Automated Root-Cause Subclassification and No-Code Fix Generation for Invalid Bug Reports

Mahmut Furkan Gön*, Emre Dinç*, Tevfik Emre Sungur*, and Eray Tüzün

***Abstract*—Context: Issues faced when using software are reported in the form of bug reports. However, many bug reports are invalid, meaning they do not require code changes and are resolved with a no-code fix. Manually determining the root cause of the invalid bug reports and providing actionable resolutions by the customer support causes a serious waste of resources.**

Objective: Our goal is to introduce a standardized taxonomy for root-cause oriented invalid bug report subclassification, and perform experiments to test the accuracy of various approaches on invalid subclassification based on root causes and no-code fix generation. We study how different configurations perform on a gold-standard benchmark we have created.

Method: Using a manually curated benchmark for higher quality analysis, we experimented with vanilla large language models (LLMs), Retrieval Augmented Generation (RAG), and agentic web search to identify invalid subclasses and generate no-code fixes. We evaluated the results against manually labeled ground truth data that includes the invalid subclass and no-code fixes from the original bug reports. We measured subclass detection performance with weighted F1-Score, and assessed no-code fix suggestions using BERTScore and Judge LLM success rates.

Results: For subclassification, RAG achieves the highest overall performance with 0.66 weighted F1, slightly outperforming vanilla LLMs at 0.65 and agentic web search at 0.64. At the subclass level, performance peaks at 0.85 F1 for Non-reproducibility in the RAG pipeline with Gemini and 0.81 for Feature Request in Vanilla LLM, while Wrong Version remains the most challenging with scores between 0.00 and 0.29. For no-code fix generation, agentic web search achieves the highest overall Judge LLM success rate at 68.9%, compared to 64.4% for RAG applications and 64.9% for vanilla LLMs, with subclass-level peaks of 87.4% for Working as Designed and 72.2% for Question.

Conclusion: This study shows that retrieval augmented generation improves invalid subclassification performance, while no-code fix generation benefits more from agentic web search, indicating that different components are effective for different tasks. The generated no-code fixes demonstrate the potential to automate customer support workflows, reduce manual triage effort, and provide immediate, actionable guidance to users within bug tracking systems.

***Index Terms*—Bug Report, Invalid Bug Reports, Invalid Bug Report Subclassification, No-Code Fixes, LLM Evaluation, LLM Agents, Customer Support Agents, Retrieval Augmented Generation**

I. Introduction

Software maintenance is an essential aspect of software development. Software maintenance costs more than two trillion USD annually [1]. Most open-source and proprietary projects address software maintenance through bug reports in bug tracking platforms such as Jira[1] or GitHub Issues[2]. End users create bug reports that reflect their problems, such as implementation issues, crashes, configuration problems, or misunderstandings about the software product. These reports are then reviewed by project owners or developers, who investigate and resolve the reported problems. From the software maintainers' perspective, bug reports can be categorized as either valid bug reports that require modifications to the source code or invalid bug reports that do not require changes to the source code, hence not inherently a software defect [2], [3], [4].

A considerable portion of submitted bug reports are labeled as invalid. For example, in the Eclipse project, about **22%** of the 121,855 reported bugs were classified as invalid [2]. While invalid bug reports usually do not require changes to the source code, they may still highlight issues within the product. In an analysis of 7,000 bug reports across five open-source projects, **33.8%** of reports initially labeled as valid bugs were actually misclassified [5]. As a result, **39%** of source files marked as defective were found to have never contained an actual bug [5]. Moreover, one study found that resolving invalid bug reports can take nearly as long as resolving valid ones, **16 to 18 days** on average across two different software products [6].

There might be several root causes why a bug report can be invalid, including a misunderstanding of an intended feature [3], [7], [8], [9], user-side configuration problems [3], [7], [8], [9], or using a wrong version of the software product [3], [8], [9]. The customer support team typically identifies the root causes of invalid bug reports [10].

After identifying the root causes, the customer support team resolves these invalid bug reports using what we introduce as ***no-code fixes***, fixes that do not require changes to the source code. In this study, we argue that each potential root cause may require distinct no-code fixes. For instance, a user configuration error could be resolved by finding the required configuration on the user's machine (see Figure 1a). In contrast, an external system dependency issue could be resolved by explaining that the issue originates from an upstream external issue (see Figure 1b). The customer support team can also consider these root causes while providing no-code fix solutions to the reporters.

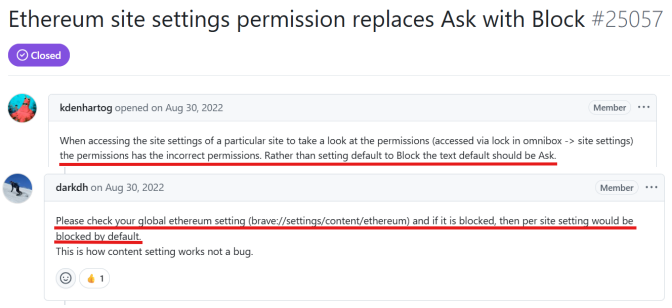
Early detection and resolution of invalid bug reports could significantly reduce the workload of all stakeholders involved in the bug tracking lifecycle, including users, customer support,

*These authors contributed equally to this work.

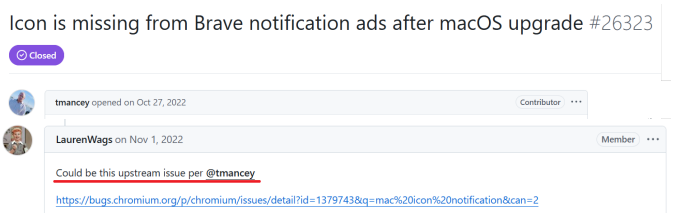
All authors are with the Department of Computer Engineering, Bilkent University, Ankara, Türkiye. Emails: {furkan.gon, emre.sungur,}@bilkent.edu.tr, eraytuzun@cs.bilkent.edu.tr, emre.dinc@ug.bilkent.edu.tr

[1]https://www.atlassian.com/software/jira

[2]https://github.com/features/issues



(a) Faulty Configuration Issue[3]



(b) External System Dependency Issue[4]

Fig. 1: Invalid Bug Report Examples with Different Root Causes

project managers/team leads, and developer teams. By identifying invalid bug reports early, the customer support team is relieved of the repetitive task of manually reviewing and classifying them. Instead, they can concentrate on verifying valid reports and reviewing automatically generated no-code fixes. Once the valid reports are classified, they are forwarded to the team leads, who then assign them to the appropriate developers. Except for a few edge cases, team leads usually prioritize valid issues that are more likely to require code changes [11]. Consequently, developers can devote their time to genuine software bugs without having to determine whether a bug report actually requires source code modification.

However, this may still leave reporters uncertain about how to resolve the invalid bug reports. Incorporating no-code fixes directly addresses this gap by guiding reporters toward the appropriate solution. As a result, it could streamline the resolution process, increasing reporter satisfaction and engagement for future contributions [10].

Some earlier studies have applied different machine learning (ML) algorithms [6], [2] to detect invalid bug reports. Some others used deep learning (DL) models [12], [13] for the same task. More recent studies have applied large language models (LLMs) [14], while others have implemented hybrid solutions that combine DL and LLM approaches [15] to detect invalid bug reports. In addition to basic validity detection, several studies have empirically investigated the root causes behind invalid bug reports [3], [8], [9], [7], [5]. However, these works are limited because they rely on manual analysis rather than proposing automated methods for invalid subclassification by root causes. Although early studies exist in this field, they do not go beyond detecting invalid bug reports or merely discussing their root causes. Invalid bug reports need to be detected, categorized by potential root causes, and resolved with no-code fixes. To the best of our knowledge, no previous study has automatically detected the root causes of invalid bug reports and generated no-code fixes to resolve them.

In our previous work [16], we employed a hybrid approach, using multiple ML models and feeding their results to a Judge LLM to determine the validity of the bug report. The objective of this study is to further develop an automated framework for subclassifying invalid bug reports by root cause and resolving them using no-code fixes. We do not include validity classification in this paper as it would shift the focus of the paper, when we want to focus deeply on invalid bug report subclassification and no-code fix suggestion. The study also aims to evaluate and compare different approaches to achieve these objectives and validate their effectiveness.

To achieve these objectives, we address the following research questions (RQs):

RQ1: How can invalid bug reports be subclassified according to their root causes?

With this research question, our aim is to identify the root causes of invalid bug reports automatically.

RQ1.1: How do LLMs perform in subclassification of invalid bug reports?

With this research question, we aim to evaluate how different LLMs perform on the subclassification task of invalid bug reports based on their root causes.

RQ1.2: Does context engineering affect the invalid bug report subclassification performance?

With this research question, we aim to evaluate the effects of retrieval augmented generation (RAG) and agentic web search on the subclassification of the invalid bug reports with LLMs.

RQ2: How can invalid bug reports be resolved with no-code fixes?

With this research question, our aim is to develop automated approaches to the crucial step of generating no-code fixes.

RQ2.1: How do LLMs perform in generating no-code fixes without subclass knowledge as prior information?

With this research question, we aim to measure the performance of LLMs without the subclass information.

RQ2.2: Does utilizing the root cause subclass information of invalid bug reports affect the no-code fix generation performance?

With this research question, we aim to examine the impact of using the root cause subclassification of invalid bug reports (RQ1) on the generation of no-code fixes.

RQ2.3: Does context engineering affect the no-code fix generation performance?

With this research question, we aim to evaluate the effects of RAG and web search tool utilization applications on no-code fix generation.

To address these RQs, we designed a multi-step evaluation framework called *IssueSupport* that leverages LLMs to subclassify invalid bug reports and generate no-code fixes. We first established a baseline (vanilla) setting, evaluating the inherent capabilities of both proprietary and open-weight LLMs to map bug reports to predefined seven invalid subclasses: **External System & Dependency Issues**, **Faulty Configuration**, **Feature Request**, **Non-reproducible**, **Question**, **Working as Designed**, and **Wrong Version**. Then, *IssueSupport* subsequently suggests

actionable no-code fixes. To isolate the impact of subclassification on the quality of generated resolutions (RQ2.1 vs RQ2.2), we conducted an ablation study that requested no-code fixes entirely without subclass priors, later comparing these outcomes with generations informed by the root cause mappings.

Building upon the vanilla baseline, we implemented an RAG pipeline to investigate the role of external context in improving classification performance (RQ1.2) and no-code fix quality (RQ2.3). Our RAG system retrieves relevant developer wiki articles and similar historically invalid bug reports utilizing dense vector search. Furthermore, we utilize a recent agentic web search framework introduced by Li et al. [17] to test the impact of web search on LLM performance. To ensure robust evaluation across all experimental paradigms, the quality of the generated no-code fixes was automatically assessed using BERTScore [18] and a Judge LLM to determine semantic and functional alignment with human-annotated ground truths. Although several classical text-matching metrics are established in the literature, they rely heavily on exact token overlaps and lack the capacity to evaluate deep contextual alignment. To avoid misleading or superficial performance variances, such metrics were intentionally excluded from our calculations.

This study contributes to the field by introducing an automated system for invalid bug subclassification into previously defined invalid subclasses and further generating no-code fixes after subclassification, which was not available in the previous studies. To the best of our knowledge, there is no particular benchmark that explicitly subclasses invalid bug report content by the root cause. We also introduce a novel benchmark comprising invalid bug reports, their invalidity subclasses, and the respective no-code fixes (if any). This dataset serves as ground truth and a benchmark for our study and possible future studies in this domain.

Our study has the following contributions:

1) A comprehensive root cause based invalid bug report subclass taxonomy and subclassification methodology.
2) A novel approach to generate no-code fixes to resolve invalid bug reports.
3) A comprehensive evaluation of the effect of prior information on the invalid bug report subclassification task.
4) A comprehensive evaluation of different experimental settings on the no-code fix generation task.
5) A new benchmark curated for no-code fix suggestions to subclassified invalid bug reports.

