LLM-Based Agents for Identifying Bug-Introducing Commits

Niklas Risse
MPI-SP and CISPA
Germany
niklas.risse@mpi-sp.org

Marcel Böhme
MPI-SP and CISPA
Germany
marcel.boehme@acm.org

Abstract

Śliwerski, Zimmermann, and Zeller (SZZ) just won the 2026 ACM SIGSOFT Impact Award for asking:

When do changes induce fixes?

Their paper from 2005 served as the foundation for a wide array of approaches aimed at identifying bug-introducing changes (or commits) from fix commits in software repositories. But even after two decades of progress, the best-performing approach from 2025 yields a modest increase of 10 percentage points in F1-score on the most popular Linux kernel dataset.

In this paper, we uncover how and why LLM-based agents can substantially advance the state-of-the-art in identifying bug-introducing commits from fix commits. We propose a simple agentic workflow based on searching a set of candidate commits and find that it raises the F1-score from 0.64 to 0.81 on the most popular Linux kernel dataset, a bigger jump than between the original 2005 method (0.54) and the previous SOTA (0.64). We also uncover *why* agents are so successful: They derive short *greppable* patterns from the fix commit diff and message and use them to effectively search and find bug-introducing commits in large candidate sets. Finally, we also discuss how these insights might enable further progress in root cause understanding and repair.

CCS Concepts

• **Security and privacy → Software and application security**; • **Software and its engineering → Software maintenance tools**; **Software libraries and repositories**.

Keywords

bug-introducing commits, SZZ, LLM-based agents, large language models, software maintenance, mining software repositories

ACM Reference Format:
Niklas Risse and Marcel Böhme. 2026. LLM-Based Agents for Identifying Bug-Introducing Commits. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26), October 12–16, 2026, Munich, Germany*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3832783.3834337

1 Introduction

The problem of identifying bug-introducing commits from fix commits has seen remarkable interest from the software engineering research community, demonstrated by the more than 1300 citations of the award-winning paper by Śliwerski, Zimmermann, and Zeller (SZZ) [33]. The problem is also highly relevant for practitioners, since identifying bug-introducing changes is essential to determine which software versions are affected by security-critical bugs [5].

Even though many new approaches have been released since 2005 [4, 6, 7, 11, 19, 25, 26], the performance gains remain fairly limited [5]. The foundational 2005 SZZ paper [33] already achieves 0.54 F1-score on a high-quality developer-annotated dataset derived from the Linux kernel [18]. On the same dataset, the state-of-the-art method from 2025 [26] scores just 10 percentage points higher, with an F1-score of 0.64. To break the trend of small incremental improvements, entirely new approaches are needed.

In this paper, we study LLM-based agents for the task of identifying bug-introducing commits (BICs) from fix commits.

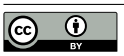
Part 1: SZZ-Agent. First, we introduce *SZZ-Agent*, which targets fix commits where the original SZZ algorithm struggles: cases where SZZ finds no candidate BICs (e.g., because the fix adds code rather than removing it), or where its prediction is likely wrong. SZZ-Agent collects candidate BICs from the file history and binary searches over them, examining the source code at each version to determine when the bug first appeared.

In our evaluation, SZZ-Agent convincingly beats all previous approaches, raising the state-of-the-art score by 9 to 13 percentage points on four popular C/C++ and Java datasets. Beyond effectiveness, we dive into *why* SZZ-Agent works using five ablation studies to investigate whether (a) the performance comes from data leakage, (b) what context SZZ-Agent relies on, (c) whether the performance gain can be explained by a better LLM-backbone, (d) what components of SZZ-Agent contribute the most to its success, and (e) how a specific hyperparameter, the commit selection threshold, controls its performance.

Part 2: Simple-SZZ-Agent. During our ablation studies, we discover that SZZ-Agent can be drastically simplified: skipping binary search entirely and letting the agent directly select from all candidates yields the best F1-score at 43% lower LLM token cost (in USD) per fix commit. Based on this finding, we propose Simple-SZZ-Agent, which collects candidates from the file history and directly asks the agent to select the bug-introducing commit—yet it outperforms SZZ-Agent. Surprisingly, it is also highly scalable: its resource consumption (e.g., tokens) is almost independent of the number of candidate commits. To understand why, we investigated what the agent actually does.

This led us to uncover *why* Simple-SZZ-Agent works so well: The agent distills the fix commit message and diff into short string patterns, greps the candidate set for matches, and then reads and evaluates the results, repeating this process until it identifies the bug-introducing commit.

To finalize our study, we propose directions for future work based on our findings. The agent's ability to compress knowledge



ASE '26, Munich, Germany

ACM ISBN 979-8-4007-2882-2/2026/10
https://doi.org/10.1145/3832783.3834337

about a bug into short greppable patterns opens avenues beyond identifying bug-introducing commits: these patterns could help practitioners understand how a bug was introduced, detect similar bugs in other code, or provide context for generating fixes.

Contributions. We make the following key contributions:

- **SZZ-Agent:** An agentic workflow that uses binary search over source code versions to identify bug-introducing commits, outperforming all previous approaches by at least 9 percentage points in F1-score on four popular datasets. We conduct five ablation studies to understand why it works.
- **Simple-SZZ-Agent:** A simpler agent discovered during our ablations that outperforms even SZZ-Agent by directly selecting the bug-introducing commit from a large candidate set. We uncover its strategy: it compresses the fix commit into short greppable patterns to efficiently search the candidates.

2 Background

Problem Definition. Given a fix commit c_{fix} that repairs a bug in a software repository, the goal is to identify the set of bug-introducing commits $B \subseteq H$, where H is the repository history prior to c_{fix}. Each commit $b \in B$ introduced a change that caused or contributed to the bug that c_{fix} resolves. In practice, $|B|$ is often one but can be larger when a bug arises from changes spread across multiple commits.

2.1 Related Work

The SZZ Algorithm. The original SZZ algorithm [33] identifies bug-introducing commits by tracing the lines deleted in a fix commit back to the commits that last modified them using version control annotations. These earlier commits form the set of bug-introducing candidates. While simple, this approach rests on the assumption that the lines removed by the fix are the same lines that introduced the bug. This assumption does not always hold: for instance, a commit may introduce a function that fails to validate its inputs, yet the fix adds a validation check rather than modifying the original function code. In such cases, the deleted lines in the fix are unrelated to the actual bug-introducing change. More generally, the approach suffers from false positives because not every deleted line is related to the bug fix, and it cannot identify bug-introducing commits for fixes that only add lines without deleting any, since there are no lines to trace back.

Filtering-based Refinements. Subsequent work tries to overcome the limitations of SZZ by refining how candidates are generated, filtered, or selected. AG-SZZ [11] uses annotation graphs and excludes non-semantic changes such as comments and formatting. MA-SZZ [6] further excludes meta-changes such as modifications to module declarations and import statements. RA-SZZ [19] integrates refactoring detection tools to remove behavior-preserving changes, which account for 6.5% of changes incorrectly flagged as bug-introducing. L-SZZ and R-SZZ [7] reduce the candidate set to a single commit: L-SZZ selects the candidate that changes the largest number of lines, while R-SZZ selects the most recent candidate. V-SZZ [4] targets security vulnerabilities specifically by recursively tracing lines back to the earliest commit that modified them, based on the observation that vulnerabilities are often introduced in early versions of the code.

Learning-based Approaches. More recent methods apply machine learning to refine which lines are traced or to assess candidates directly. Neural-SZZ [25] uses a graph neural network to rank the deleted lines in the fix commit by bug-relevance, then applies SZZ only to the top-ranked lines. LLM4SZZ [26], published at ACM ISSTA 2025 and to our knowledge the current state-of-the-art technique, uses two LLM-based strategies: rank-based identification, which selects and ranks buggy statements in the fix commit, and context-enhanced identification, which prompts the LLM with contextual information about the fix commit and each SZZ candidate to assess whether a candidate truly introduced the bug.

All of these previous approaches remain critically limited by their performance. Despite two decades of refinements, the cumulative improvement over the original SZZ remains limited to 5–10 percentage points in F1-score using the five datasets in our evaluation setup (see §3 and §4).

Software Engineering Agents. Agents are software systems that use large language models (LLMs) trained to invoke external tools such as command-line interfaces, file readers, and web search [24, 31] to achieve goals specified by a user or other agents. Agents have proven effective across a range of software engineering tasks, including automated program repair and autonomous issue resolution [9, 30, 32]. Several mature agent platforms exist today: Claude Code [2] is a widely adopted proprietary coding agent, while OpenHands [27] is a popular open-source platform that supports multiple LLM backends. LLM-based agents promise to be more effective than previous approaches at determining when a bug was first introduced, as they can reason about the semantics of code changes [8], navigate repository histories with tools, and examine source code across versions. Whether this promise holds is the central question of this paper.

