-only rounds have failed: one final diagnosis can reconcile the container's source path and the test's behavioral oracle. It is a representative execution trace, not a component-wise causal ablation; without evaluating the two crossed combinations, we do not claim that each individual edit is independently necessary.

G FAILURE-MODE ANALYSIS

Figure 7 summarizes the major failure modes observed in unsuccessful reproduction runs of AGENTBUG-SMITH. Moreover, some failure distributions vary across backbone models, suggesting that some failure modes are model-dependent. The results also indicate that further improvements may require targeted enhancements at different stages of the reproduction pipeline, such as

Table 7: Failure modes of unsuccessful reproduction runs.

Failure mode	DeepSeek-v3.2	GPT-4.1-mini	Kimi-k2.5
Single cause: environment			
Dependency installation	44	33	19
Docker build configuration	19	15	40
Missing imports or incompatible APIs	15	25	7
Python packaging metadata	15	27	4
Native build or compiler	12	28	5
Container disk exhaustion	12	0	40
Missing verifier or Docker logs	11	5	56
Single cause: test			
Oracle or exercised behavior	17	30	4
Test command crashed or timed out	4	9	1
Coupled cause			
Environment and test together	13	8	3
Total unsuccessful runs	162	180	179

more robust dependency resolution, environment construction, and execution-feedback collection. In particular, future efforts could prioritize improving dependency installation and Docker image construction, as well as preserving diagnostic logs when container disk exhaustion occurs.

H Stability Metric

To check stability, we independently classify each retained issue $m{=}5$ times. Let A_i be the set of issues labeled *Yes* in run i, and let p_j be the fraction of *Yes* answers for issue j. We report set agreement and normalised entropy

$$S_{\text{set}} = \frac{\left|\bigcap_{i=1}^{m} A_i\right|}{\left|\bigcup_{i=1}^{m} A_i\right|} \qquad S_{\text{entropy}} = 1 - \frac{1}{n}\sum_{j=1}^{n} H(p_j), \tag{2}$$

where $H(p) = -p\log_2 p - (1-p)\log_2(1-p)$ is binary entropy. Both scores are 1 when every run returns the same label for every issue.

I Skill Distillation Process

This section presents the concrete process of skill distillation, which basically follow the common pipelines design in previous work (Zhao et al., 2024; Wang et al., 2024b). The skill π is a human-readable `SKILL.md` document paired with a set of lightweight pattern scanners. It is neither a fine-tuned model but a plain-text instruction that the agent reads as part of its prompt. The pipeline refines π iteratively over the training set through the following steps.

1. **Initialization.** The pipeline begins with an empty skill, and the agent attempts each training issue $i \in \mathcal{I}_{\text{tr}}$ with no guidance beyond its default prompt.
2. **Failure diagnosis.** When the agent fails on a training issue, the pipeline diagnoses whether the failure matches a recognizable bug shape. For example, an overly narrow string predicate, an unhandled empty payload at an API boundary, a missing fallback branch, or a dropped keyword argument at a call site.
3. **Generic scanner proposal.** If a failure matches a known shape, the skill is updated by adding or tightening a generic scanner: a short instruction that tells the agent what syntactic or semantic pattern to look for and what minimal edit to apply. Crucially, proposed updates are constrained to remain within the *feasible set*:

   $$\Pi_{\text{safe}} = \big\{\pi : \pi \text{ contains no repository name, file path, issue ID, or gold-patch text}\big\}, \tag{3}$$

   so that the skill cannot memorize any specific codebase. Any proposed update that references the current repository is automatically rejected:

   $$\pi_{t+1} = \begin{cases} \pi' & \text{if } \pi' \in \Pi_{\text{safe}}, \\ \pi_t & \text{otherwise.} \end{cases} \tag{4}$$

4. **Freezing.** After one pass through all training issues, the skill π_T is frozen. No further updates are made before evaluation on the held-out test set.

J Detailed analysis of fixing effectiveness

Figure 7 compares the fixing rate of studied software agents across the bugs of different harness components. Overall, tool related bugs are the one with highest fixing rate for all studied agents. Figure8 shows the unique and overlap of correctly resolved issues across the studied software agents on Live-Harness-Bench. All the studied agents collectively resolved a total of 24 unique issues, with mini-SWE-agent and OpenHands demonstrating the highest overall efficacy by participating in 18 and 17 total resolutions respectively. mini-SWE-agent achieved the strongest independent problem-solving capacity with 7 total resolutions.

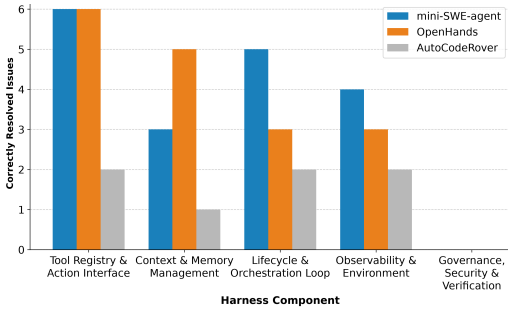

Figure 7: Effectiveness of Agents on Live-Harness-Bench.

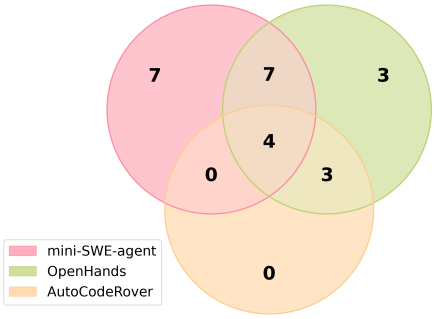

Figure 8: Venn Diagram of Resolved Bugs.

K Case: harness-sdk #362

This is one of the five new held-out fail-to-pass successes. C5 was not specialized to this repository.

Issue

`strands-agents/harness-sdk#362` reports that `system_prompt` is not passed into `structured_output`. `Agent.structured_output_async` already stores `self.system_prompt`, then calls the model with only the schema and the messages, so a fail-to-pass spy still sees `system_prompt=None`.

Without the skill

src/strands/agent/agent.py · structured_output_async()

```
events = self.model.structured_output(output_model, self.messages)
```

Function-level localization misses this call. Typical attempts edit Bedrock caching, rewrite a formatter, or emit no patch.

Generated patch

src/strands/agent/agent.py:460 · C5, plumb a missing keyword

```
- events = self.model.structured_output(output_model,
self.messages)
+ events = self.model.structured_output(
+  output_model, self.messages, system_prompt=self.system_prompt)
```

Golden patch

harness-sdk#466 · structured_output_async()

```
- events = self.model.structured_output(output_model,
self.messages)
+ events = self.model.structured_output(
+  output_model, self.messages, system_prompt=self.system_prompt)
```

The generated call-site edit is identical to gold. The original fail-to-pass test turns green, the AI judge marks the patch *correct*, and function-level localization flips from miss to hit. The other four new successes have the same chain: capability → blamed site → one-line legal edit → F2P that matches gold.