The remainder of this paper is organized as follows. Section II reviews the relevant literature and discusses prior studies on invalid bug reports and their classification, as well as other points concerning no-code fix generation, highlighting the research gap. Section III presents the curated benchmark and the methodology for the *IssueSupport* framework, detailing the design and workflow. Section IV reports the experimental results and the performance evaluation of different experimental settings. Section V provides a detailed discussion of the findings and their implications. Section VI presents the possible threats to validity. Finally, Section VII concludes the paper by summarizing the key outcomes of the obtained results and suggesting directions for future work.

II. Related Work and Background

In this section, we review the existing literature relevant to our study, establishing the foundational background on bug report validity and the root causes of invalid bug reports. We also examine the modern application of LLMs, with and without the additional context provided by RAG or web search. We inspect their usage in software maintenance and customer support workflows, highlighting the critical research gap in automated resolution for invalid bug reports.

A. Bug Report Validity and Automated Triaging

The high volume of incoming bug reports necessitates robust automated triaging systems. Herzig et al. [5] demonstrated that a significant percentage of bug reports are fundamentally misclassified by users, often acting as feature requests or routine tasks, which severely impacts the reliability of bug prediction models and wastes developer time. To mitigate this manual overhead, early empirical studies explored supervised [19], unsupervised [20], and semi-supervised [21] ML algorithms to automatically categorize bug reports. As the field advanced, researchers adopted DL to capture the semantic complexities of natural language descriptions [22], [23], [13].

However, treating bug validity purely as a binary classification stops short of addressing what aspects of the reports are missing for them to be valid. Recognizing this, further studies focused on extracting explanatory patterns, like the structural parts of the bug report or some keywords. He et al. [12] proposed a DL-based approach to simultaneously determine validity and extract these patterns, while transformer architectures like BERT [24] have been leveraged for similar rationale extraction.

These studies primarily treat validity as a routing or binary classification problem, stopping short of addressing the specific actions required once a report is deemed invalid.

Building on the need for actionable feedback, our previous system, Judge the Votes [16], extended beyond mere classification by providing users with automated, context-specific suggestions. Yet, this earlier work did not measure the quality of these fixes or subclassify invalid reports based on their root causes.

B. Subclassification of Invalid Bug Reports

Understanding *why* a bug report is invalid is also crucial, as the root cause might dictate the subsequent resolution strategy [3], [2]. A primary cause of invalidity is duplication, which has been extensively addressed by specialized word embedding models [25] and DL techniques [26] designed to retrieve similar historical reports.

Beyond duplicates, reports are frequently marked invalid because they stem from distinct root causes, such as user misunderstandings, faulty environment configurations, or they are feature requests mistakenly reported as bugs. In order to define these root causes, researchers have conducted empirical analyses and proposed various taxonomies to subclassify invalid bug reports.

Sun [7] attributed many invalid bug reports to testing errors or a lack of system knowledge. Similarly, Su et al. [8] found

TABLE I: Root Causes of Invalid Bug Reports Mentioned by Source

	Sources				
Identified Common Root Causes	**Laiq et al. (2023) [3]**	**Su et al. (2017) [8]**	**Panichella et al. (2021) [9]**	**Sun (2011) [7]**	**Herzig et al. (2013) [5]**
External System & Dependency Issues		Limitations and Behaviors of External Constraints		Error in External Systems	Issues Caused by Third Party Libraries
Faulty Configuration & Environment Setup	Faulty Configuration	Wrong Settings	Configuration Problem on the User Side	Wrong Configuration Parameters	
Feature Request	New Requirement		Feature Requests	Feature Requests	Feature Request
Non-reproducibility	Non-reproducible	Irreproducible Defects	Not Replicable Bug	Reproducing Not Successful	
Question			Question		
Working as Designed	Working as Expected	Working as Designed	Wrong Usage of Functionality	Misunderstanding on Functionality	Does Not Comply with User Expectations
Wrong Version - Already Fixed	Wrong Version		Problem Already Fixed with the New Version	Build was Not New Enough to Contain the Fix	Versions are Open to Backport

that 45% of invalid defects are actually "Working as Designed." Herzig et al. [5] further highlighted this expectation gap, showing that over 33% of reported bugs are simply misclassified feature requests or documentation issues. Panichella et al. [9] found that GitHub "wontfix" issues are often just support questions or out-of-scope requests. Laiq et al. [3] used topic modeling (LDA) to extract broader patterns, identifying non-reproducibility, faulty environments, and user inexperience as key drivers of invalid bug reports. The overview of the mentioned invalid bug report root causes per source is displayed in Table I. None of the previous studies focused on fully automated invalid subclass detection. As bug tracking paradigms evolve into the Generative AI era [10], [27], subclassification has become a prerequisite for routing invalid bug reports away from human developers and toward intelligent support agents.

C. LLMs and RAG in Software Maintenance

State-of-the-art LLMs have revolutionized automated bug triage by enabling sophisticated reasoning over technical documentation and bug reports [28]. For instance, the *Judge The Votes* framework [16] evaluated the performance of LLMs in validity classification, leveraging model voting mechanisms and comparisons between similar issues to determine report legitimacy. Despite these capabilities, LLMs often hallucinate when lacking project-specific context, leading to inaccurate or generic resolutions [29], [30].

To address these limitations, recent studies have heavily adopted RAG to ground LLM outputs in repository realities [31]. By retrieving relevant documents before generation, RAG ensures that the model's responses are anchored in the specific codebase and history of a project. For example, *RAG4Tickets* [32] employs semantic embeddings to retrieve historical bug reports, passing them as context to an LLM to generate context-aware ticket resolutions.

RAG has also been applied to assess the quality and authenticity of the reports themselves. A 2026 study [33] utilized RAG combined with zero-shot reasoning to successfully detect fabricated, AI-generated bug reports in bounty programs. Furthermore, tools like *ImproBR* [27] use RAG pipelines alongside LLMs to automatically detect missing execution steps in bug reports and dynamically generate improved, reproducible instructions. While these advanced systems significantly reduce the maintenance burden by contextualizing and refining code-level bugs, they largely ignore the distinct workflow required for providing direct answers to invalid bug reports, such as configuration problems or conflicting user expectations.

D. Agentic Web Search for Information Retrieval

Integrating web search agents addresses the siloed nature of traditional maintenance tools by treating the live web as dynamic external memory [17], [34]. Unlike RAG systems that rely on a static knowledge base and that can retrieve outdated or duplicate information, agentic web search, which is often framed as *Agentic Deep Research* [34], uses autonomous feedback loops to verify user-reported symptoms against real-time external sources such as upstream changelogs, third-party issue trackers, and community discussions. By going beyond internal wikis, these agents can uncover "External Dependency" or "Wrong Version" root causes that static systems would miss, providing the depth of context needed for large-scale software products.

Recent frameworks further show that technical triage often requires multi-step retrieval rather than a single query. Systems such as *WebExplorer* [35] and *BrowseMaster* [36] scale tool usage and employ specialized agent architectures to support

long-horizon reasoning on the web. In particular, *WebExplorer* enables agents to iteratively refine queries and navigate complex web structures, interactively following links and filtering developer discussions to reconstruct the chain of events leading to the problem. Finally, *WebThinker* [17], foundational framework used in our study, utilizes "Think-Search-and-Draft" architecture, which allows an LLM to interleave its internal chain-of-thought with deep navigation of web elements.

E. Bridging Customer Support and Bug Tracking with No-Code Fixes

As LLM capabilities expand, their role is shifting from passive classification tools to active resolution assistants [37], [38]. In a standard bug tracking framework, valid bug reports are routed to developers. However, invalid bugs, often stemming from user misunderstandings, faulty configurations, or external system dependencies, traditionally require time-consuming customer support intervention. This dynamic closely mirrors the challenges faced in consumer-facing applications, where a massive volume of user-reported issues are not actual system failures, but rather requests for usage workarounds or policy clarifications [39]. The fundamental nature of these consumer-facing queries is similar to the subclasses of invalid bug reports in software engineering; both scenarios involve users struggling with the intended design or configuration rather than encountering a source code defect [2]. Because consumer issues and invalid bug reports share these underlying root causes, they naturally necessitate a similar resolution process. Recent benchmarks like *ECom-Bench* [40] evaluate the capacity of multimodal LLM agents to resolve real-world customer support issues by simulating user interactions and providing immediate solutions. In app-centric ecosystems, instead of silently dismissing an invalid bug report, the AI agent should act as an advanced support bot [10], which actively resolves the issue by generating actionable *no-code fixes*, such as step-by-step UI configuration guides, explanations of design choices, or workaround instructions. Applying this customer support resolution paradigm directly to software engineering allows organizations to automate the handling of invalid bug reports, which might improve first-contact resolution rates while keeping developers strictly focused on valid bug reports, which are source code defects.

F. The Need for a No-Code Fix Benchmark

While extensive benchmarks exist for evaluating LLMs on source code generation and program repair, such as SWE-bench [41] and Livecodebench [42], there is a notable gap in benchmarking the generation of natural language no-code fixes. Recent large-scale datasets, such as *GitBugs* [43], provide over 150,000 bug reports with resolution metadata, which greatly supports tasks such as duplicate detection and automated triaging. However, datasets like GitBugs do not explicitly label bug report content by invalid subclasses, nor do they assess the semantic correctness or user-helpfulness of the actual textual responses provided to reporters. The lack of a standardized benchmark for evaluating AI-generated workarounds limits the development of automated no-code fix systems. This gap motivates the creation of our novel benchmark to systematically evaluate how well AI can generate accurate, safe, and helpful no-code fixes for invalid bug reports.

III. METHODOLOGY

In this section, we explain the curated taxonomy of invalid subclasses, the *IssueSupport* Benchmark creation steps, and the *IssueSupport* methodology.

A. Invalid Bug Report Taxonomy Based on Root Cause

The literature review leads us to a distinct taxonomy based on root causes for invalid bug reports. Unlike generic labels mostly used in GitHub repositories, *IssueSupport* Benchmark prioritizes the **root causes** of invalidity. The final clusters of invalid subclasses are **External System & Dependency Issues**, **Faulty Configuration & Environment Setup**, **Feature Request**, **Non-reproducibility**, **Question**, **Working as Designed (Conflicting Expectation)**, and **Wrong Version (Already Fixed)**. While previous studies [25], [26] often classified Duplicate reports as a subclass of invalid bugs, we exclude them from our taxonomy. This decision is twofold: first, a duplicate may still represent a valid bug that requires actual code modifications. Second, duplicates are typically resolved simply by linking to a parent issue rather than offering a distinct solution. Because our study focuses on extracting unique, context-aware no-code fixes, duplicate reports fall outside the scope of our analysis. The mapping definitions of each invalid subclass per source are given in Table I. As detailed in Table II, the taxonomy organizes these reports by three primary dimensions:

- ***Invalid Subclass (Root Cause)***: Represents the root cause of the invalidity.
- ***Description***: Outlining the specific symptoms or traits of the report.
- ***High-Level No-Code Fix Guideline***: Identifies actionable, non-invasive solutions such as configuration guidance or workarounds.

By applying this taxonomy to the real-world bug reports, we can effectively distinguish between the root causes, such as configuration mismatches, conflicting user expectations, and external dependencies. This structured approach ensures that the final benchmark is not merely a collection of invalid bug reports, but a categorized dataset optimized for automated resolution by invalid subclass.

B. IssueSupportBenchmark

1) Data Selection & Fetching: To achieve our primary objective of subclassifying invalid bug reports and generating no-code fixes, we first established a method to distinguish valid reports from invalid ones. Our dataset focuses on user-facing open-source repositories on GitHub, which could provide a rich source of bug reports.

Before detailing our dataset, it is important to clarify the terminology used in this ecosystem. In the context of GitHub, **GitHub Issues** correspond directly to **bug reports**. Throughout this paper, we treat GitHub Issues as the primary unit of analysis for bug reports, and we use the two terms interchangeably.