3 Part 1: SZZ-Agent

To test whether the capabilities of LLM-based agents translate into effective bug-introducing commit identification, we design SZZ-Agent, an agentic approach to identify bug-introducing commits. In this section we explain how SZZ-Agent works (§3.1) and present our evaluation (§3.2–§3.3).

3.1 Approach

Figure 1 shows an overview of SZZ-Agent. It consists of two stages.

Stage 1: SZZ-based Identification. The approach starts from a fix commit, defined by its diff, showing which lines were added and/or deleted, and its commit message. Via standard SZZ, all deleted lines are traced back to the commits that last introduced them, forming the SZZ candidate set. We then provide a LLM-based agent with the fix commit message, the fix commit diff, and for each candidate commit its message and diff (all with five lines of context). The agent is instructed to analyze the fix to understand the bug, examine each candidate for whether it introduced the bug, and select the earliest bug-introducing commit, or `NONE` if no candidate is responsible.

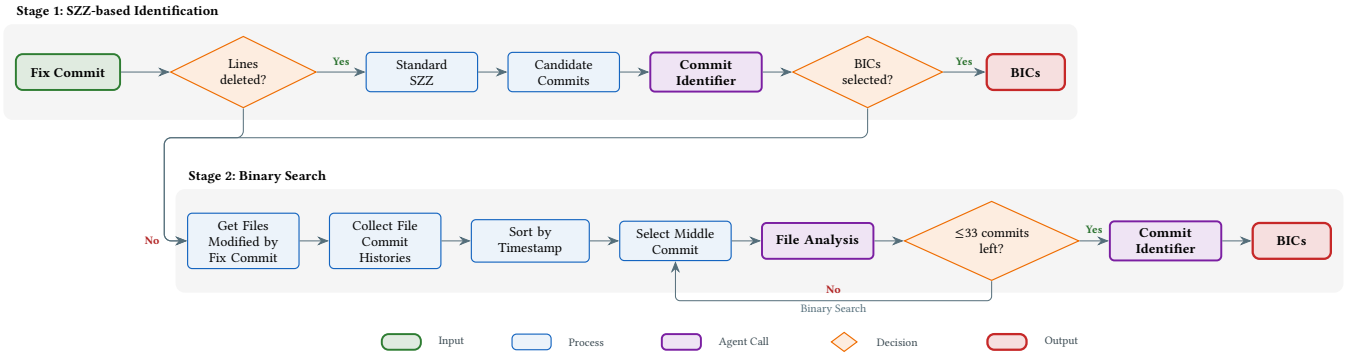


Figure 1: SZZ-Agent: Stage 1 applies standard SZZ to the deleted lines and uses an agent to select BICs from the candidates. Stage 2 activates when Stage 1 fails: it collects file histories, binary searches over commits by having an agent analyze the source code at each midpoint, and narrows the search space until the remaining candidates are few enough for direct selection.

Stage 2: Binary Search. For entries where Stage 1 abstains or where no SZZ candidates exist, we apply a second stage that operates over the full commit history of every file touched by the fix. We then perform a binary search over this history: at each step, the agent receives the full file contents at the candidate commit, the buggy version immediately before the fix, the fixed version, the fix commit message, and the fix commit diff. Based on this, the agent determines whether the bug is already present at the candidate commit. Once the search window narrows to a tunable threshold of candidates (default: 33; see §3.3.6 for an ablation), the approach switches to direct candidate selection, presenting the remaining candidates' diffs and messages for the agent to choose the most likely bug-introducing commit.

3.2 Experimental Setup

The goal of the evaluation is to investigate how effective SZZ-Agent is and why it works. To achieve this goal, we ask the following research questions:

RQ1 (Effectiveness): How effective is SZZ-Agent at identifying bug-introducing commits?

RQ2 (Ablations): Why is SZZ-Agent effective?
(a) Data Leakage: Can the effectiveness of SZZ-Agent be explained by data leakage?
(b) LLM Backbone: Is SZZ-Agent more effective than previous state-of-the-art because it uses better LLMs?
(c) Context: What contextual information enables SZZ-Agent to determine the BICs?
(d) Stage Contribution: What is the contribution of Stage 1 vs. Stage 2?
(e) Hyperparameter: How does the performance vary with the threshold for direct commit selection in Stage 2?

The following paragraphs describe the experimental setup shared across all experiments. Experiment-specific details are provided in the individual setup paragraphs in §3.3.

3.2.1 Datasets and Pre-processing. We use the most popular datasets across two programming language families, which are also the datasets used in the LLM4SZZ evaluation [26].

DS_LINUX. Since October 2013, Linux kernel developers label bug-fixing patches with the commit identifiers of the corresponding bug-introducing commits via `Fixes:` tags. Lyu et al. [18] leverage this practice to construct a developer-annotated dataset of 76,046 fix commit and bug-introducing commit pairs.

DS_LINUX-26. To evaluate on fix commits published after the training data cutoff of the LLMs we use, we apply the methodology of Lyu et al. [18] to collect 5275 fix commit and bug-introducing commit pairs from the Linux kernel for fix commits published between September 1, 2025 and January 31, 2026.

DS_GITHUB-c and DS_GITHUB-j. Rosa et al. [22] construct a developer-annotated dataset from GitHub projects by using natural language processing to identify fix commits in which developers explicitly reference the bug-introducing commits, followed by manual filtering. We split this dataset by language into DS_GITHUB-c (C/C++ projects) and DS_GITHUB-j (Java projects).

DS_APACHE. Wen et al. [28] manually collect and verify the bug-introducing commits of 333 bugs from seven projects of the Apache ecosystem. We use the version of this dataset from the LLM4SZZ evaluation [26], which retains 243 of these bugs.

Pre-processing. To prevent the agents from identifying the bug-introducing commit by matching commit hashes referenced in the fix commit message (e.g., in `Fixes:` tags), we redact commit hash references from all provided diffs and messages. Specifically, all occurrences of ground-truth commit hashes are replaced with `COMMIT_HASH`, and lines containing `Fixes:` tags that reference these commits are removed entirely. The agents cannot recover this redacted information: they operate on a working directory that contains the relevant files and diffs as separate files rather than the actual git repository, and internet access is blocked.

3.2.2 Baselines. We compare against eight SZZ variants described in §2: SZZ, AG-SZZ, L-SZZ, R-SZZ, MA-SZZ, RA-SZZ, V-SZZ, and LLM4SZZ. For all baselines except V-SZZ and LLM4SZZ, we use the implementations provided by Rosa et al. [23]. For V-SZZ and

Table 1: RQ1: SZZ-Agent vs. baselines across four datasets. Best results per dataset in bold.

	DS_LINUX			DS_GITHUB-c			DS_GITHUB-j			DS_APACHE		
Approach	**Precision**	**Recall**	**F1-Score**	**Precision**	**Recall**	**F1-Score**	**Precision**	**Recall**	**F1-Score**	**Precision**	**Recall**	**F1-Score**
SZZ	0.49	0.60	0.54	0.46	0.61	0.53	0.48	**0.68**	0.57	0.41	**0.58**	0.48
AG-SZZ	0.48	0.58	0.52	0.46	0.64	0.53	0.45	0.59	0.51	0.31	0.42	0.36
L-SZZ	0.44	0.43	0.44	0.38	0.38	0.38	0.41	0.41	0.41	0.29	0.26	0.27
R-SZZ	0.48	0.47	0.47	0.59	0.58	0.59	0.47	0.47	0.47	0.40	0.36	0.38
MA-SZZ	0.45	0.57	0.50	0.45	0.64	0.53	0.43	0.56	0.49	0.30	0.42	0.35
RA-SZZ	0.45	0.57	0.50	0.45	0.64	0.53	0.37	0.48	0.42	0.24	0.36	0.29
V-SZZ	0.44	0.55	0.49	0.46	0.61	0.53	0.48	0.67	0.56	0.40	0.57	0.47
LLM4SZZ	0.65	0.64	0.64	0.63	0.62	0.63	0.55	0.55	0.55	0.56	0.48	0.51
SZZ-Agent	**0.78**	**0.76**	**0.77**	**0.75**	**0.74**	**0.75**	**0.67**	0.67	**0.67**	**0.65**	0.56	**0.60**

LLM4SZZ, we use the implementations provided by the respective authors [4, 26]. To the best of our knowledge, LLM4SZZ [26], published at ACM ISSTA 2025, represents the state of the art.

3.2.3 Agents. We use Claude Code [2] as our agent framework. Claude Code provides 30 built-in tools [3], including bash command execution, file reading and editing, glob and grep search, web search, and sub-agent spawning—capabilities that are essential for inspecting source code across repository versions. We block all agents from using web search tools (e.g., WebSearch and WebFetch in Claude Code) to prevent them from searching the internet for solutions. As LLM backbones, we use models from Anthropic and OpenAI; see the individual experiment setups for details on which models we used. We explore open-source agents and open-source LLM backbones in Part 2 (§4.3.4).

3.2.4 Metrics. Following Lyu et al. [18], we evaluate using precision, recall, and F1-score. Since each fix commit c_{fix} can have multiple ground-truth bug-introducing commits B and multiple predicted commits $\hat{B}$, we compute per-fix-commit precision and recall as:

$$\text{Precision}(c_{\text{fix}}) = \frac{|B \cap \hat{B}|}{|\hat{B}|}, \quad \text{Recall}(c_{\text{fix}}) = \frac{|B \cap \hat{B}|}{|B|}. \tag{1}$$

We then macro-average across all fix commits and compute F1-score as the harmonic mean of the averaged precision and recall.

3.3 Experiments

3.3.1 RQ1: Effectiveness. We evaluate whether SZZ-Agent outperforms existing SZZ variants.

Setup. We run SZZ-Agent with Claude Opus 4.5 as the LLM backbone on all four established datasets: DS_LINUX (200 samples), DS_GITHUB-c (100 samples), DS_GITHUB-j (75 samples, limited by the number of Java entries in the dataset), and DS_APACHE (all 243 samples). Sample sizes were chosen based on budget considerations; we use the Wilcoxon signed-rank test [29] at $p < 0.01$ to assess statistical significance and report the rank-biserial correlation [10] as effect size. We include the sampling scripts with fixed random seeds for exact reproducibility in our artifact. We compare against all eight baselines described in §3.2.

Results. Table 1 shows the results. SZZ-Agent outperforms all baselines on every dataset. Compared to LLM4SZZ, the previous state of the art, SZZ-Agent improves F1-score by 13 percentage points on DS_LINUX (0.77 vs. 0.64), 12 on DS_GITHUB-c (0.75 vs. 0.63), 12 on DS_GITHUB-j (0.67 vs. 0.55), and 9 on DS_APACHE (0.60 vs. 0.51). SZZ-Agent achieves the highest precision and recall on all datasets, with the exception of recall on DS_GITHUB-j and DS_APACHE, where standard SZZ achieves 0.68 (vs. 0.67) and 0.58 (vs. 0.56), respectively. All improvements are statistically significant ($p < 0.01$) with large effect sizes, except for DS_GITHUB-j vs. LLM4SZZ where the effect size is large but $p = 0.013$.

3.3.2 RQ2a: Data Leakage. Since the fix commits in DS_LINUX, DS_GITHUB-c, DS_GITHUB-j, and DS_APACHE were published before the training data cutoff of Claude Opus 4.5 (August 2025 [1]), the model may have encountered them during training. We investigate whether SZZ-Agent's effectiveness can be explained by such data leakage.

Setup. We evaluate SZZ-Agent on DS_LINUX-26, which we collected by applying the same methodology as Lyu et al. [18] used for DS_LINUX but on more recent kernel data. It contains 100 randomly sampled fix commits published between September 2025 and January 2026—after the training data cutoff of Claude Opus 4.5.

Results. Table 2 shows the results. SZZ-Agent achieves 0.81 F1-score on DS_LINUX-26, which is even higher than its 0.77 on DS_LINUX. All improvements over baselines are statistically significant ($p < 0.01$) with large effect sizes. This rules out data leakage as an explanation for SZZ-Agent's effectiveness.

3.3.3 RQ2b: LLM Backbone. We investigate whether SZZ-Agent's improvement over LLM4SZZ is simply due to using a more capable LLM.

Setup. We run LLM4SZZ with three different LLM backbones—gpt-4o-mini (its default), gpt-5.2, and Claude Opus 4.5—and compare against SZZ-Agent with Claude Opus 4.5. All experiments use DS_LINUX (200 samples).

Results. Table 3 shows the results. Upgrading the LLM backbone of LLM4SZZ from gpt-4o-mini to gpt-5.2 or Claude Opus 4.5 does not meaningfully improve its F1-score (0.64 → 0.61 → 0.65). In contrast, SZZ-Agent with the same Claude Opus 4.5 backbone achieves 0.77. This demonstrates that SZZ-Agent's advantage stems from its agentic approach rather than from using a better LLM.

> ***Finding 1:*** *SZZ-Agent outperforms all previous approaches by at least 9 percentage points in F1-score on four established datasets (RQ1). This improvement cannot be explained by data leakage (RQ2a) or by the use of a more capable LLM backbone (RQ2b), but stems from the agentic approach itself.*

Table 2: RQ2a: SZZ-Agent vs. baselines on DS_LINUX-26 (post training data cutoff). Best results in bold.

Approach	Precision	Recall	F1-Score
SZZ	0.54	0.59	0.56
AG-SZZ	0.55	0.60	0.57
L-SZZ	0.52	0.51	0.51
R-SZZ	0.52	0.51	0.52
MA-SZZ	0.51	0.58	0.54
RA-SZZ	0.51	0.58	0.54
V-SZZ	0.44	0.49	0.47
LLM4SZZ	0.62	0.61	0.62
SZZ-Agent (Ours)	**0.81**	**0.80**	**0.81**

Table 3: RQ2b: LLM4SZZ with different LLM backbones vs. SZZ-Agent. Best results in bold.

Approach	Precision	Recall	F1-Score
LLM4SZZ (gpt-4o-mini)	0.65	0.64	0.64
LLM4SZZ (gpt-5.2)	0.61	0.60	0.61
LLM4SZZ (claude-opus-4-5)	0.66	0.65	0.65
SZZ-Agent (claude-opus-4-5)	**0.78**	**0.76**	**0.77**

Table 4: RQ2c: Impact of withholding the fix commit message and diff. Best results in bold.

Approach	With Message	With Diff	Precision	Recall	F1-Score
SZZ	–	–	0.54	0.59	0.56
SZZ-Agent	✗	✗	0.49	0.48	0.48
SZZ-Agent	✗	✓	0.74	0.73	0.73
SZZ-Agent	✓	✗	0.78	0.77	0.77
SZZ-Agent	✓	✓	**0.81**	**0.80**	**0.81**

3.3.4 RQ2c: Context Ablation. We investigate which contextual information enables SZZ-Agent to identify bug-introducing commits.

Setup. We ablate the fix commit message and the fix commit diff by selectively withholding them from the agent. We run all variants on DS_LINUX-26 with Claude Opus 4.5.

Results. Table 4 shows the results. Without both the commit message and the diff, SZZ-Agent drops to 0.48 F1-score, below standard SZZ (0.56). Providing only the diff recovers performance to 0.73, and providing only the message reaches 0.77—each individually already exceeding the previous state-of-the-art (LLM4SZZ, 0.62 on DS_LINUX-26). This suggests that the two sources carry partially redundant information. However, combining both achieves 0.81, indicating that each source also provides complementary information not captured by the other.

> ***Finding 2:*** *The fix commit message and the fix commit diff carry partially redundant information: each alone already surpasses the previous state-of-the-art. However, combining both yields the highest performance (0.81 F1), showing that neither source fully subsumes the other.*

3.3.5 RQ2d: Stage 1 vs. Stage 2. We investigate the individual contribution of each stage to the overall pipeline.

Table 5: RQ2d: Contribution of Stage 1 and Stage 2. Costs are average per fix commit. Best results in bold.

Approach	Precision	Recall	F1-Score	Cost (USD)
Stage 1 Only	0.48	0.47	0.48	**$0.18**
Stage 2 Only	0.77	0.76	0.76	$1.20
SZZ-Agent	**0.81**	**0.80**	**0.81**	$0.67

Setup. We run Stage 1 and Stage 2 in isolation and compare against the combined pipeline. All experiments use DS_LINUX-26 with Claude Opus 4.5. We also report the average cost per fix commit in USD.

Results. Table 5 shows the results. Stage 1 alone achieves only 0.48 F1-score at $0.18 per fix commit, while Stage 2 alone reaches 0.76 at $1.20. The combined pipeline achieves the best F1-score of 0.81 at $0.67, demonstrating that both stages are complementary: Stage 1 cheaply resolves straightforward cases, reducing the number of entries that require the more expensive Stage 2.