TABLE II: Taxonomy of Invalid Bug Reports: Root Causes, Descriptions, High-Level No-Code Fix Guidelines and Examples

Invalid Subclass (Root Cause)	Description	High-Level No-Code Fix Guideline	Example
External System & Dependency Issues	Failure triggered by defects, outages, or limitations in third-party systems or hardware outside the maintainers' control.	Direct the user to the relevant third-party system and explain that the issue originates from an external dependency.	**R:** *Brave icon is missing from Notification Ads after macOS update.* **M:** *It is an* ***upstream issue*** *origins from Chromium.* [44]
Faulty Configuration & Environment Setup	Incorrect user-side setting, parameter, or local environment mismatches before the execution.	Explain why the problem arises. Describe the required configuration and how to apply the necessary modifications.	**R:** *Tabs aren't visible or clickable after going full screen mode.* **M:** *Enabling* ***Always show toolbar in full screen*** *option will work.* [45]
Feature Request	User requests new functionality or enhancements rather than identifying a fault in the existing code.	Inform the user that the report is a feature request rather than a bug. If there is a workaround solution, explain it. Then, decide whether to implement the requested feature and notify users accordingly.	**R:** *Like Chrome & Firefox, can Brave also have a development version?* **M:** *We have already Nightly, Beta and Dev versions. No need to proceed.* [46]
Non-reproducibility	Cannot recreate the bug due to missing steps, race conditions, or one-time occurrences.	Ask the user for a detailed description, Steps to Reproduce (S2R), Expected Behavior (EB), Observed Behavior (OB), and complete environmental details if any of them are missing. If they are already known, but the bug exhibits intermittent behavior, suggest trying again at a different time.	**R:** *Crash happened in Brave Ads page.* **M:** *Blocked until we get* ***more information*** *regarding crashes.* [47]
Question	The report is an inquiry seeking help or discussion rather than a functional defect.	Provide a solution if available, or inform the user if the question cannot be answered.	**R:** *What format does Brave use to store date/time for Ads feature?* **M:** *Unix Epoch, thanks.* [48]
Working as Designed (Conflicting Expectation)	Software functions according to specifications, but the user perceives the correct behavior as a defect due to conflicting expectations.	Explain the system or specific feature in natural language.	**R:** *The browser keeps getting restarted after each update. I lose all my tabs.* **M:** *Changes only reflect when browser is restarted. This current behavior is* ***expected****.* [49]
Wrong Version (Already Fixed)	Issue exists in an outdated or unsupported version but is already resolved in a newer release.	Inform the user that the issue has already been resolved in a newer version and kindly request that the user updates to that version.	**R:** *While setting up Brave Wallet, the recovery phase shows duplicate options, indicating a posible bug.* **M:** *It was* ***already resolved*** *in version 1.35.x.* [50]

Note: The reporter (R) and the maintainer (M) actors have been referred in abbreviations at the Example column.

To ensure high data quality and relevance, we established strict static criteria for selecting our target repository:

- **Popularity:** The repository must have at least 20,000+ stars, ensuring it is a widely used and active project.
- **Volume:** The repository must contain 40,000+ bug reports to provide a statistically significant dataset.
- **Labeling:** Crucially, the repository must actively use a set of labels that explicitly categorize **invalid** bug reports.

After conducting research based on these parameters, we identified the **Brave**[5] browser repository as the ideal candidate. It not only met our volume and popularity thresholds but also maintained a rigorous labeling system for subclassifying invalid issues, making it highly suitable for our classification goals.

By mining the Brave repository with the GitHub API, we constructed a comprehensive dataset comprising **38,789** closed bug reports (captured as of January 19, 2026). Following the criteria specified in the Brave Browser wiki[6], we identified **8,289** reports as **invalid** based on labels that the project considers *invalid*. The distribution of labels within the invalid subset, with their explanations and counts, is detailed in Table III. Please note that because a single bug report can have multiple labels, the total number of labels does not match the number of invalid bug reports.

2) Evaluation Benchmark: The Evaluation Benchmark is a curated subset of invalid bug reports designed to provide a high-quality foundation for our analysis. While the initial extraction from the Brave repository yielded 8,289 reports, their native labeling system often reflects implementation-specific details rather than the fundamental reasons for invalidity. To bridge this gap, we implemented a curation pipeline that re-evaluates these reports based on the previously defined root-cause oriented **invalid subclass taxonomy**. The comprehensive workflow for this benchmark curation and manual labeling methodology is illustrated in Figure 2.

a) Automated Filtering: Starting with 8,289 invalid bug reports, we have decided to filter out some of the Brave-native invalid labels mentioned in Table III, as some of those labels' objectives are not aligned with this paper's goals. Here are the reasons for the filtered-out labels.

closed/duplicate*:* Duplicate reports were removed as they typically point to existing tickets rather than requiring unique

[5]https://github.com/brave/brave-browser

[6]https://github.com/brave/brave-browser/wiki/Issue-with-missing-milestones

TABLE III: Invalid Bug Report Labels, Descriptions, and Distribution in Brave Browser Repository [51]

Label	Description	Count
closed/duplicate	Issue has already been reported.	2285
closed/invalid	Not an issue with the browser. Possibly opened by mistake or there could be user error.	2083
closed/stale	Issue is no longer relevant, often due to inactivity or deprecated functionality.	1647
closed/not-actionable	Closed because there is no way to reproduce the error or there is no clear action the team can take.	985
closed/wontfix	Working as intended. The reported behavior will not be fixed.	925
closed/works-for-me	Issue was closed because nobody was able to reproduce.	712
closed/no-milestone	No longer an issue. Not fixed by a particular release or milestone.	297
question	Seeking inquiries or clarifications.	194
support	Addition/edits to articles on support website.	66
closed/workaround	Issue has been resolved through a workaround and does not require a client-side fix.	3
closed/fixable-by-custom-rules	Issue is fixable using custom rules or filter lists rather than code changes.	1
Total		**8289**

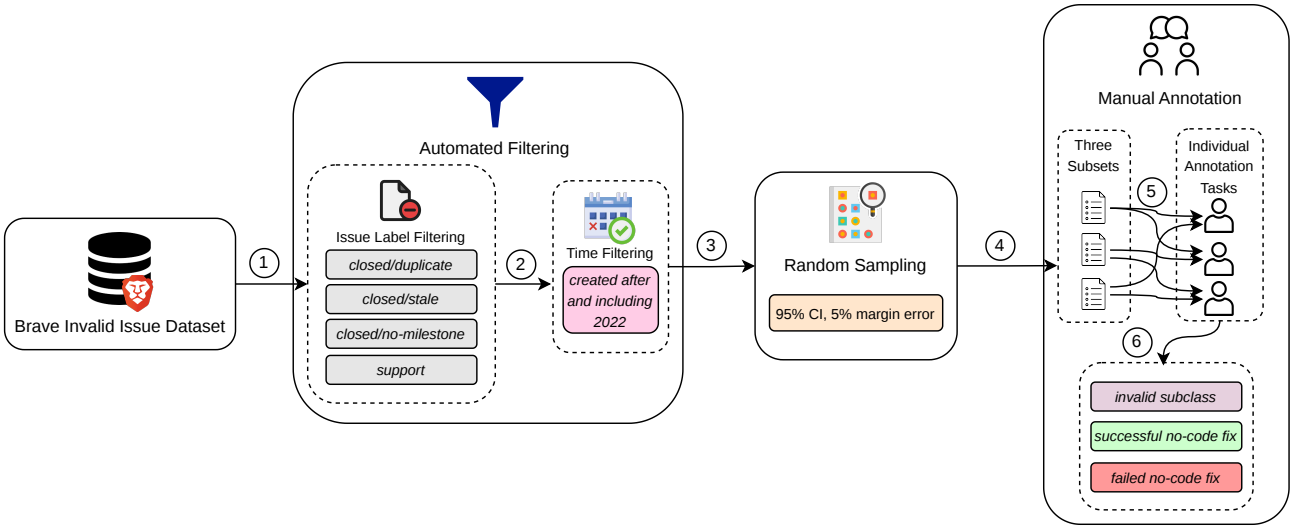


Fig. 2: Overview of Evaluation Benchmark Curation Workflow

no-code fixes.

***closed/stale**:* Excluded due to prolonged inactivity and incomplete comment threads. While these issues became irrelevant over time, this does not imply invalidity; they simply lack sufficient information to definitively determine their actual validity.

***closed/no-milestone**:* Excluded because they represent valid issues that were resolved in subsequent releases without being assigned to a specific milestone. Thus, they cannot be attributed to invalidity or user-side errors.

***support**:* Excluded because maintainers typically redirect these reporters to support channels rather than discussing issue resolution. Consequently, these reports lack the technical context and root cause analysis required to evaluate their validity.

After these exclusions, the filtered dataset consists of **4,463** invalid bug reports.

Furthermore, among these 4,463 invalid bug reports, we decided to keep the reports created after and including 2022 (6 years in until the first created issue report in 2016). Main reasons to apply this filter are:

- **Rich Knowledge Base and Discussion History**: Over several years, the Brave community has generated a vast corpus of technical discussions and community-driven solutions. By focusing on more recent reports, we leverage this established knowledge base, which provides the depth of context necessary to identify and reconstruct recurring no-code fixes.
- **Project Maturity and Maintenance Phase**: As the project reached a mature maintenance phase, it adopted standardized reporting templates and established a stable code base.

The filtered dataset consists of **1,404** invalid bug reports.

b) Random Sampling: To establish a high-quality "gold standard" for our evaluation benchmark, we conducted a manual annotation process on a statistically significant subset of the data. Using Cochran's sample size formula [52], we randomly selected **302** reports from the total pool of invalid bug reports ($N = 1,404$). This sampling strategy ensures a 95% confidence

level with a 5% margin of error. This sampled subset allows for a robust assessment of both invalidity subclassification and the extraction of proposed no-code fixes.

c) Manual Annotation Task Assignment: The manual validation of the 302 sampled invalid bug reports was conducted by three authors using a rotating-pair configuration. The dataset was partitioned into three subsets, with each subset assigned to a different pair of authors for independent review. This distribution resulted in each author assessing two-thirds of the total sample. In cases where the two primary annotators assigned conflicting labels, the third author who had not participated in the initial assessment of that specific subset reviewed the report to make the final determination.

d) Manual Annotation: Before finalizing the rules and steps of the manual evaluation, the three authors initially manually annotated seven random invalid bug reports independently. Then, the labeling reasoning and perspectives have been discussed to finalize the manual annotation keypoints for a robust annotation process.

During the manual analysis, we considered several key data points in the analysis process:

- **Issue Metadata:** We examined the title and description for clarity and quality of the demonstration of the issue, alongside the original labels assigned by developers and temporal events (e.g., the duration between comments).
- **Content and Artifacts:** We evaluated the comment threads for technical quality and relevance, including attachments like screenshots, screen recordings, and GitHub-specific reactions (e.g., *thumbs up*, *rocket* emojis) which signal community consensus.

e) Annotation Granularity: The manual annotation was conducted at two levels:

1) **Report-Level Classification:**
   - *Subclassification of Invalids:* Each report is assigned a single invalid subclass according to the taxonomy defined in Table II, identifying the specific root cause of its invalidity.
   - *Outlier Annotation:* To ensure the integrity of the benchmark and account for real-world edge cases, we introduced three outlier labels:
     a) *Valid*: Assigned to reports that, upon manual review, are found to represent actual software defects requiring a code change.
     b) *No Conclusion*: Reserved for reports where the provided content and subsequent comment threads do not contain sufficient information to reach a definitive classification.
     c) *Developer Workflow*: Reserved for reports that are opened for internal maintenance tasks that only include the details, such as the task description and acceptance criteria.
     d) *Others*: Reserved for reports that fall outside the scope of the established classification taxonomy.