> ***Finding 3:*** *Stage 1 and Stage 2 are complementary. Stage 1 cheaply resolves straightforward cases, while Stage 2 handles the remaining difficult cases. Their combination achieves the best F1-score at moderate cost.*

3.3.6 RQ2e: Commit Selection Threshold. We investigate how the commit selection threshold—the number of remaining candidates at which Stage 2 switches from binary search to direct selection—affects performance and cost.

Setup. We vary the threshold across values of 8, 32, 128, 512, and ∞ (no binary search, always direct selection). All experiments use DS_LINUX-26 with Claude Opus 4.5.

Results. Table 6 shows the results. Performance is remarkably stable across thresholds, ranging from 0.76 to 0.82 F1-score. Higher thresholds reduce cost ($0.68 at threshold 8 vs. $0.33 at ∞) because fewer binary search steps are needed. The ∞ threshold—which skips binary search entirely and always uses direct selection—achieves the best F1-score (0.82) at the lowest cost ($0.33).

> ***Finding 4 (Most Surprising):*** *Skipping binary search entirely and always using direct candidate selection yields the best F1-score (0.82) at the lowest cost. The agent can identify the bug-introducing commit from a large candidate set in a single step, without iterative narrowing. This observation led us to question whether binary search is necessary at all, and motivated the development of a drastically simplified approach: Simple-SZZ-Agent (§4).*

4 Part 2: Simple-SZZ-Agent

Our experiment on varying the commit selection threshold (RQ2e, §3.3.6) revealed that skipping binary search entirely and always using direct candidate selection yields the best performance at the lowest cost. If the agent can identify the bug-introducing commit from a large candidate set in a single step, do we need the complexity

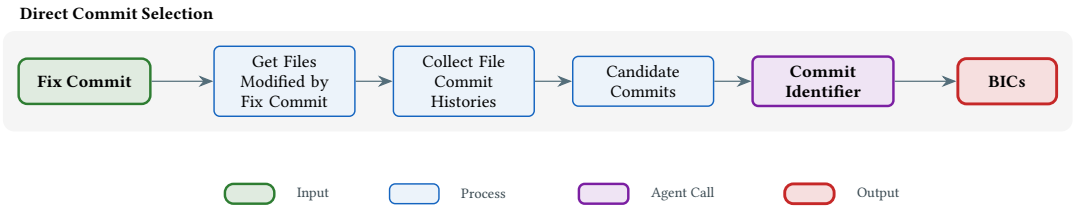


Figure 2: Simple-SZZ-Agent: The fix commit's file histories are collected into a single candidate set. An agent with access to developer tools directly selects the bug-introducing commit, without SZZ-based filtering or binary search.

Table 6: RQ2e: Impact of the commit selection threshold. Costs are average per fix commit. Best results in bold.

Commit Identifier Threshold	Precision	Recall	F1-Score	Cost (USD)
8	0.76	0.75	0.76	$0.68
32	0.81	0.80	0.81	$0.67
128	0.80	0.79	0.80	$0.39
512	0.81	0.80	0.81	$0.35
∞	**0.82**	**0.81**	**0.82**	**$0.33**

of SZZ-Agent at all? Motivated by this finding, we design Simple-SZZ-Agent, a drastically simplified approach that removes both the SZZ-based first stage and the binary search.

4.1 Approach

Figure 2 shows an overview of Simple-SZZ-Agent. Given a fix commit, the approach collects the commit histories of all files modified by the fix, forming a single candidate set. It then provides an LLM-based agent with the fix commit message, the fix commit diff, and the full candidate set. The agent has access to standard developer tools (file reading, grep, glob) and is instructed to identify the bug-introducing commit. Since our ablations showed that omitting binary search improves both performance and cost (RQ2e), and that Stage 1 alone contributes less than Stage 2 (RQ2d), Simple-SZZ-Agent removes both: no SZZ-based filtering, no binary search, and no multi-stage pipeline—it relies entirely on the agent to find the bug-introducing commit among all candidates.

4.2 Experimental Setup

We evaluate whether Simple-SZZ-Agent can match SZZ-Agent's performance and investigate why it works.

RQ3 (Effectiveness): Can Simple-SZZ-Agent achieve a similar performance to SZZ-Agent at identifying bug-introducing commits?

RQ4 (Analysis): How can we explain the surprising success of Simple-SZZ-Agent?
(a) Strategy: How can Simple-SZZ-Agent find the needle (BIC) in the haystack (large candidate commit set)?
(b) Failures: Where does Simple-SZZ-Agent still fail?
(c) Generalization: Can we expect even better results with better agents and/or better models?

We use the same datasets, pre-processing, baselines, and metrics as in Part 1 (§3.2). Unless stated otherwise, all experiments use DS_LINUX-26 with Claude Opus 4.5 as the LLM backbone and Claude Code as the agent framework.

4.3 Experiments

4.3.1 RQ3: Effectiveness. We evaluate whether Simple-SZZ-Agent can match or exceed the performance of SZZ-Agent despite its drastically simpler design.

Setup. We run Simple-SZZ-Agent on DS_LINUX, DS_GITHUB-c, DS_GITHUB-j, and DS_APACHE, and compare against SZZ, LLM4SZZ, and SZZ-Agent.

Results. Table 7 shows the results. Simple-SZZ-Agent matches or outperforms SZZ-Agent on every dataset. On DS_LINUX, it achieves 0.81 F1-score compared to 0.77 for SZZ-Agent, an improvement of 4 percentage points. On DS_GITHUB-c, both achieve 0.75. On DS_GITHUB-j, Simple-SZZ-Agent achieves 0.75 compared to 0.67 for SZZ-Agent, an improvement of 8 percentage points. On DS_APACHE, Simple-SZZ-Agent achieves 0.65 compared to 0.60 for SZZ-Agent, an improvement of 5 percentage points. Appendix A.1 additionally reports the results of all approaches split by whether the fix commit and the bug-introducing commit overlap at the file and function level.

Finding 5: *Simple-SZZ-Agent matches or outperforms SZZ-Agent on all datasets, despite removing both the SZZ-based first stage and the binary search. A single agent call over the full candidate set is sufficient.*

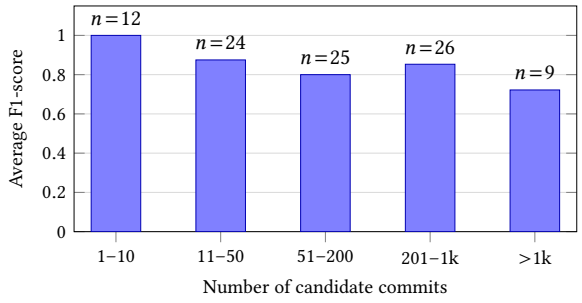
4.3.2 RQ4a: How Does the Agent Find the Needle in the Haystack? Simple-SZZ-Agent receives hundreds or even thousands of candidate commits, yet it identifies the bug-introducing commit reliably. We investigate how.

Setup. We log all tool calls and analyze the behavior of the agent on DS_LINUX-26 (100 fix commits). We record the relationship between the number of candidate commits and the agent's F1-score and resource consumption (cost, tokens, tool calls), the distribution of tool calls by type, and the sources of grep patterns.

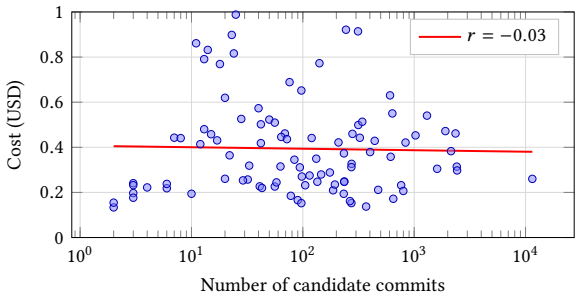
Results. We expected that scanning a large candidate set would degrade performance and require substantially more resources. To our surprise, neither turned out to be the case. Figure 3a shows the relationship between candidate set size and F1-score. Performance remains stable across candidate set sizes, with F1-scores above 0.80 for sets of up to 1,000 candidates. Only for the largest sets

Table 7: RQ3: Simple-SZZ-Agent vs. SZZ-Agent across four datasets. Best results per dataset in bold.

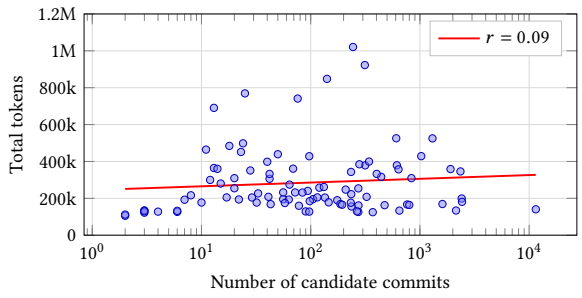
	DS_LINUX			DS_GITHUB-c			DS_GITHUB-j			DS_APACHE		
Approach	**Precision**	**Recall**	**F1-Score**	**Precision**	**Recall**	**F1-Score**	**Precision**	**Recall**	**F1-Score**	**Precision**	**Recall**	**F1-Score**
SZZ	0.49	0.60	0.54	0.46	0.61	0.53	0.48	0.68	0.57	0.41	0.58	0.48
LLM4SZZ	0.65	0.64	0.64	0.63	0.62	0.63	0.55	0.55	0.55	0.56	0.48	0.51
SZZ-Agent	0.78	0.76	0.77	**0.75**	0.74	**0.75**	0.67	0.67	0.67	0.65	0.56	0.60
Simple-SZZ-Agent	**0.82**	**0.81**	**0.81**	**0.75**	**0.75**	**0.75**	**0.75**	**0.75**	**0.75**	**0.70**	**0.60**	**0.65**



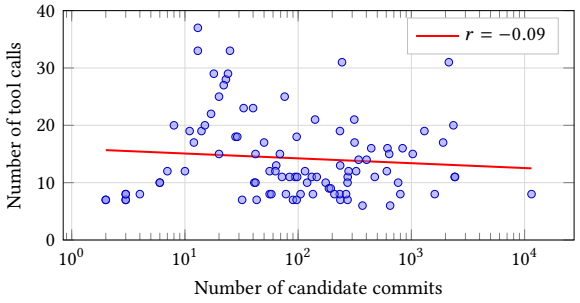
(a) F1-score.



(b) Cost per instance (USD).



(c) Total tokens consumed.



(d) Total tool calls.

Figure 3: Relationship between the number of candidate commits and F1-score, cost, token usage, and tool calls.

(>1,000 candidates) does F1-score decrease to 0.72. Figure 3 shows the relationship between the number of candidate commits and resource consumption. Cost, token usage, and tool calls show no strong correlation with the number of candidates, indicating that the agent does not scale linearly with the candidate set size.

To understand how the agent achieves this, we examined its tool usage. Figure 4 shows the distribution of tool calls. The agent predominantly uses Read (9.8 calls on average) and Grep (2.7 calls), with few calls to other tools. This reveals that the agent does not read all candidates sequentially; instead, it uses targeted searches to narrow down the candidate set. Figure 5 shows what the agent greps for. The most frequent grep patterns are derived from the removed lines of the fix commit diff (48.1% of grep calls) and the fix commit message (43.6%). The agent distills these sources into short patterns and uses them to search the candidate commits for matches. Across 243 grep calls, patterns have a median length of 21 characters (mean 25.2, std. dev. 14.8, min 6, max 104); the majority (63.8%) are plain strings rather than regular expressions.

Three examples illustrate this strategy:

(1) **Pattern from fix commit message.** The fix [15] replaces `fsleep()` with `udelay()` because `fsleep()` must not be called from atomic context. The agent greps for `"fsleep"` and directly locates the bug-introducing commit [14], which introduced the `fsleep()` call.

(2) **Pattern from removed lines.** The fix [16] corrects a firmware parser that assumed dword-aligned block sizes. The agent greps for `"blk->size » 2"`, the faulty expression removed by the fix, and matches it to the bug-introducing commit [12] that added this line. This is analogous to what SZZ does—tracing removed lines back to the commit that introduced them—but the agent performs it via grep rather than `git blame`.

(3) **Pure-addition fix.** The fix [17] adds a NULL-pointer check for `smc_ib_is_sg_need_sync()` (3 insertions, 0 deletions). Standard SZZ cannot handle this case at all, because there are no removed lines to trace back via `git blame`. The agent instead extracts the function name from the commit message, greps for `"smc_ib_is_sg_need_sync"`, and identifies the bug-introducing commit [13] that introduced this function.

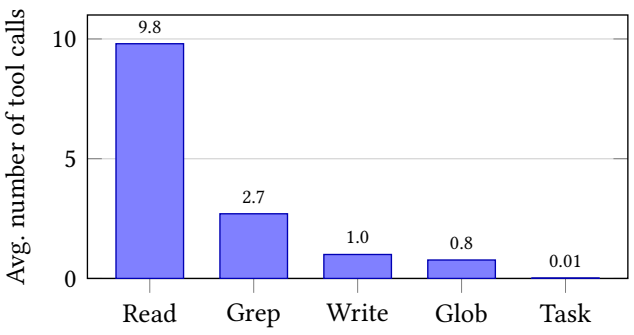


Figure 4: Distribution of tool calls per tool type during agent execution.

Finding 6 (Key): *Cost and performance are largely independent of the candidate commit set size: the agent finds the needle without examining the full haystack. It does so by deriving short greppable patterns (median 21 characters) from the fix commit diff and message and using them to search the candidate set.*

4.3.3 RQ4b: Where Does Simple-SZZ-Agent Fail? We perform a manual error analysis to understand the failure cases.

Setup. We manually inspect all incorrect predictions of Simple-SZZ-Agent on DS_LINUX-26 (100 fix commits). For each failure, we investigate the fix commit, the ground truth bug-introducing commit, the predicted bug-introducing commit, and the explanation provided by the agent, and categorize them by failure mode.

Results. Figure 6 shows the results. Of 100 fix commits, Simple-SZZ-Agent correctly identifies the bug-introducing commit in 87 cases. Of the 13 incorrect predictions, 4 fail because the true bug-introducing commit is not in the candidate set (i.e., the bug was introduced in a file not modified by the fix). The remaining 9 failures occur when the bug-introducing commit is in the candidate set but the agent does not select the correct one.

We further categorize these 9 cases through manual analysis: in 3 cases, the agent performs an incorrect analysis—it identifies the correct code area but confuses commits that modified the relevant code with the commit that originally introduced the flaw; in 2 cases, the agent selects a commit within 3 positions of the correct one (near miss), including one case where the predicted and ground truth commits are consecutive commits from the same patch series; in 3 cases, the ground truth label itself is questionable; and in 1 case, we could not determine the reason for the failure because the fix and ground truth bug-introducing commits were too complex for us to analyze as non-Linux-kernel experts. Since we are not Linux kernel experts, we also cannot determine with certainty whether the 3 questionable labels are incorrect; we include a detailed report for each case in our artifact.

Finding 7: *Of the 13 failures, 4 occur because the bug-introducing commit is not in the candidate set. Of the remaining 9, 3 are due to incorrect agent analysis, 2 are near misses, 3 involve questionable ground truth labels, and 1 we could not categorize.*

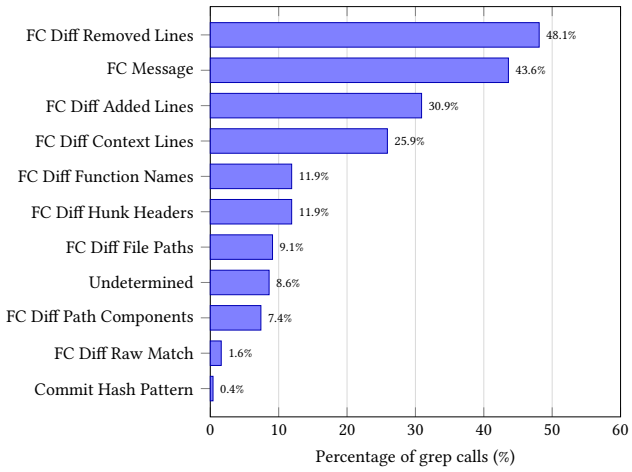


Figure 5: Distribution of grep search sources used by the agent. "FC" stands for "Fix Commit".

4.3.4 RQ4c: Better Models and Agents. We investigate whether Simple-SZZ-Agent's performance generalizes across different LLM backbones and agent frameworks, and whether cheaper or more capable configurations can improve results.

Setup. We evaluate Simple-SZZ-Agent with two agent frameworks—Claude Code and OpenHands—across multiple LLM backbones: Claude Haiku 4.5, Claude Sonnet 4.5, Claude Opus 4.5, as well as minimax-m2.5 and glm-5 for OpenHands. All experiments use DS_LINUX-26 (100 samples).