   While our curation pipeline aims to isolate invalid bug reports, including these categories ensures the benchmark remains a transparent and representative reflection of the repository's reporting environment.
2) **Comment-Level Extraction:**
   - *No-Code Fix Identification:* We isolated individual comments that proposed no-code fix suggestions to the reported issue. Unlike the report-level classification, which is mutually exclusive, this is a multi-label task; a single bug report may contain multiple distinct comments identified as potential no-code fixes.
   - *Solution Verifiability:* To evaluate the reliability of these interventions, we extracted both **successful** and **failed** suggestions based on the following criteria:
     – **Successful Fixes:** Comments where the proposed workaround or clarification was explicitly validated. Evidence of success includes: (1) direct acknowledgment from the reporter via text or reaction (e.g., a thumbs-up emoji), (2) the issue is immediately closed following the suggestion and not opposed by the reporter, or (3) a maintainer closing the issue while citing the specific comment as the resolution, such as *'Issue closed based on this workaround...'*.
     – **Failed Fixes:** Comments containing suggestions that were attempted but ultimately rejected. These are identified by subsequent feedback from the reporter or maintainers indicating the solution was ineffective, such as *'This did not work'*.

f) Inter-rater Reliability: We evaluated the agreement between the annotators using Cohen's kappa (κ) for invalid subclass annotation and the Jaccard similarity coefficient for the unstructured no-code fix comment extraction. The resulting scores were $\kappa = \mathbf{0.5732}$, for all assessed bug reports, with a moderate agreement [53] for *invalid subclass* annotation. For *no-code fix comment extraction*, we observed a Jaccard similarity of $J_s = \mathbf{0.7322}$ for successful no-code fixes and $J_f = \mathbf{0.9338}$ for failed no-code fixes, confirming high agreement on the selection of the no-code fix content.

g) Final Benchmark & Observations: Following this process, we established a gold-standard evaluation benchmark containing both verified invalid subclasses and no-code fixes. The final invalid subclass distribution is given in Table IV. Based on the labeling results, 114 **(37.75%)** of the sampled bug reports do not belong to the pre-defined invalid subclass taxonomy, where 60 **(19.87%)** of these bug reports are labeled as valid, 23 **(7.62%)** of them are labeled as no conclusion and two (**0.66%**) of them were labeled as duplicate, which are considered as **Others**. This distribution highlights the practical inconsistency between the invalid label definitions and the final label annotation.

Furthermore, the benchmark details the distribution of no-code fixes across the invalid subclasses. Out of the 188 base annotations, a total of 159 successful and seven failed no-code fixes were identified. Notably, there is a pronounced need for no-code fixes in specific subcategories. Excluding Non-reproducibility and Feature Request, every single report in the remaining five invalid subclasses contained a successful no-code fix, reinforcing that their definitions heavily depend on how the fix resolves the issue. Interestingly, while External System, Faulty Configuration, and Question represent mid-to-lower overall volumes, they accounted for all seven of the

Subclass	Count	Count Percentage	Having Succesful No-Code Fix	Having Failed No-Code Fix
Working as Designed	48	15.89%	48	0
Feature Request	41	13.58%	26	0
Non-reproducibility	35	11.59%	21	0
External System & Dependency Issues	20	6.62%	20	3
Faulty Configuration & Environment Setup	19	6.29%	19	2
Question	13	4.30%	13	2
Wrong Version	12	3.97%	12	0
Base Annotations Total	**188**	**62.25%**	**159**	**7**
Valid	60	19.87%	–	–
Developer Workflow	29	9.60%	–	–
No Conclusion	23	7.62%	–	–
Others	2	0.66%	–	–
Outlier Annotations Total	**114**	**37.75%**	–	–
Total	**302**	**100.00%**	**159**	**7**

TABLE IV: Manual Annotation Results for the Sampled Invalid Bug Reports

failed no-code fix attempts in the dataset. This phenomenon underscores the potential complexity of providing actionable no-code fix for these specific operational, environmental, and design-related issues.

Particular patterns and outliers across the bug reports have emerged after manual annotation and constructive discussions between the authors.

As an example of the outlier pattern, the manually annotated two *duplicate* bug reports do not have the original Brave issue label *closed/duplicate*.

Example: In issue #24089[7], the contributor 'srirambv' directly asks, '*Dupe of #23898?*' and the author confirms, '*yep, that is a dupe*,' even though the only **invalid Brave label** on the bug report was '*closed/invalid*'.

One of the sub-patterns observed inside the *no conclusion* subset is that the issue remains stale for a subsequent time and then the particular feature related to the bug report gets deprecated in later updates.

[7]https://github.com/brave/brave-browser/issues/24089

Example: In issue #21011[8], the reporter claims that the IPFS feature inside the Brave Browser has flyout menus and by quote '*flyout menu items should have an ellipsis, as they require further user input*', which was reported in February 10, 2022. However, the issue remains stale until September 2, 2024 without any kind of clarification or suggestion about the issue. In September 2, 2024; the collaborator '*vadimstruts*' responds by quote '*The IPFS local node and scheme has been deprecated*.' Since the particular bug report remained stale for nearly 2.5 years without any fix suggestions or clarifications; no one can possibly understand whether the issue is valid or invalid.

A common pattern among Brave maintainers is the creation of "follow-up" issues that lack self-contained context.

Example: Issue #25842[9] addressed an 81% failure rate in crash dump uploads on the Brave Stable channel caused by an incorrect sampling probability. Although a patch was deployed and the issue is closed on October 20, 2022, issue #26148[10] was opened that same day as a direct follow-up. That issue particularly identified that the original problem persisted during frequent, consecutive crashes. However, the child issue relied entirely on the parent issue for context, by quote, '*This is follow-up for #25842.*'.

The poor maintenance choice is that the child issue is inherently coupled with the parent issue, and the good practice should have been the parent issue being **re-opened** after another edge-case occurs [54].

The other pattern is about the ground-truth no-code fixes. Some of the no-code fix comments left by maintainers tend to reference another bug report without sufficient explanation on what the final conclusion is.

Example: Issue #33511[11] is about the "Select and Search" function of the "Lazy Chrome" extension, which stopped working after they updated to Brave version 1.59. However, the same issue has been opened in the Lazy Chrome repository[12] and the main solution discussions took part in that issue report. As a result, in the Brave issue report, the author responds, by quote, '*Figured out due to change in brave's extension permission control frog1014/lazy_chrome#22*'.

The curated benchmark serves as a valuable asset for the software engineering research community. By providing a ground truth set of invalid bug reports paired with their specific no-code solutions, it enables the rigorous evaluation of automated classification models and solution recommendation systems, facilitating the development of tools that can alleviate the maintenance burden in open-source projects.

C. IssueSupport Methodology

We experimented with four distinct methodologies: (1) *Vanilla LLM Pipeline*, (2) *Vanilla LLM Pipeline Without Prior Invalid Subclass Information*, (3) *RAG Pipeline*, and (4) *Agentic Web Search Pipeline*. The tested methodologies return one invalid subclass and one suggested no-code fix, except for (2), and differ based on the tools and sources they use. Figure 3 presents the overall methodology.

[8]https://github.com/brave/brave-browser/issues/21011
[9]https://github.com/brave/brave-browser/issues/25842
[10]https://github.com/brave/brave-browser/issues/26148
[11]https://github.com/brave/brave-browser/issues/33511
[12]https://github.com/frog1014/lazy_chrome/issues/22

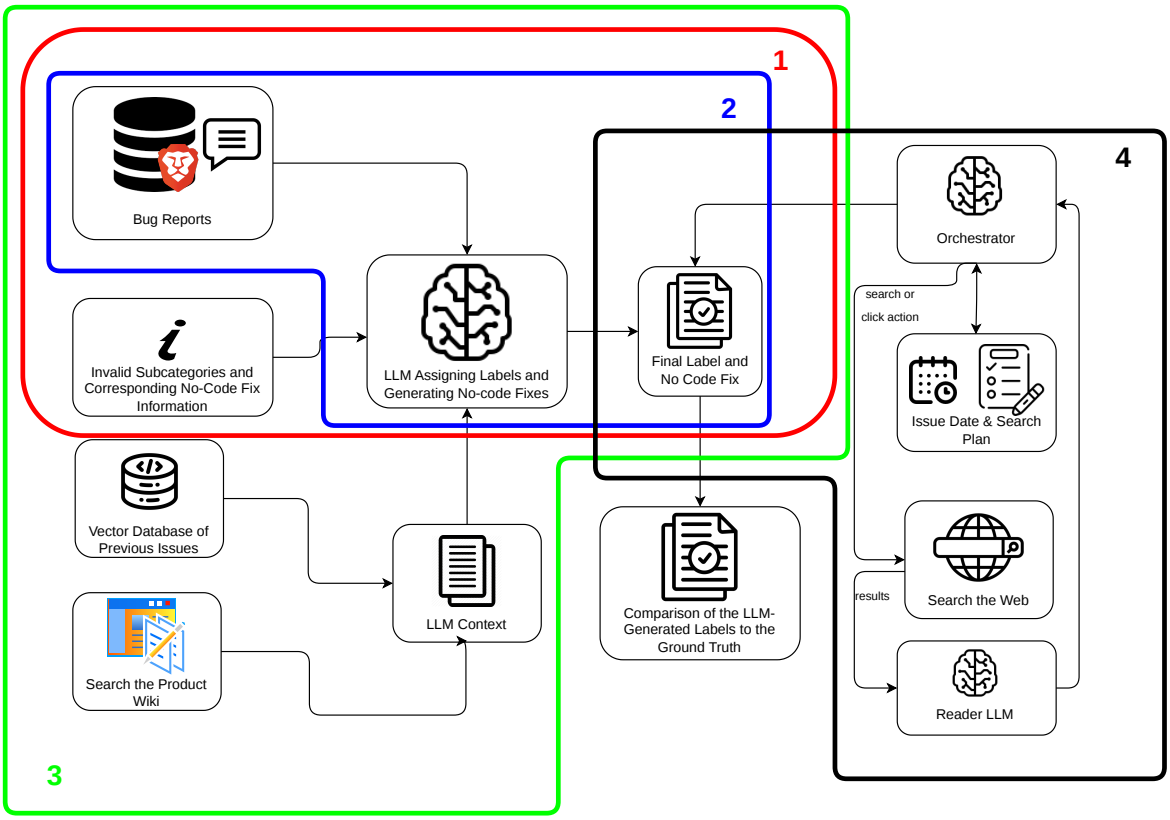


Fig. 3: Overview of *IssueSupport* Methodology

1) Prompt Configurations: All mentioned approaches utilize **zero-shot** prompting strategy, consisting of two distinct components: a system prompt and a user prompt.

The system prompt constructs the task's context by formally defining both an *invalid bug report* and a *no-code fix*. It instructs the model to classify the given report into a specific invalid subclass and generate an appropriate no-code fix. To guide this reasoning, the prompt incorporates the subclass descriptions and high-level no-code fix guidelines outlined in Table II, except for approach (2). Finally, the model is strictly instructed to return its response in a structured JSON format with the given three fields: *classification* (except for (2)), *reasoning*, and *no_code_fix*. Since approach (2) is for ablation study purposes, it lacks subclass descriptions and guidelines for the no-code fixes in the system prompt.

Conversely, the user prompt serves strictly as the data input layer, injecting only the raw markdowns of *title* and *body* of the bug report for vanilla LLM pipelines, and additionally *retrieved information* for RAG and agentic web search pipelines. The complete base system and user prompt templates are available in the `prompt.py` file of our replication package[13].

2) Vanilla LLM Pipeline: The *Vanilla LLM* pipeline serves as our primary zero-shot baseline. In this setup, we have evaluated three proprietary and three open-source LLMs.

These LLMs were selected based on their **Intelligence Index** from Artificial Analysis[14]. This index evaluates core capabilities, such as *reasoning* and *knowledge*, by aggregating results from diverse datasets. According to the leaderboard from March 25, 2026, selected proprietary models are Gemini 3.1 Pro Preview, GPT-5.4 (xhigh), and Claude Opus 4.6 (max), and the open-source models are GLM-5 [55], MiniMax M2.7 [56], and Kimi 2.5 [57]. In this setting, we picked the best-performing open-source and proprietary LLMs based on their overall **vanilla invalid subclassification performances** for use in other experimental setups.