Results. Table 8 shows the results. Simple-SZZ-Agent is effective across all tested configurations. Even the cheapest configuration (OpenHands with minimax-m2.5 at $0.04 per fix commit) achieves 0.79 F1-score, outperforming all previous non-agentic approaches. The best configuration (OpenHands with Claude Sonnet 4.5) achieves 0.86 F1-score at $0.38 per fix commit, surpassing the Claude Opus 4.5 configurations. Within the same agent framework, more capable models generally perform better, though the difference between Sonnet and Opus is small or even reversed. The OpenHands configurations consume three to four times more tokens than their Claude Code counterparts, so their higher scores may be attributable to higher token usage rather than architectural differences. Equalizing token budgets across setups is not straightforward: OpenHands only supports hard cutoffs (maximum budget or iterations) that terminate execution, and Claude Opus 4.5 lacks the fine-grained thinking-effort controls of later models. Appendix A.2 breaks down the effort of each configuration by prediction outcome: in almost all configurations, unsuccessful predictions consume more resources than successful ones.

Finding 8: *Simple-SZZ-Agent generalizes across agent frameworks and LLM backbones. Even the worst-performing configuration (Claude Code with Claude Haiku 4.5, 0.74 F1) outperforms the previous state of the art by 12 percentage points, and the best configuration reaches 0.86 F1-score, suggesting that further progress with improved models and agents is likely.*

5 Discussion

We discuss directions for future work motivated by our findings. While these directions are beyond the scope of this paper, we hope they inspire other researchers to explore them. We cover research on identifying bug-introducing commits (SZZ research), root cause understanding, and program repair.

5.1 SZZ Research

Closing the Gap. Our results represent a substantial step toward solving the problem of identifying bug-introducing commits from fix commits. Simple-SZZ-Agent achieves up to 0.86 F1-score on the Linux kernel dataset, compared to 0.62 for the previous state of the art. Still, the problem is not fully solved. Our error analysis (§4.3.3) revealed that of the 13 failures, 4 occur because the bug-introducing commit is not in the candidate set, 3 are due to incorrect agent analysis, 2 are near misses, 3 involve questionable ground truth labels, and 1 we could not categorize. These failure modes point to two main areas for improvement.

First, the incorrect analyses and near misses (5 cases total) suggest that the agent's reasoning about which commit introduced a bug can still be improved. Our results in §4.3.4 show that different model and agent combinations yield meaningful performance differences, indicating that continued model improvements are a promising avenue.

Second, the 4 cases where the bug-introducing commit is not in the candidate set reveal a limitation of the current candidate collection strategy: Simple-SZZ-Agent only searches the commit histories of files modified by the fix, so when the bug was introduced in a different file, the correct commit is never considered. Appendix A.1 quantifies this limitation across all datasets: it affects 5.5% to 13.2% of cases, and the baselines also fail almost completely on this subset. How to expand the candidate set beyond the files touched by the fix without introducing prohibitive infrastructure costs is an open question that we leave for future work.

5.2 Root Cause Understanding

Our analysis in §4.3.2 revealed that agents compress information about a bug into short greppable patterns that locate its introducing changes. This compression itself may have value beyond identifying the bug-introducing commit: it captures, in a concise form, what made the code buggy.

A practitioner who knows how to fix a bug does not always understand how it was introduced. The patterns the agent derives and the bug-introducing changes it identifies could help provide this understanding. Our context ablation (§3.3.4) showed that the agent achieves 0.73 F1-score with only the diff and no commit message, demonstrating that it can extract the essential characteristics of a bug from code changes alone. Whether these patterns can be used to generate useful explanations of how a bug was introduced—especially when the fix commit message is uninformative—is an open question for future work.

5.3 Program Repair

In program repair, the goal is to generate fixes for bugs. Our approach currently operates after a fix has already been written: it takes the fix commit as input and traces back to the bug-introducing commit. An open question is whether the bug-introducing commit can also be identified from the outputs of bug oracles instead of fix commits—for example, from crash reports of a fuzzer, failing unit tests, or alerts from monitoring software. If so, agents could provide the bug-introducing changes as context for generating the fix itself, enabling agents to both locate and fix bugs. We leave this question for future work.

6 Threats to Validity

We discuss threats to the internal, external, and construct validity of our study.

6.1 Internal Validity

Sample Size. Budget constraints limit our evaluation to 75–243 fix commits per dataset, which may not capture the full distribution of bug types and fix patterns. We mitigate this by using fixed random seeds for sampling, reporting statistical significance tests and effect sizes for all comparisons with previous approaches, and providing all sampling scripts for reproducibility.

Prompt Sensitivity. The instructions given to the agent may influence its behavior and effectiveness. We did not perform systematic prompt optimization; our prompts are straightforward task descriptions and are released with our artifact.

6.2 External Validity

Language and Project Scope. Our evaluation covers C/C++ and Java projects across five datasets derived from the Linux kernel, GitHub repositories, and the Apache ecosystem. Results may not generalize to other languages or project types. We selected these datasets because they are the most popular in SZZ research and allow direct comparison with prior work.

LLM Evolution. Our results depend on current LLM capabilities and future models may yield different results. Our evaluation across six models and two agent frameworks (§4.3.4) demonstrates that the approach is not tied to a specific model.

Generalizability to Novel Projects. None of our experiments ensure that the bug-introducing commits themselves are absent from the training data; even in DS_LINUX-26, these commits may predate the training cutoff. This is not data leakage, since the information that identifies them as bug-introducing, the `Fixes:` tag and bug report, is created only after the cutoff and is therefore absent from the training data (§3.3.2). It does, however, leave open how well SZZ-Agent and Simple-SZZ-Agent transfer to novel projects whose commit history is entirely absent from the training data. Fine-tuning on the target project's commit history could mitigate this, allowing the model to learn project-specific coding patterns and commit conventions before it is applied to identify bug-introducing commits.

6.3 Construct Validity

Ground Truth Quality. We rely on developer-annotated bug-introducing commits as ground truth. For DS_LINUX and DS_LINUX-26, these annotations come from `Fixes:` tags added by Linux kernel developers, which may occasionally be incorrect or incomplete.

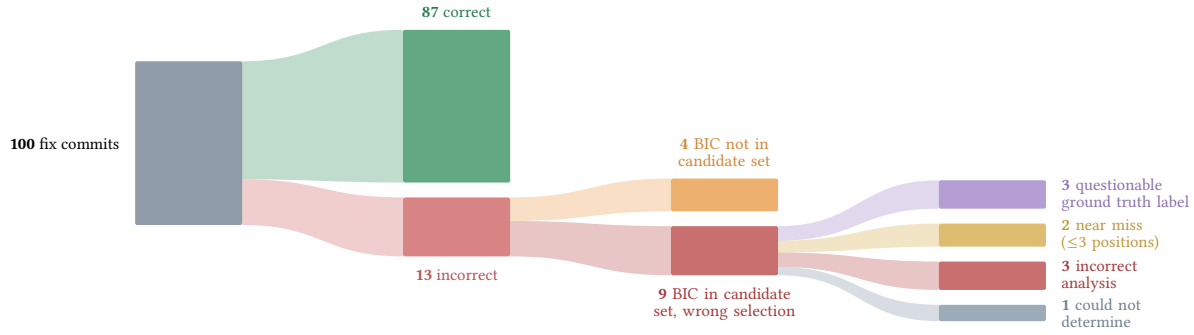


Figure 6: Manual error analysis of Simple-SZZ-Agent on 100 fix commits from DS_LINUX-26.

Table 8: RQ4c: Simple-SZZ-Agent with different underlying models and agents. Best results in bold.

					Avg. per Fix Commit		
Agent	**Model**	**Precision**	**Recall**	**F1-Score**	**Cost**	**Tool Calls**	**Tokens**
Claude Code	claude-haiku-4-5	0.75	0.73	0.74	$0.09	14.2	339k
Claude Code	claude-sonnet-4-5	0.81	0.80	0.80	$0.25	16.7	290k
Claude Code	claude-opus-4-5	0.83	0.81	0.82	$0.38	14.3	**284k**
OpenHands	claude-haiku-4-5	0.80	0.79	0.79	$0.14	15.7	1.3M
OpenHands	claude-sonnet-4-5	**0.87**	**0.86**	**0.86**	$0.38	**14.0**	1.0M
OpenHands	claude-opus-4-5	0.84	0.82	0.83	$0.63	16.9	1.0M
OpenHands	minimax-m2.5	0.80	0.78	0.79	**$0.04**	20.0	1.3M
OpenHands	glm-5	0.79	0.78	0.78	$0.26	17.8	754k

For DS_GITHUB-c and DS_GITHUB-j, the annotations were constructed by identifying fix commits where developers explicitly reference bug-introducing commits, followed by manual filtering [22]. For DS_APACHE, the bug-introducing commits were manually collected and verified by Wen et al. [28]. Inaccuracies in these labels would affect all approaches equally, but could bias absolute performance estimates.

Benchmark Validity. Risse et al. [20, 21] showed that benchmarks in machine learning for vulnerability detection admit top scores without the capability they claim to measure: the samples lack the context required to decide the label, and classifiers achieve high scores through spurious correlations. Our evaluation seems to be subject to the same critique: To identify the bug-introducing commit, the annotating developers may have drawn on information beyond the provided context, such as bug reports or runtime behavior, leaving some labels undecidable from the input alone, and the agents may exploit superficial cues such as matching removed lines (§4.3.2) rather than understanding the bug. However, neither threat invalidates our claims. We do not claim that the agents understand bugs, but that they identify the developer-annotated bug-introducing commit more accurately than all previous approaches under identical inputs and labels. Unlike function-level vulnerability detection, our benchmark task is not a proxy for a different capability but the deployment task itself: identifying the developer-annotated commit is itself the task in practice.

Single Bug-Introducing Commit Assumption. Our evaluation metrics assume that each fix commit has a well-defined set of bug-introducing commits. In practice, bugs can result from complex interactions across multiple commits, and the notion of a single introducing commit may oversimplify the true causal history. This limitation applies to all SZZ approaches and is inherited from the standard evaluation methodology [18].

7 Conclusion

We introduced SZZ-Agent, an agentic approach that uses binary search over source code versions to identify bug-introducing commits, outperforming all previous approaches by at least 9 percentage points in F1-score on four established datasets. Through five ablation studies, we established that this improvement stems from the agentic approach itself, not from data leakage or a more capable LLM backbone.

During our ablations, we discovered that binary search is unnecessary: Simple-SZZ-Agent, which gives a single agent the full candidate set, matches or outperforms SZZ-Agent on all datasets. It distills the fix commit into short greppable patterns to search the candidate set, explaining why cost and performance remain largely independent of the candidate set size. Simple-SZZ-Agent generalizes across agent frameworks and LLM backbones, and even cheap configurations outperform all previous non-agentic approaches.

Beyond identifying bug-introducing commits, the agent's ability to compress bug knowledge into short patterns opens avenues for future work in root cause understanding and program repair. We believe that agents, equipped with simple developer tools like grep, offer a powerful and general strategy for navigating large search spaces in software engineering—and that the problems we studied here are only the beginning.

Table 9: F1-score split by file-level overlap between fix commit and bug-introducing commits. The agents only search file-overlap candidates (N/A without overlap). Best F1 per column in bold; entry counts in headers.

	DS_LINUX		DS_GITHUB-c		DS_GITHUB-j		DS_APACHE	
Approach	**With** (n = 189)	**Without** (n = 11)	**With** (n = 90)	**Without** (n = 10)	**With** (n = 66)	**Without** (n = 7)	**With** (n = 211)	**Without** (n = 32)
SZZ	0.57	0.00	0.57	**0.10**	0.63	0.00	0.55	0.01
AG-SZZ	0.55	0.00	0.58	**0.10**	0.56	0.00	0.41	0.02
L-SZZ	0.46	0.00	0.42	0.00	0.46	0.00	0.32	0.00
R-SZZ	0.50	0.00	0.64	**0.10**	0.51	0.00	0.43	**0.03**
MA-SZZ	0.53	0.00	0.58	0.07	0.54	0.00	0.40	0.02
RA-SZZ	0.53	0.00	0.58	0.07	0.46	0.00	0.33	0.02
V-SZZ	0.52	0.00	0.57	**0.10**	0.61	0.00	0.54	0.01
LLM4SZZ	0.68	0.00	0.69	**0.10**	0.60	0.00	0.59	0.00
SZZ-Agent	0.81	N/A	0.82	N/A	0.74	N/A	0.68	N/A
Simple-SZZ-Agent	**0.86**	N/A	**0.83**	N/A	**0.82**	N/A	**0.75**	N/A

Table 10: F1-score split by function-level overlap between fix commit and bug-introducing commits. Best F1 per column in bold; entry counts in headers.

	DS_LINUX		DS_GITHUB-c		DS_GITHUB-j		DS_APACHE	
Approach	**With** (n = 93)	**Without** (n = 107)	**With** (n = 70)	**Without** (n = 30)	**With** (n = 49)	**Without** (n = 24)	**With** (n = 149)	**Without** (n = 94)
SZZ	0.62	0.48	0.58	0.40	0.62	0.44	0.56	0.35
AG-SZZ	0.62	0.44	0.59	0.40	0.55	0.42	0.43	0.23
L-SZZ	0.48	0.40	0.39	0.37	0.42	0.40	0.32	0.20
R-SZZ	0.57	0.39	0.65	0.43	0.48	0.44	0.48	0.22
MA-SZZ	0.58	0.43	0.59	0.39	0.52	0.42	0.44	0.22
RA-SZZ	0.58	0.43	0.59	0.39	0.46	0.34	0.35	0.20
V-SZZ	0.62	0.38	0.58	0.40	0.62	0.42	0.54	0.34
LLM4SZZ	0.76	0.54	0.68	0.50	0.62	0.40	0.60	0.38
SZZ-Agent	0.83	0.71	**0.84**	0.53	0.80	0.40	0.70	0.44
Simple-SZZ-Agent	**0.89**	**0.75**	0.83	**0.57**	**0.86**	**0.52**	**0.76**	**0.47**

Table 11: Average effort per fix commit of Simple-SZZ-Agent with different models and agents, split by successful vs. unsuccessful predictions. Errored entries are excluded. Lowest values per column in bold.

		Successful (Avg. per Fix Commit)				*Unsuccessful (Avg. per Fix Commit)*			
Agent	**Model**	**n**	**Cost**	**Tool Calls**	**Tokens**	**n**	**Cost**	**Tool Calls**	**Tokens**
Claude Code	claude-haiku-4-5	75	$0.08	**13.2**	295k	21	$0.13	17.8	494k
Claude Code	claude-sonnet-4-5	81	$0.25	16.5	**273k**	15	$0.31	17.7	383k
Claude Code	claude-opus-4-5	83	$0.38	14.0	281k	13	$0.46	16.3	**306k**
OpenHands	claude-haiku-4-5	80	$0.14	14.1	1.1M	16	$0.22	23.5	2.2M
OpenHands	claude-sonnet-4-5	87	$0.39	13.7	963k	9	$0.52	16.7	1.6M
OpenHands	claude-opus-4-5	84	$0.62	16.3	919k	12	$0.88	21.0	1.6M
OpenHands	minimax-m2.5	80	**$0.04**	19.4	1.2M	15	**$0.06**	23.5	1.6M
OpenHands	glm-5	79	$0.31	19.4	737k	15	$0.16	**9.4**	440k

A Appendix

A.1 Results Split by Fix–BIC Overlap

File-Level Overlap. Table 9 splits each dataset by whether at least one true bug-introducing commit modifies a file that the fix commit also modifies. In 5.5% (DS_LINUX) to 13.2% (DS_APACHE) of cases, there is no such overlap. Since SZZ-Agent and Simple-SZZ-Agent collect their candidate commits from the histories of the files modified by the fix, they cannot identify the true bug-introducing commit in these cases by construction; in Tables 1 and 7, they count as failures. The baselines, whose candidate sets are not restricted in this way, also fail almost completely on this subset (F1 $\leq$ 0.10).

Function-Level Overlap. Table 10 splits each dataset by whether the fix commit and at least one true bug-introducing commit modify the same function. All approaches perform substantially better when the modified functions overlap; for example, Simple-SZZ-Agent achieves 0.89 vs. 0.75 F1-score on DS_LINUX. Simple-SZZ-Agent outperforms all baselines on both subsets of every dataset, but cases without function-level overlap remain substantially harder for every approach.

A.2 Effort Split by Prediction Outcome

Table 11 splits the average effort per fix commit of each RQ4c configuration (Table 8) into successful and unsuccessful predictions. In all configurations except OpenHands with glm-5, unsuccessful predictions consume more cost, tool calls, and tokens than successful ones, suggesting that the agents search longer before settling on an incorrect prediction.

Data Availability Statement

To ensure the reproducibility of our results and to provide transparency in our research, we have made all related scripts and data publicly available in our artifact: https://doi.org/10.5281/zenodo.19336271. Detailed documentation regarding the artifact structure, environment setup, and instructions on how to reproduce the experiments is provided in the `README.md` file located in the root directory of the artifact.