3) Without Prior Invalid Subclass Information: To rigorously quantify the significance of our proposed taxonomy and structured guidelines, we formulated the *Without Prior Information* ablation pipeline. This setup mirrors the Vanilla LLM pipeline; however, all invalid subclass priors are deliberately removed from the system prompt. The LLM is neither provided with the definitions of the seven invalid subclasses nor the tailored guidelines for generating no-code fixes. Instead, the model is prompted generically as an "expert software engineer" and must deduce a no-code fix resolution relying solely on its intrinsic, pre-trained knowledge.

4) RAG Pipeline: The *RAG* pipeline is designed to ground the LLM's deductive reasoning in historical project data and project wiki. We populated two distinct vector databases: one containing **historical bug reports** and another con-

[13]https://figshare.com/s/dc79aaf924dac61a1095?file=64399383

[14]https://artificialanalysis.ai/methodology/intelligence-benchmarking

taining **official Brave Wiki documentation**[15]. We utilized the `zembed-1`[16] text embedding model, selected from the Agentset embedding models leaderboard[17], to generate vector embeddings. These embeddings are stored in `LanceDB`[18], chosen for its efficient metadata filtering, which is critical for our chronological data leakage prevention.

When evaluating a new bug report, the system uses the report's title and body as the query for a dense vector search. We employ a two-stage retrieval process: first, the system retrieves the top $k = 20$ most semantically similar candidates from each database. To strictly prevent data leakage, historical bug reports are chronologically filtered using a metadata constraint (`created_at < current_created_at`) to ensure only prior issues are retrievable.

These initial candidates are then processed by the `zerank-2`[19] cross-encoder re-ranking model, also selected from the Agentset leaderboard[20]. The reranker refines the selection to the top $k = 5$ most relevant contexts. These highest-ranked contexts are appended to the LLM prompt, providing the contextual data the model needs to formulate its final classification and no-code fix.

5) The Agentic Web Search Pipeline: To incorporate dynamic external evidence, we implemented an agentic web search pipeline loosely inspired by WebThinker-style reason–search–observe loops [17]. Given a bug report title, body, and creation date, the pipeline first constructs a compact research brief containing the issue context and the information need. An orchestrator LLM then performs a bounded information-gathering loop: it proposes natural-language search queries, retrieves web results through the Serper Search API [58], removes the canonical original issue page to mitigate direct leakage, and summarizes the retained evidence with source URLs. When useful, the agent can also request a page visit; fetched page content is compressed into the same evidence format.

The search trajectory is constrained to limit cost, latency, and noise. In our experiments, the outer loop was capped at 35 turns and at most three search actions; page visits were treated as part of the same bounded trajectory. Search results were obtained from the live web index at experiment time, rather than reconstructed from the issue creation date, which we discuss as a threat to validity. Tool observations were concatenated in chronological order and passed as retrieved context to the downstream classifier/generator, which then produced the final invalid subclass and no-code fix.

All language-model calls in the browsing orchestrator were issued through a single OpenAI-compatible client configured to OpenRouter [59]. For the OpenRouter-Kimi configuration, orchestration used `moonshotai/kimi-k2.5` with the default sampling temperature and a large token budget, while the final classifier used the same backbone with a lower temperature, the same token budget, and JSON-object response formatting when supported by the endpoint. Gemini-only agentic runs used `google/gemini-3.1-pro-preview`, with `google/gemini-2.5-flash` for query generation, page compression, and evidence summarization.

[15]https://github.com/brave/brave-browser/wiki

[16]https://zeroentropy.dev/articles/introducing-zembed-1-the-worlds-best-multilingual-text-embedding-model/

[17]https://agentset.ai/embeddings

[18]https://www.lancedb.com/

[19]https://zeroentropy.dev/articles/zerank-2-advanced-instruction-following-multilingual-reranker/

[20]https://agentset.ai/rerankers

D. Evaluation

To evaluate the effectiveness of the proposed methodologies, we executed each pipeline in three independent trials to reduce the variance that can be caused by the probabilistic nature of LLMs [60]. We kept the backbone LLMs at their default temperature settings to take advantage of their inherent reasoning capabilities, avoiding the constraints of the zero-temperature configuration.

The evaluation is structured to address our two primary research questions:

In RQ1, we assess the performance of the invalid subclassification task using the **weighted F1-Score**, which provides a reliable measure given the imbalanced distribution of invalid subclasses. For each bug report, the final classification label is determined by majority vote across the three independent runs.

In RQ2, in order to evaluate the semantic and functional utility of the generated no-code fixes, we employ a dual-metric approach.

- **BERTScore F1:** This metric provides a token-level semantic similarity analysis. We compute this metric by comparing model outputs specifically against the successful ground-truth fix, since semantic closeness to the failure case is not the target objective.
- **Judge-LLM Evaluation:** To assess practical alignment, we utilize an LLM-as-a-judge framework. The judge role is fulfilled by `google/gemini-3.1-pro-preview`, selected for its strong instruction-following and reasoning capabilities, operating at its default temperature setting. The Judge LLM is presented with a *contrastive three-part prompt* consisting of the predicted fix, a successful ground-truth no-code fix, and a failed ground-truth no-code fix. This allows the judge to perform a qualitative comparative assessment, determining whether the model's suggestion aligns with successful resolution patterns or mimics known failure modes.

For the evaluation in RQ2, we constrain our analysis to a subset of 159 bug reports that possess successful ground-truth no-code fixes, as ground-truth data is unavailable for the remainder of the benchmark.

IV. Results

We structurally organize our experimental findings to address our primary research questions regarding the invalid bug report subclassification and the no-code fix generation approach. To facilitate the reproducibility of our findings and support future research in automated subclassification and no-code fix generation for invalid bug reports, our complete replication package is publicly available at https://figshare.com/s/dc79aaf924dac61a1095.

TABLE V: Comprehensive Subclassification Performance: Vanilla, RAG, and Agentic Web Search (Weighted F1)

Subclass / Pipeline	Gemini 3.1	GPT-5.4	Opus 4.6	Kimi K2.5	GLM-5	Minimax M2.7	RAG+Gemini	RAG+Kimi	Agentic+Gemini	Agentic+Kimi
External System & Dep.	0.62	0.56	**0.71**	0.47	0.49	0.46	0.67	0.42	0.57	0.43
Faulty Config & Env.	**0.34**	0.30	0.23	**0.34**	0.24	0.07	0.27	0.26	0.32	0.29
Feature Request	**0.81**	0.76	0.74	0.77	0.75	0.75	0.79	0.76	0.77	0.79
Non-reproducibility	0.79	0.81	0.77	0.63	0.78	0.66	**0.85**	0.76	0.76	0.72
Question	0.76	0.75	0.71	0.73	0.69	0.69	**0.79**	0.71	0.76	**0.79**
Working as Designed	0.66	0.60	0.49	0.64	0.57	0.49	**0.70**	0.62	0.67	0.64
Wrong Version	0.00	0.00	0.19	0.26	0.00	0.25	0.00	0.26	0.20	**0.29**
Overall Weighted F1	0.65	0.61	0.59	0.60	0.58	0.53	**0.66**	0.60	0.64	0.62

*A. **RQ1: How can invalid bug reports be subclassified according to their root causes?***

To evaluate how effectively models identify the root causes of invalid bug reports, we used the Weighted F1-Score as our primary metric to account for class imbalance. The results are summarized in Table V.

*1) **RQ1.1: How do LLMs perform in subclassification of invalid bug reports?:*** Evaluating the zero-shot baseline (Vanilla LLM) reveals the intrinsic capabilities of LLMs for subclassification without external context. As shown in Table V, Gemini 3.1 Pro is the best-performing model in the Vanilla setup, achieving an overall Weighted F1-score of 0.65. Conversely, Minimax M2.7 is the worst-performing model, with an overall Weighted F1-score of 0.53.

At the subclass level, models generally exhibit strong performance in classifying *Feature Request* (e.g., Gemini 3.1 Pro at 0.81, GPT-5.4 at 0.76) and *Non-reproducibility* (e.g., GPT-5.4 at 0.81), indicating that these categories likely contain explicit linguistic cues. However, *Wrong Version* remains the most challenging category across all models in the Vanilla setup, with both Gemini 3.1 Pro and GPT-5.4 scoring a 0.00, and Minimax M2.7 scoring a marginal 0.25.

*2) **RQ1.2: Does context engineering affect the invalid bug report subclassification performance?:*** The integration of context engineering yields notable, however, model-dependent shifts in classification accuracy. According to Table V, RAG + Gemini 3.1 Pro achieves the highest overall subclassification performance across all experimental setups, reaching a Weighted F1 of 0.66. RAG boosts Gemini's performance in specific subclasses, peaking at 0.85 F1 for *Non-reproducibility* and improving *Working as Designed* to 0.70.

For Kimi K2.5, introducing an Agentic Web Search pipeline acts as the most effective context engineering method, improving its overall Weighted F1 from 0.60 (Vanilla) to 0.62, while RAG does not affect the overall Weighted F1 score. Interestingly, Agentic Web Search slightly degrades Gemini's overall subclassification Weighted F1 score to 0.64, though it successfully lifts its performance on the difficult *Wrong Version* subclass from 0.00 to 0.20. The worst-performing context-engineered setup is **RAG + Kimi K2.5**, which maintains an aggregate score of 0.60 but suffers severe degradation in the *External System & Dependency* subclass (dropping to 0.42 from 0.47 in the Vanilla setting) and in the *Faulty Configuration & Environment Setup* subclass (dropping to 0.26 from 0.34 in the Vanilla setting).

*B. **RQ2: How can invalid bug reports be resolved with no-code fixes?***

To evaluate the semantic fidelity and functional utility of generated no-code fixes, we employ BERTScore and a Judge-LLM success rate. The results are distributed across Table VI and Table VII.

*1) **RQ2.1: How do LLMs perform in generating no-code fixes without subclass knowledge as prior information?:*** As illustrated in Table VII, Gemini 3.1 Pro emerges as the best-performing model in the "Without Priors" setup, achieving an overall Judge success rate of 62.4%. Kimi K2.5 performs worst in this isolated setup, achieving an overall Judge success rate of 58.1%.

At the subclass level, Gemini 3.1 Pro demonstrates strong intuitive baseline performance in the *Faulty Configuration & Environment Setup* (70.2%) and *Working as Designed* (80.6%) categories. Conversely, Kimi K2.5 outperforms Gemini in *External System & Dependency Issues* (62.1% vs. 56.7%) and *Feature Request* (60.3% vs. 59.0%) when prior subclass knowledge is absent.

*2) **RQ2.2: Does utilizing the root cause subclass information of invalid bug reports affect the no-code fix generation performance?:*** Providing root cause subclass definitions generally improves the practical resolution capabilities of the models, though the impact varies heavily by subclass. Comparing the "Without Priors" pipeline to the "Vanilla LLM" pipeline in Table VII, Gemini 3.1 Pro sees a marginal increase in its overall Judge success rate (from 62.4% to 63.1%), while Kimi K2.5 experiences a more pronounced improvement (from 58.1% to 60.9%).

Analyzing the subclass-level shifts reveals that providing prior knowledge allows Gemini 3.1 Pro to significantly improve in *Non-reproducibility* (from 44.4% to 61.9%), though it unexpectedly drops in *Faulty Configuration & Environment Setup* (from 70.2% to 52.6%). Kimi K2.5 sees its most notable gains in *Non-reproducibility* (42.9% to 65.1%) and *Wrong Version* (37.1% to 44.4%).

*3) **RQ2.3: Does context engineering affect the no-code fix generation performance?:*** The application of context engineering mechanisms, including RAG and Agentic Web Search, yields varying results depending on the underlying model and the specific invalid subclass.

As detailed in Table VII, the Agentic Web Search pipeline combined with Gemini 3.1 Pro achieves the highest overall Judge success rate of 68.9%. This configuration demonstrates notable improvements in specific subclasses, achieving 74.4% in *Question* and increasing the *Wrong Version* success rate to 41.7% compared to its Vanilla baseline of 25.0%. Conversely,

TABLE VI: No-Code Fix Evaluation Across LLMs (Vanilla Only)

		Vanilla LLM	
Class	**Model**	**Judge (%)**	**BERTScore F1**
External System & Dependency Issues	Gemini 3.1 Pro	63.3%	**0.83**
	GPT-5.4	**73.3%**	0.82
	Opus 4.6	68.3%	0.82
	Kimi K2.5	66.7%	0.82
	GLM-5	55.0%	0.82
	Minimax M2.7	38.3%	0.81
Feature Request	Gemini 3.1 Pro	**59.0%**	**0.83**
	GPT-5.4	56.4%	0.82
	Opus 4.6	56.4%	0.82
	Kimi K2.5	56.4%	0.82
	GLM-5	50.0%	0.82
	Minimax M2.7	51.3%	0.82
Faulty Configuration & Environment Setup	Gemini 3.1 Pro	**52.6%**	**0.82**
	GPT-5.4	47.4%	0.81
	Opus 4.6	33.3%	**0.82**
	Kimi K2.5	36.8%	0.81
	GLM-5	19.3%	0.81
	Minimax M2.7	10.5%	0.81
Non-reproducibility	Gemini 3.1 Pro	61.9%	**0.81**
	GPT-5.4	**74.6%**	0.80
	Opus 4.6	68.3%	0.80
	Kimi K2.5	65.1%	0.80
	GLM-5	61.9%	0.80
	Minimax M2.7	55.6%	0.80
Question	Gemini 3.1 Pro	51.3%	**0.83**
	GPT-5.4	41.0%	0.81
	Opus 4.6	53.8%	0.82
	Kimi K2.5	**59.0%**	0.82
	GLM-5	51.3%	0.83
	Minimax M2.7	41.0%	0.82
Working as Designed (Conflicting Expectations)	Gemini 3.1 Pro	85.4%	**0.83**
	GPT-5.4	**88.9%**	0.82
	Opus 4.6	66.0%	0.82
	Kimi K2.5	75.0%	0.82
	GLM-5	70.8%	0.82
	Minimax M2.7	52.1%	0.82
Wrong Version (Already Fixed)	Gemini 3.1 Pro	25.0%	0.81
	GPT-5.4	19.4%	0.81
	Opus 4.6	**52.8%**	0.81
	Kimi K2.5	44.4%	0.81
	GLM-5	22.2%	0.81
	Minimax M2.7	38.9%	0.81
Overall	Gemini 3.1 Pro	63.1%	**0.82**
	GPT-5.4	**64.9%**	0.81
	Opus 4.6	58.4%	**0.82**
	Kimi K2.5	60.9%	0.81
	GLM-5	51.8%	**0.82**
	Minimax M2.7	42.9%	0.81

implementing Agentic Web Search with Kimi K2.5 results in an overall Judge success rate of 43.1%, the lowest among its tested configurations.

The RAG pipeline provides a different distribution of performance. RAG combined with Gemini 3.1 Pro yields an overall success rate of 64.4%, which is higher than its Vanilla performance of 63.1%. This setup achieves the highest recorded success rate in the *Working as Designed* subclass at 88.2%. However, RAG underperforms the Vanilla baseline in certain subclasses; for instance, Gemini 3.1 Pro's success rate in *Faulty Configuration & Environment Setup* drops from 52.6% in Vanilla to 49.1% with RAG, and Kimi K2.5 drops from 36.8% to 26.3% in the same category.

V. Discussion

A. Discussion on Research Questions

The experimental findings across our research questions reveal a complex interplay between model reasoning capabilities and the utility of external context. Below, we interpret these results, identify key drivers of performance, and propose methodologies to overcome the observed pitfalls.

*1) **RQ1: How can invalid bug reports be subclassified according to their root causes?**:* The results for **RQ1.1** and **RQ1.2** demonstrate that while LLMs possess an inherent baseline for bug triage, their performance is highly sensitive to the nature of the subclass. The high success in *Feature Request* and *Non-reproducibility* suggests these classes rely on explicit linguistic cues. However, the struggle with *Faulty Configuration* highlights a **Domain Specificity Gap**. The impact of context engineering (RAG and agentic web search) was non-linear. While Gemini 3.1 Pro leveraged RAG to improve its understanding of *Working as Designed* issues, it still could not identify *Wrong Version* issues. This indicates that **retrieval relevance** is not equivalent to **ground truth verification**.

*2) **RQ2: How can invalid bug reports be resolved with no-code fixes?**:* The transition from identifying a bug to resolving it (**RQ2.1** and **RQ2.2**) revealed a significant decoupling between *BERTScore* and *Judge Success Rate*. The consistent BERTScore (0.81–0.82) suggests models are excellent at mimicking the "style" of a fix. However, the lower Judge scores for models like Minimax M2.7 (42.9%) reveal a lack of **functional logic**. Consequently, researchers and tool designers should avoid relying on passive textual similarity metrics to evaluate automated triage systems. Instead, future evaluation frameworks should prioritize the generation of verifiable resolution actions, such as executable reproduction steps or configuration scripts, that allow for programmatic or human-in-the-loop validation.

Our comparison between the *Without Priors* and *Vanilla* pipelines in RQ2.1 reveals a more nuanced relationship than initially assumed. Omitting the root-cause subclass information causes only a **marginal decrease in overall performance**: the overall Judge success rate drops by just **0.7%** for Gemini 3.1 Pro and **2.8%** for Kimi K2.5. Rather than acting as a universally beneficial signal, prior subclass information has a **category-dependent effect**. Removing the prior decreases performance for some subclasses, such as *Non-reproducibility* and *External System & Dependency Issues*. Conversely, withholding the prior improves performance in certain cases; most notably, the *Without Priors* approach performs better in the *Faulty Configuration & Environment Setup* subclass, yielding higher success rates for both models.

For instance, in Brave issue 23741 in Table VIII, the *Without Priors* pipeline successfully identifies a user misconfiguration related to default wallet settings, while the other pipelines fail. We attribute this to the fact that providing root-cause subclasses forces the model to evaluate all possible invalid categories. When a bug report is missing minor details, models

TABLE VII: Class-Based No-Code Fix Evaluation (Gemini 3.1 Pro and Kimi with Different Setups)

		Without Priors		Vanilla LLM		RAG		Agentic Search	
Class	**Model**	**Judge (%)**	**BERTScore F1**	**Judge (%)**	**BERTScore F1**	**Judge (%)**	**BERTScore F1**	**Judge (%)**	**BERTScore F1**
External System & Dependency Issues	Gemini 3.1 Pro	56.7%	0.83	63.3%	0.83	63.3%	0.83	**68.3%**	0.83
	Kimi K2.5	62.1%	0.81	**66.7%**	0.82	48.3%	0.82	46.7%	0.82
Feature Request	Gemini 3.1 Pro	59.0%	0.83	59.0%	0.83	61.5%	0.83	**62.8%**	0.83
	Kimi K2.5	**60.3%**	0.81	56.4%	0.82	59.0%	0.82	42.9%	0.82
Faulty Configuration & Environment Setup	Gemini 3.1 Pro	**70.2%**	0.83	52.6%	0.82	49.1%	0.82	52.6%	0.82
	Kimi K2.5	**46.4%**	0.81	36.8%	0.81	26.3%	0.81	20.4%	0.81
Non-reproducibility	Gemini 3.1 Pro	44.4%	0.81	61.9%	0.81	66.7%	0.81	**69.8%**	0.81
	Kimi K2.5	42.9%	0.80	**65.1%**	0.80	57.1%	0.80	47.6%	0.80
Question	Gemini 3.1 Pro	64.1%	0.83	51.3%	0.83	53.8%	0.83	**74.4%**	0.83
	Kimi K2.5	51.3%	0.81	**59.0%**	0.82	**59.0%**	0.82	43.6%	0.82
Working as Designed (Conflicting Expectations)	Gemini 3.1 Pro	80.6%	0.83	85.4%	0.83	**88.2%**	0.83	84.0%	0.83
	Kimi K2.5	77.1%	0.82	75.0%	0.82	**77.8%**	0.82	54.9%	0.82
Wrong Version (Already Fixed)	Gemini 3.1 Pro	33.3%	0.81	25.0%	0.81	22.2%	0.81	**41.7%**	0.82
	Kimi K2.5	37.1%	0.80	**44.4%**	0.81	27.8%	0.81	38.9%	0.82
Overall	Gemini 3.1 Pro	62.4%	0.82	63.1%	0.82	64.4%	0.82	**68.9%**	0.82
	Kimi K2.5	58.1%	0.81	**60.9%**	0.81	55.8%	0.81	43.1%	0.82

tend to overemphasize the *Non-reproducibility* category and unnecessarily ask the reporter for more information. However, much like human maintainers, a model can often infer a configuration error despite these small omissions. By removing the need to weigh multiple subclass options, the *Without Priors* approach avoids this distraction and focuses more directly on producing the correct fix.

The findings from **RQ2.3** show that context engineering has **mixed effects** on no-code fix generation. Although additional context can improve model reasoning, its effectiveness strongly depends on the issue category.

RAG performs well in categories that benefit from **historical examples** or prior developer discussions, such as *Non-reproducibility* and *Feature Request*. For instance, if a bug report lacks required environment information or S2R, by comparing the historical examples that lack such information, LLM might easily decide the *Non-reproducibility* and generate a suitable no-code fix. Most notably, RAG outperforms Vanilla Gemini in the *Working as Designed* subclass. For example, in Brave issue 31299 (see Table VIII), the RAG-generated response correctly captured the key point: the reported increase in memory usage was an intended design trade-off rather than a bug. We attribute this improvement to RAG retrieving past reports with similar discussions about memory usage. Without this background information, the Vanilla pipeline instead asked the reporter for additional details.

However, additional context can also **degrade performance**. In highly specific categories such as *Faulty Configuration & Environment Setup* and *Wrong Version*, retrieved information appeared to distract the model from the exact issue described in the prompt. Instead of focusing on the concrete problem, the model likely generalized from retrieved documents, resulting in fewer successful fixes than the basic model. Overall, these results suggest that RAG is useful for providing historical grounding, but it may introduce unhelpful noise for strict, version-specific or configuration-specific problems.

Agentic web search produced the best overall performance for Gemini 3.1 Pro (**68.9%**), but caused a substantial performance drop for Kimi K2.5 (**43.1%**). One possible explanation is that agentic loops introduce **recursive noise**. For less capable models, each search step creates another opportunity to drift from the original user intent, potentially leading to a *hallucination spiral* [61]. To stabilize agentic performance, future systems could constrain the search process through predefined logical transitions, such as Search $\rightarrow$ Verify $\rightarrow$ Summarize, rather than allowing the model to determine its own path fully autonomously. However, prior work suggests that such constrained approaches may underperform more flexible agentic search methods [62].

Another key finding is the improvement in the challenging *Wrong Version* category, where Agentic Web Search increased the success rate from **27.3%** to **42.4%**. For example, in Brave issue 34522 in Table VIII, the Agentic Web Search pipeline correctly identified that the issue had already been resolved in version 1.62.49, whereas the Vanilla pipeline incorrectly treated the behavior as intentional. Although the ground-truth fix points to version 1.61.86, version 1.62.49 was committed on November 13, 2023. Since this predates the bug report on November 24, 2023, the suggested version was still a valid and available upgrade for the user. This example demonstrates that standard LLM and RAG approaches may lack the external temporal context needed to recognize when a reported issue has already been patched in a recent release.