Acknowledgments

This research is funded by the European Union. Views and opinions expressed are however those of the author(s) only and do not

necessarily reflect those of the European Union or the European Research Council Executive Agency. Neither the European Union nor the granting authority can be held responsible for them. This work is supported by ERC grant (Project AT_SCALE, 101179366).

References

[1] Anthropic. 2024. *Models Overview*. Anthropic PBC. Claude API Documentation. https://platform.claude.com/docs/en/about-claude/models/overview

[2] Anthropic. 2025. *Claude Code*. https://github.com/anthropics/claude-code

[3] Anthropic. 2025. *Claude Code Tools Reference*. https://code.claude.com/docs/en/tools-reference

[4] Lingfeng Bao, Xin Xia, Ahmed E. Hassan, and Xiaohu Yang. 2022. V-SZZ: automatic identification of version ranges affected by CVE vulnerabilities. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 2352–2364. doi:10.1145/3510003.3510113

[5] Xingchu Chen, Chengwei Liu, Jialun Cao, Yang Xiao, Xinyue Cai, Yeting Li, Jingyi Shi, Tianqi Sun, Haiming Chen, and Wei Huo. 2025. Vulnerability-Affected Versions Identification: How Far Are We?. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2970–2982. doi:10.1109/ASE63991.2025.00244

[6] Daniel Alencar da Costa, Shane McIntosh, Weiyi Shang, Uirá Kulesza, Roberta Coelho, and Ahmed E. Hassan. 2017. A Framework for Evaluating the Results of the SZZ Approach for Identifying Bug-Introducing Changes. *IEEE Trans. Softw. Eng.* 43, 7 (July 2017), 641–657. doi:10.1109/TSE.2016.2616306

[7] Steven Davies, Marc Roper, and Murray Wood. 2014. Comparing text-based and dependence-based approaches for determining the origins of bugs. *Journal of Software: Evolution and Process* 26, 1 (2014), 107–139. arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.1619 doi:10.1002/smr.1619

[8] Alex Gu, Baptiste Roziere, Hugh James Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida Wang. 2024. CRUXEval: A Benchmark for Code Reasoning, Understanding and Execution. In *Proceedings of the 41st International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 235)*, Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (Eds.). PMLR, 16568–16621. doi:10.48550/arXiv.2401.03065

[9] Haolin Jin, Linghan Huang, Haipeng Cai, Jun Yan, Bo Li, and Huaming Chen. 2025. From LLMs to LLM-based Agents for Software Engineering: A Survey of Current, Challenges and Future. arXiv:2408.02479 [cs.SE] doi:10.48550/arXiv.2408.02479

[10] Dave S. Kerby. 2014. The Simple Difference Formula: An Approach to Teaching Nonparametric Correlation. *Comprehensive Psychology* 3 (2014), 11.IT.3.1. doi:10.2466/11.IT.3.1

[11] Sunghun Kim, Thomas Zimmermann, Kai Pan, and E. James Jr. Whitehead. 2006. Automatic Identification of Bug-Introducing Changes. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE '06)*. IEEE Computer Society, USA, 81–90. doi:10.1109/ASE.2006.23

[12] Linux Kernel. 2020. *drm/msm/a6xx: A640/A650 GMU firmware path*. https://github.com/torvalds/linux/commit/c6ed04f856a4

[13] Linux Kernel. 2022. *net/smc: optimize for smc_sndbuf_sync_sg_for_device and smc_rmb_sync_sg_for_cpu*. https://github.com/torvalds/linux/commit/0ef69e788411

[14] Linux Kernel. 2025. *drm/amd/display: Fix 'failed to blank crtc!'*. https://github.com/torvalds/linux/commit/01f60348d8fb

[15] Linux Kernel. 2025. *drm/amd/display: use udelay rather than fsleep*. https://github.com/torvalds/linux/commit/1d66c3f2b8c0

[16] Linux Kernel. 2025. *drm/msm/a6xx: Fix GMU firmware parser*. https://github.com/torvalds/linux/commit/b4789aac9d34

[17] Linux Kernel. 2025. *net/smc: fix one NULL pointer dereference in smc_ib_is_sg_need_sync()*. https://github.com/torvalds/linux/commit/ba1e9421cf1a

[18] Yunbo Lyu, Hong Jin Kang, Ratnadira Widyasari, Julia Lawall, and David Lo. 2024. Evaluating SZZ Implementations: An Empirical Study on the Linux Kernel. *IEEE Trans. Softw. Eng.* 50, 9 (Sept. 2024), 2219–2239. doi:10.1109/TSE.2024.3406718

[19] Edmilson Campos Neto, Daniel Alencar da Costa, and Uirá Kulesza. 2018. The impact of refactoring changes on the SZZ algorithm: An empirical study. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 380–390. doi:10.1109/SANER.2018.8330225

[20] Niklas Risse and Marcel Böhme. 2024. Uncovering the limits of machine learning for automatic vulnerability detection. In *Proceedings of the 33rd USENIX Conference on Security Symposium* (Philadelphia, PA, USA) *(SEC '24)*. USENIX Association, USA, Article 238, 18 pages.

[21] Niklas Risse, Jing Liu, and Marcel Böhme. 2025. Top Score on the Wrong Exam: On Benchmarking in Machine Learning for Vulnerability Detection. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA018 (June 2025), 23 pages. doi:10.1145/3728887

[22] Giovanni Rosa, Luca Pascarella, Simone Scalabrino, Rosalia Tufano, Gabriele Bavota, Michele Lanza, and Rocco Oliveto. 2021. Evaluating SZZ Implementations Through a Developer-informed Oracle. In *Proceedings of the 43rd International Conference on Software Engineering* (Madrid, Spain) *(ICSE '21)*. IEEE Press, 436–447. doi:10.1109/ICSE43902.2021.00049

[23] Giovanni Rosa, Luca Pascarella, Simone Scalabrino, Rosalia Tufano, Gabriele Bavota, Michele Lanza, and Rocco Oliveto. 2023. A comprehensive evaluation of SZZ Variants through a developer-informed oracle. *Journal of Systems and Software* 202 (2023), 111729. doi:10.1016/j.jss.2023.111729

[24] Timo Schick, Jane Dwivedi-Yu, Roberto Dessí, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: language models can teach themselves to use tools. In *Proceedings of the 37th International Conference on Neural Information Processing Systems* (New Orleans, LA, USA) *(NIPS '23)*. Curran Associates Inc., Red Hook, NY, USA, Article 2997, 13 pages. doi:10.48550/arXiv.2302.04761

[25] Lingxiao Tang, Lingfeng Bao, Xin Xia, and Zhongdong Huang. 2024. Neural SZZ Algorithm. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering* (Echternach, Luxembourg) *(ASE '23)*. IEEE Press, 1024–1035. doi:10.1109/ASE56229.2023.00037

[26] Lingxiao Tang, Jiakun Liu, Zhongxin Liu, Xiaohu Yang, and Lingfeng Bao. 2025. LLM4SZZ: Enhancing SZZ Algorithm with Context-Enhanced Assessment on Large Language Models. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA016 (June 2025), 23 pages. doi:10.1145/3728885

[27] Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. 2024. OpenHands: An Open Platform for AI Software Developers as Generalist Agents. *arXiv* (2024). doi:10.48550/arxiv.2407.16741

[28] Ming Wen, Rongxin Wu, Yepang Liu, Yongqiang Tian, Xuan Xie, Shing-Chi Cheung, and Zhendong Su. 2019. Exploring and exploiting the correlations between bug-inducing and bug-fixing commits. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 326–337. doi:10.1145/3338906.3338962

[29] Frank Wilcoxon. 1945. Individual Comparisons by Ranking Methods. *Biometrics Bulletin* 1, 6 (1945), 80–83. doi:10.2307/3001968

[30] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. SWE-agent: agent-computer interfaces enable automated software engineering. In *Proceedings of the 38th International Conference on Neural Information Processing Systems* (Vancouver, BC, Canada) *(NIPS '24)*. Curran Associates Inc., Red Hook, NY, USA, Article 1601, 125 pages. doi:10.48550/arXiv.2405.15793

[31] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. In *International Conference on Learning Representations (ICLR)*. doi:10.48550/arXiv.2210.03629

[32] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. AutoCodeRover: Autonomous Program Improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Vienna, Austria) *(ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 1592–1604. doi:10.1145/3650212.3680384

[33] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. 2005. When do changes induce fixes? *SIGSOFT Softw. Eng. Notes* 30, 4 (May 2005), 1–5. doi:10.1145/1082983.1083147

Received 2026-03-26; accepted 2026-06-18