B. Implications for Researchers

Our findings suggest several directions for future research on automated no-code fix generation. The comparison between evaluation metrics shows that commonly used similarity-based metrics may not capture real-world effectiveness. While BERTScore measures semantic similarity, it fails to distinguish performance differences across invalid subclasses and is often non-informative. In contrast, Judge-LLM evaluation provides a

Issue Number	Generated High-Quality No-Code Fix	Generated Low-Quality No-Code Fix	Ground-Truth No-Code Fix
34522	**Agentic Web Search:** This issue has already been resolved in Brave version 1.62.49 and later releases. **Please update your browser to the latest stable version (1.62.x or newer)** to receive the fix for the sidebar button margin alignment. You can update by navigating to brave://settings/help or downloading the latest version from the official Brave website...	**Vanilla LLM**: The sidebar button's margin design is intentional and follows the UI specification. The slight asymmetry in the icon's border padding is designed for visual balance with adjacent UI elements and overall sidebar aesthetics. This is not a bug but a deliberate design choice...	**Please test this with 1.61.86 or above**. According to https://github.com/brave/brave-core/commits/1.61.x, I'm not sure 1.61.85 includes this fix. I confirmed that 1.61.86 fixed this issue. [63]
31299	**RAG:** The observed increase in baseline memory usage for a blank page is expected and stems from recent features or underlying Chromium updates included in v1.54.70+. This is not a memory leak or defect, and the browser is functioning as designed. **You do not need to take any action, but you can continue to monitor performance to ensure it remains stable in typical browsing scenarios**.	**Vanilla LLM**: To properly investigate this memory degradation issue, please provide: (1) Detailed Steps to Reproduce (S2R) - describe how to trigger the memory issue outside of the automated dashboard test...	The degradation is the result of enabling extra UAF protection (aka BackupRefPtr). So it's about +10Mb memory to have extremely important exploit protection in the browser process. **That's look like a reasonable trade off. Also we saw the similar memory differences when it was enabled for win**. [64]
23741	**Vanilla LLM Without Priors: Navigate to brave://settings/wallet** and ensure your Default cryptocurrency wallet is set to Brave Wallet (Prefer extensions) if you are using an extension like MetaMask. Temporarily disable any conflicting extensions…	**Other Pipelines**: Please provide a detailed description along with explicit steps to reproduce (S2R), expected behavior (EB), observed behavior (OB)...	...Please make sure you aren't testing with multiple wallet extensions installed. If you do want Brave Wallet though in that case you can set your default wallet to Brave Wallet **in brave://settings/wallet**. [65]

TABLE VIII: Comparison of Generated and Ground-Truth No-Code Fixes

more practical assessment by considering functional correctness and alignment with successful resolutions. Future research should therefore move beyond purely lexical or embedding-based metrics and adopt evaluation methods that better reflect utility in software maintenance contexts.

Call for Standardized Evaluation Frameworks: The lack of a unified evaluation framework for no-code fix generation remains a critical gap. Our study shows the need for standardized benchmarks that combine invalid subclassification accuracy with no-code fix quality, enabling consistent comparisons across models and methodologies. Such frameworks should include both automated metrics and structured evaluation protocols so future studies can reliably measure progress and generalize findings across datasets and project environments.

Call for Manual Evaluation of No-Code Fixes: Beyond automated metrics, our results highlight the importance of manual assessment. Readability, clarity, and practical usefulness are central to whether a no-code fix is actionable for end users. Future research should incorporate human-in-the-loop evaluation, including qualitative analysis and user studies, to better capture these dimensions. Adding such human-centered criteria can improve the reliability and real-world applicability of AI-generated no-code fixes.

C. Implications for Practitioners

Our findings suggest that practitioners should start with project-grounded retrieval when reliable internal data is available. Teams with issue histories, internal wikis, support playbooks, or documentation can use these sources before relying on broader web search. In our benchmark, *RAG + Gemini* achieves the strongest overall subclassification performance in Table V, including strong results for *External System & Dependency Issues*, where past closures and product documentation often provide more useful evidence than generic web results.

Use Context Engineering in Stages: Agentic web search is still valuable for no-code fix generation. As shown in Table VII, *Agentic Web Search + Gemini* achieves the highest aggregate Judge success rate. However, practitioners should first anchor the model in trusted project-specific corpora, then add live web search when the issue depends on upstream systems, external dependencies, or time-sensitive facts. Weaker models may degrade when tool traces introduce noise rather than useful constraints.

Adapt Pipelines to Local Workflows: Company deployments require adaptation to local data and governance constraints. Embedding stores, access control, retention policies, prompt versions, schemas, and internal sources all affect results. Practitioners should repeat key ablations in their own environment, including taxonomy priors on/off, different retrieval corpora, temperature settings, and repeated-run aggregation. For auditability, retrieved URLs and chunk identifiers should be logged for both RAG and agentic web search outputs.

Measure Operational Value: Production value should be measured through operational signals, not aggregate F1 alone. Teams can track when high-confidence subclassification reroutes work away from engineering, accepted no-code fixes remove follow-up cycles, or drafts reduce time-to-first-response. These counts can be converted into estimated effort savings and compared against API cost, retrieval maintenance, and human review. Rare subclasses should be monitored separately, since weighted averages can hide weak minority-class performance.

VI. Threats to Validity

Threats to the validity of our findings exist, primarily stemming from the nature of conducting an empirical study using archival data from a software development repository. We organize this discussion according to the three standard categories of validity: external, internal, and construct validity.

A. External Validity

Our study is conducted exclusively on the Brave Browser repository, which may limit the generalizability of the findings. Different projects may exhibit varying development practices, issue-reporting behaviors, and maintenance dynamics, potentially leading to different outcomes.

The dataset is derived from a repository with rich historical data and an actively maintained wiki. Projects lacking such data sources may present different invalid subclass distributions and model performance. To mitigate this threat, we evaluate configurations both with and without repository-specific data.

B. Internal Validity

Due to the opaque training processes of LLMs, we cannot guarantee that the models have not been exposed to portions of our benchmark during pretraining. Such exposure may bias the results and artificially affect the performance.

The use of RAG introduces the risk of incorporating information that became available after the original bug report was created. To mitigate this threat, we restrict retrieval to snapshots of wiki content that are time-aligned with the corresponding test sample and bug reports that have been resolved before the creation of the corresponding test sample.

Agentic web search performs data retrieval through a search platform that gives web results at that moment [66], hence the data that would be retrieved by web search would be different. Therefore, it may have been an impediment to the performance of agentic web search that the retrieval and issue dates do not match. We excluded the original issue link from the search results to prevent data leakage. Furthermore, despite these rigorous date-limiting and exclusion strategies across our pipelines, inadvertent data leakage might still occur. Information regarding a bug might have propagated through secondary channels, mirrored repositories, or broader developer discussions that bypassed our temporal filters.

LLM outputs are inherently stochastic under default temperature settings. This variability may lead to inconsistent results across runs. To keep experimental robustness, we perform three independent executions per sample and base our analysis on the aggregated outputs. However, during the subclassification task, if all three runs produce different subclasses, we employ an arbitrary tie-breaker by defaulting to the result of the first run. While pragmatic for automated evaluation, this deterministic but unweighted selection lacks a statistical foundation and could marginally affect the final distribution.

C. Construct Validity

The invalid bug report taxonomy is defined based on root cause analysis; however, it represents only one possible classification scheme. Alternative taxonomies may yield different categorizations and affect the interpretation of the results.

The validity of no-code fixes is inferred from issue discussions rather than verified through controlled reproduction in a sandbox environment. In order to diminish the validity of the no-code fixes, we kept attention on observable signals such as user confirmations like reactions and validation comments or maintainer actions (e.g., issue closure). The BERTScore metric, which measures semantic similarity, may overestimate the alignment between generated and reference fixes, leading to overly optimistic evaluations [67].

VII. Conclusion

This study introduces and evaluates an automated framework for subclassifying invalid bug reports based on root causes and generating corresponding no-code fixes. For subclassification, RAG achieves the highest overall performance with 0.66 weighted F1, slightly outperforming vanilla LLMs at 0.65 and agentic web search at 0.64. At the subclass level, RAG performs best on Non-reproducibility with 0.85 F1 and on External System & Dependency Issues with 0.67 F1, while agentic web search shows relatively better performance on Faulty Configuration with up to 0.32 F1. Feature Request and Question remain consistently high across all pipelines, reaching up to 0.81 and 0.79 F1 respectively, whereas Wrong Version remains the most challenging subclass across all approaches with scores between 0.00 and 0.29.

For no-code fix generation, agentic web search achieves the highest overall success rate at 68.9%, outperforming vanilla LLMs at 63.1%. At the subclass level, RAG performs best for Working as Designed with up to 88.2%, while agentic web search achieves the highest performance for Question at 74.4% and External System & Dependency Issues at 68.3%. Wrong Version remains challenging across all pipelines, with success rates ranging from 22.2% to 44.4%.

For future work, we aim to evaluate the proposed framework across a broader range of repositories, including industrial and closed-source projects, to assess generalizability. Improving performance for challenging subclasses, such as wrong version, remains a key direction. Additionally, integrating real-time pipelines for continuous bug report processing and developing user-facing tools and dashboards can further enhance the practical applicability of automated no-code fix systems.

REFERENCES

[1] H. Krasner, "The cost of poor software quality in the us: A 2022 report," Consortium for Information & Software Quality (CISQ), Tech. Rep., Dec. 2022, accessed 2025-11-07. [Online]. Available: https://www.it-cisq.org/wp-content/uploads/sites/6/2022/11/CPSQ-Report-Nov-22-2.pdf

[2] Y. Fan, X. Xia, D. Lo, and A. E. Hassan, "Chaff from the wheat: Characterizing and determining valid bug reports," *IEEE Transactions on Software Engineering*, vol. 46, no. 5, pp. 495–525, 2020.

[3] M. Laiq, N. bin Ali, J. Börstler, and E. Engström, "A data-driven approach for understanding invalid bug reports: An industrial case study," *Information and Software Technology*, vol. 164, p. 107305, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584923001593

[4] J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?" in *Proceedings of the 28th international conference on Software engineering*, 2006, pp. 361–370.

[5] K. Herzig, S. Just, and A. Zeller, "It's not a bug, it's a feature: How misclassification impacts bug prediction," in *2013 35th International Conference on Software Engineering (ICSE)*, 2013, pp. 392–401.

[6] M. Laiq, N. b. Ali, J. Böstler, and E. Engström, "Early identification of invalid bug reports in industrial settings – a case study," in *Product-Focused Software Process Improvement*, D. Taibi, M. Kuhrmann, T. Mikkonen, J. Klünder, and P. Abrahamsson, Eds. Cham: Springer International Publishing, 2022, pp. 497–507.

[7] J. Sun, "Why are bug reports invalid?" in *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*. IEEE, 2011, pp. 407–410.

[8] Y. Su, P. Luarn, Y.-S. Lee, and S.-J. Yen, "Creating an invalid defect classification model using text mining on server development," *Journal of Systems and Software*, vol. 125, pp. 197–206, 2017.

[9] S. Panichella, G. Canfora, and A. Di Sorbo, ""won't we fix this issue?" qualitative characterization and automated identification of wontfix issues on github," *Information and Software Technology*, vol. 139, p. 106665, 2021.

[10] U. B. Torun, M. T. Demircan, M. F. Gön, and E. Tüzün, "Past, present, and future of bug tracking in the generative ai era," *ACM Transactions on Software Engineering and Methodology*, 2026. [Online]. Available: https://doi.org/10.1145/3806655

[11] G. Yang, J. Ji, and J. Kim, "Enhanced bug priority prediction via priority-sensitive long short-term memory–attention mechanism," *Applied Sciences*, vol. 15, no. 2, p. 633, 2025.

[12] J. He, L. Xu, Y. Fan, Z. Xu, M. Yan, and Y. Lei, "Deep learning based valid bug reports determination and explanation," in *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*, 2020, pp. 184–194. [Online]. Available: https://doi.org/10.1109/ISSRE5003.2020.00026

[13] Z. Li, M. Pan, Y. Pei, T. Zhang, L. Wang, and X. Li, "Deeplabel: Automated issue classification for issue tracking systems," in *Proceedings of the 13th Asia-Pacific Symposium on Internetware*, 2022, pp. 231–241.

[14] M. Laiq, N. bin Ali, J. Börstler, and E. Engström, "A comparative analysis of ml techniques for bug report classification," *Journal of Systems and Software*, vol. 227, p. 112457, 2025. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121225001256

[15] X. Du, Z. Liu, C. Li, X. Ma, Y. Li, and X. Wang, "Llm-brc: A large language model-based bug report classification framework," *Software Quality Journal*, vol. 32, no. 3, pp. 985–1005, 2024.

[16] E. Dinç and E. Tüzün, "Judge the votes: A system to classify bug reports and give suggestions," in *Proceedings of the 2nd ACM International Conference on AI-powered Software (AIWare '25)*, 2025.

[17] X. Li, J. Jin, G. Dong, H. Qian, Y. Wu, J.-R. Wen, Y. Zhu, and Z. Dou, "Webthinker: Empowering large reasoning models with deep research capability," 2025. [Online]. Available: https://arxiv.org/abs/2504.21776

[18] T. Zhang, V. Kishore, F. Wu, K. Q. Weinberger, and Y. Artzi, "Bertscore: Evaluating text generation with bert," *arXiv preprint arXiv:1904.09675*, 2019.

[19] N. Pandey, D. Sanyal, A. Hudait, and A. Sen, "Automated classification of software issue reports using machine learning techniques: an empirical study," *Innovations in Systems and Software Engineering*, vol. 13, 12 2017.

[20] N. Limsettho, H. Hata, A. Monden, and K. Matsumoto, "Unsupervised bug report categorization using clustering and labeling algorithm," *International Journal of Software Engineering and Knowledge Engineering*, vol. 26, pp. 1027–1053, 09 2016.

[21] I. Chawla and S. Singh, "Automated labeling of issue reports using semi supervised approach," *Journal of Computational Methods in Sciences and Engineering*, vol. 18, pp. 1–15, 01 2018.

[22] H. Qin and X. Sun, "Classifying bug reports into bugs and non-bugs using lstm," in *Proceedings of the 10th Asia-Pacific Symposium on Internetware*, 2018, pp. 1–4.

[23] X. Ye, F. Fang, J. Wu, R. Bunescu, and C. Liu, "Bug report classification using lstm architecture for more accurate software defect locating," in *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*. IEEE, 2018, pp. 1438–1445.

[24] Q. Meng and J. Visser, "Which bug reports are valid and why? using the bert transformer to classify bug reports and explain their validity," in *Proceedings of the 4th European Symposium on Software Engineering (ESSE 2023)*, 2023, pp. 52–60.

[25] A. Budhiraja, K. Dutta, M. Shrivastava, and R. Reddy, "Towards word embeddings for improved duplicate bug report retrieval in software repositories," in *Proceedings of the 2018 ACM SIGIR International Conference on the Theory of Information Retrieval (ICTIR '18)*, 2018, pp. 167–170.

[26] J. Deshmukh, K. Annervaz, S. Podder, S. Sengupta, and N. Dubash, "Towards accurate duplicate bug retrieval using deep learning techniques," in *2017 IEEE International conference on software maintenance and evolution (ICSME)*. IEEE, 2017, pp. 115–124.

[27] E. F. Akyol, M. Dedeler, and E. Tüzün, "Improbr: Bug report improver agent using llms," in *Proceedings of the International Conference on Evaluation and Assessment in Software Engineering (EASE '26)*. ACM, 2026.

[28] Z. Wang, J. Li, M. Ma, Z. Li, Y. Kang, C. Zhang, C. Bansal, M. Chintalapati, S. Rajmohan, Q. Lin *et al.*, "Large language models can provide accurate and interpretable incident triage," in *2024 IEEE 35th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2024, pp. 523–534.

[29] A. T. Kalai, O. Nachum, S. S. Vempala, and E. Zhang, "Why language models hallucinate," *arXiv preprint arXiv:2509.04664*, 2025.

[30] K. Kiashemshaki, A. Khosravani, A. Hosseinpour, and A. Akhavan, "Automated bug triaging using instruction-tuned large language models," *arXiv preprint arXiv:2508.21156*, 2025.

[31] Y. Tao, Y. Qin, and Y. Liu, "Retrieval-augmented code generation: A survey with focus on repository-level approaches," *arXiv preprint arXiv:2510.04905*, 2025.

[32] M. Baqar, "Rag4tickets: Ai-powered ticket resolution via retrieval-augmented generation on jira and github data," *arXiv preprint arXiv:2510.08667*, 2025.

[33] K. Ren, "Credibility assessment of fabricated bug reports via large language models: A study on detecting fake software issues," 2025.

[34] W. Zhang, Y. Li, Y. Bei, J. Luo, G. Wan, L. Yang, C. Xie, Y. Yang, W.-C. Huang, C. Miao, H. P. Zou, X. Luo, Y. Zhao, Y. Chen, C. Chan, P. Zhou, X. Zhang, C. Zhang, J. Shang, M. Zhang, Y. Song, I. King, and P. S. Yu, "From web search towards agentic deep research: Incentivizing search with reasoning agents," 2025. [Online]. Available: https://arxiv.org/abs/2506.18959

[35] J. Liu, Y. Li, C. Zhang, J. Li, A. Chen, K. Ji, W. Cheng, Z. Wu, C. Du, Q. Xu, J. Song, Z. Zhu, W. Chen, P. Zhao, and J. He, "Webexplorer: Explore and evolve for training long-horizon web agents," 2025. [Online]. Available: https://arxiv.org/abs/2509.06501

[36] X. Pang, S. Tang, R. Ye, Y. Du, Y. Du, and S. Chen, "Browsemaster: Towards scalable web browsing via tool-augmented programmatic agent pair," 2025. [Online]. Available: https://arxiv.org/abs/2508.09129

[37] H. Ruan, Y. Zhang, and A. Roychoudhury, "Specrover: Code intent extraction via llms," in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE, 2025, pp. 963–974.

[38] Y. Zhang, H. Ruan, Z. Fan, and A. Roychoudhury, "Autocoderover: Autonomous program improvement," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 1592–1604.

[39] M. Sudeep, "Revolutionizing customer service: The impact of large language models on chatbot performance," *INTERNATIONAL JOURNAL*, vol. 10, no. 5, pp. 721–730, 2024.

[40] H. Wang, X. Peng, H. Cheng, Y. Huang, M. Gong, C. Yang, Y. Liu, and J. Lin, "Ecom-bench: Can llm agent resolve real-world e-commerce customer support issues?" in *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing: Industry Track*, 2025, pp. 276–284.

[41] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan, "Swe-bench: Can language models resolve real-world github issues?" *arXiv preprint arXiv:2310.06770*, 2023.

[42] N. Jain, K. Han, A. Gu, W.-D. Li, F. Yan, T. Zhang, S. Wang, A. Solar-Lezama, K. Sen, and I. Stoica, "Livecodebench: Holistic and contamination free evaluation of large language models for code," *arXiv preprint arXiv:2403.07974*, 2024.

[43] A. Patil, "Gitbugs: Bug reports for duplicate detection, retrieval augmented generation, triage, and more," *arXiv e-prints*, pp. arXiv–2504, 2025.
[44] "Icon is missing from brave notification ads after macos upgrade," https://github.com/brave/brave-browser/issues/26323, accessed: 2026.
[45] "Full screen mode on mac make tabs and url section disappear," https://github.com/brave/brave-browser/issues/35808, accessed: 2026.
[46] "Development version request," https://github.com/brave/brave-browser/issues/21405, accessed: 2026.
[47] "Crash in brave ads," https://github.com/brave/brave-browser/issues/34144, accessed: 2026.
[48] "What format does brave use to store date/time for ads," https://github.com/brave/brave-browser/issues/27157, accessed: 2026.
[49] "Update without restarting," https://github.com/brave/brave-browser/issues/20778, accessed: 2026.
[50] "Possible display bug on recovery phrase screen," https://github.com/brave/brave-browser/issues/20796, accessed: 2026.
[51] b. contributors, "Labels · brave/bravebrowser," https://github.com/brave/brave-browser/labels, 2026, accessed: 2026-01-05.
[52] W. G. Cochran, *Sampling Techniques*. Hoboken: John Wiley & Sons, 2007.
[53] J. R. Landis and G. G. Koch, "The measurement of observer agreement for categorical data," *biometrics*, pp. 159–174, 1977.
[54] B. Kucuk, I. Hanhan, and E. Tuzun, "Characterizing duplicate bugs: Perceptions of practitioners and an empirical analysis," *Journal of Software: Evolution and Process*, vol. 35, no. 11, p. e2446, 2023.
[55] A. Zeng, X. Lv, Z. Hou, Z. Du, Q. Zheng, B. Chen, D. Yin, C. Ge, C. Huang, C. Xie *et al.*, "Glm-5: from vibe coding to agentic engineering," *arXiv preprint arXiv:2602.15763*, 2026.
[56] MiniMax AI, "Minimax m2.7: Early echoes of self-evolution," https://huggingface.co/MiniMaxAI/MiniMax-M2.7, 2026, technical report and model release.
[57] K. Team, T. Bai, Y. Bai, Y. Bao, S. Cai, Y. Cao, Y. Charles, H. Che, C. Chen, G. Chen *et al.*, "Kimi k2. 5: Visual agentic intelligence," *arXiv preprint arXiv:2602.02276*, 2026.
[58] Serper. (2026) Serper: The world's fastest and cheapest Google Search API. Accessed: 2026-04-21. [Online]. Available: https://serper.dev/
[59] OpenRouter, "Openrouter api reference," https://openrouter.ai/docs/api/reference/overview, 2026, accessed: 2026-04-23.
[60] J. Niimi, "A simple ensemble strategy for llm inference: Towards more stable text classification," in *International Conference on Applications of Natural Language to Information Systems*. Springer, 2025, pp. 189–199.
[61] M. Zhang, O. Press, W. Merrill, A. Liu, and N. A. Smith, "How language model hallucinations can snowball," in *Proceedings of the 41st International Conference on Machine Learning*, ser. ICML'24. JMLR.org, 2024.
[62] J. Wu, J. Zhu, Y. Liu, M. Xu, and Y. Jin, "Agentic reasoning: A streamlined framework for enhancing llm reasoning with agentic tools," *arXiv preprint arXiv:2502.04644*, 2025. [Online]. Available: https://arxiv.org/html/2502.04644v2
[63] "Brave issue 34522, wrong version no-code fix example," https://github.com/brave/brave-browser/issues/34522#issuecomment-1827025260, accessed: 2026.
[64] "Brave issue 31299, working as designed no-code fix example," https://github.com/brave/brave-browser/issues/31299#issuecomment-1608043966, accessed: 2026.
[65] "Brave issue 23741, faulty configuration no-code fix example," https://github.com/brave/brave-browser/issues/23741#issuecomment-1169167446, accessed: 2026.
[66] Google Search Help, "Why your google search results differ from others," https://support.google.com/websearch/answer/12412910?hl=en, 2025, accessed: 2026-04-24.
[67] M. Hanna and O. Bojar, "A fine-grained analysis of BERTScore," in *Proceedings of the Sixth Conference on Machine Translation*, L. Barrault, O. Bojar, F. Bougares, R. Chatterjee, M. R. Costa-jussa, C. Federmann, M. Fishel, A. Fraser, M. Freitag, Y. Graham, R. Grundkiewicz, P. Guzman, B. Haddow, M. Huck, A. J. Yepes, P. Koehn, T. Kocmi, A. Martins, M. Morishita, and C. Monz, Eds. Online: Association for Computational Linguistics, Nov. 2021, pp. 507–517. [Online]. Available: https://aclanthology.org/2021.wmt-1.59/