On Developers' Perceptions of Detection and Resolution of Dependability Issues in Closed Source Projects

João Correia
PUC-Rio
Rio de Janeiro, RJ, Brazil
jcorreia@inf.puc-rio.br

Daniel Coutinho
PUC-Rio
Rio de Janeiro, RJ, Brazil
dcoutinho@inf.puc-rio.br

Alessandro Garcia
PUC-Rio
Rio de Janeiro, RJ, Brazil
afgarcia@inf.puc-rio.br

Rafael de Mello
UFRJ
Rio de Janeiro, RJ, Brazil
rafaelmello@ic.ufrj.br

Wesley K. G. Assunção
NC State University
Raleigh, NC, USA
wguezas@ncsu.edu

Caio Barbosa
PUC-Rio
Rio de Janeiro, RJ, Brazil
cbarbosa@inf.puc-rio.br

Nelio Cacho
UFRN
Natal, RN, Brazil
neliocacho@dimap.ufrn.br

Thais Batista
UFRN
Natal, RN, Brazil
thais.batista@ufrn.br

Abstract

As software systems grow in size and complexity, ensuring their dependability becomes a challenging task. Dependability attributes such as reliability, security, integrity, and maintainability are essential to software projects. However, developers of closed source projects often struggle to address dependability-related issues efficiently. Large Language Models (LLMs) have shown promise in automating tasks like code generation, but their effectiveness in automatically detecting and addressing dependability issues in closed-source project settings remains uncertain. It is also unclear how developers perceive these fixes. In this paper, we introduce a multi-agent LLM pipeline that automatically identifies and fixes a wide range of dependability issues. The pipeline is composed of four distinct stages, each handled by an agent responsible for detection, validation, fixing, and judgment. To evaluate the approach, we conducted a study with eight experienced developers who reviewed a quite diverse set of fixes generated for three closed-source industrial Python systems currently in production. The results show a strong agreement on the relevance of the detected issues, with an average rating of 4.47 out of 5, and general satisfaction with the proposed fixes, which received an average score of 4.35. A qualitative analysis of developer feedback revealed that the pipeline is effective in addressing problems in all dependability attributes. We observed differences in developers' evaluations of issue relevance and the criteria for approval during a code review, reflecting the inherently subjective nature of dependability assessment in industrial settings. These results suggest our pipeline shows valuable potential in automatically identifying and repairing dependability issues while: (1) reinforcing that developer involvement remains essential to resolve subjective disagreements, and (2) demonstrating that our pipeline can supplement recent LLM-based work on dependability problem detection and resolution, which is currently limited to open-source projects and a narrow set of problem types.

Keywords

Software Dependability, Large Language Models; Program Repair

1 Introduction

Many closed source software projects need to be dependable. Software dependability is a non-functional requirement defined as the ability to deliver service that can justifiably be trusted [2]. Software dependability is a non-functional requirement that encompasses various attributes like reliability, security, integrity, and maintainability [2, 11]. Addressing these attributes is a critical part [21, 24, 26, 28] of closed source software projects, but detecting and resolving dependability problems based on natural-language issue reports by developers are challenging [19]. The increasing complexity of these projects makes the manual detection and resolution of a wide range of dependability issues a resource-intensive activity [8]. Due to this complexity, recent work has observed that developers tend to neglect some of these attributes, such as integrity and reliability [5, 18, 19], when maintaining their software systems.

The advent of Large Language Models (LLMs) has shown significant promise in automating various development tasks, from question answering and code synthesis to automated repair [4, 12, 24]. This has led to the exploration of advanced techniques, such as *LLM-as-a-Judge* where models assess the quality and correctness of generated code, moving beyond traditional metrics that often fail to capture semantic correctness [22]. However, they are not focused on addressing various dependability-related issues in closed source projects. In fact, recent research on LLMs focuses on detecting and resolving specific dependability-related faults, such as maintainability's code smells (e.g., [26]) and security vulnerabilities (e.g., [28, 30, 32]), strictly based on resource-intensive source code analyses and restricted to open source projects only.

These existing works do not assess how LLMs can depart from real dependability problems reported upfront as natural-language issue descriptions by closed-source project developers. Despite advancements, much of the existing work on dependability [7, 31] does not fully represent the multifaceted nature of dependability in complex, industrial software systems. Consequently, there is a limited understanding of how effective these automated approaches

are in detecting and fixing genuine dependability issues in issue-driven, real-world contexts and, just as importantly, how developers perceive and value their outputs. Without human-centered evaluation, it is difficult to assess whether such tools can be meaningfully integrated into existing development workflows.

In this paper, we aim to fill this gap by proposing and evaluating a multi-agent LLM pipeline designed to automatically detect and fix software dependability issues. Our approach employs a pipeline composed of four sequential stages: *Issue Detector*, *Issue Validator*, *Code Fixer*, and *Code Judger*. This pipeline analyzes files in closed source software repositories to identify problems and generate actionable fixes. To assess its practical relevance, we conducted a study involving eight experienced developers from private development settings in Brazil. These developers evaluated a range of selected solutions generated by our pipeline for three closed-source, Python systems currently in production with stringent dependability needs.

Our qualitative analysis reveals that the pipeline is capable of detecting issues across all dependability dimensions based on issue reports by developers. However, it performs particularly strongly in integrity, reliability, and maintainability, the three most frequently detected dimensions. Some examples of detection include missing serializer-level validation, improper error handling, unlocalized exception messages, design-level vulnerabilities and domain-specific smells. In particular, an intriguing finding was that security and confidentiality concerns, such as exposed tokens and personally identifiable information leaks, were also identified, though less frequently, it may reflect the inherent difficulty of detecting context-dependent vulnerabilities through code-level static analysis alone.

The quantitative results indicate a positive perception among developers, who acknowledged both the relevance of the identified issues (mean Likert score of 4.47 out of 5) and the effectiveness of the proposed fixes (mean Likert score of 4.35 out of 5) in the analyzed closed source projects. The patch approval dimension also received favorable ratings (4.16), indicating that most fixes were considered acceptable for merging through standard code review practices. Regarding developer feedback, positive assessments were associated with categories such as *fix effectiveness and reliability*, as well as *bug recognition and relevance*, indicating a strong alignment between the pipeline's outputs and established development practices. When disagreements occurred, they typically stemmed from differing assessments of issue severity, contextual misalignment between the proposed fix and system-specific design decisions, or uncertainty about the necessity of certain validation checks — patterns captured under the Incomplete or *misaligned fixes*, *low severity issue*, and *uncertain parameter validation* categories.

These findings reinforce that, while the pipeline demonstrates strong potential for automatically identifying and repairing dependability issues, human oversight remains essential to resolve subjective disagreements and ensure that automated recommendations align with the architectural and operational context of each system. In summary, the main contributions of this paper are:

(i) the design and implementation of a novel, multi-agent pipeline that leverages multiple LLMs to automatically detect and resolve software dependability issues in closed source projects;
(ii) a comprehensive, human-centered evaluation of the pipeline's effectiveness and relevance, conducted with professional developers on three real-world industrial systems;
(iii) a mixed-method analysis of developer perceptions, combining quantitative ratings and qualitative feedback to identify the factors influencing the acceptance of automatically generated fixes;
(iv) a characterization of the dependability dimensions that the pipeline most effectively addresses, offering insights into the current strengths and limitations of such a solution; and,
(v) a set of empirical findings suggesting that our pipeline can complement recently proposed LLM-based approaches [7, 26, 28, 30–32], strictly assessed in open-source projects and which focus only on a narrow set of dependability faults based on source code analyses.

2 Related Work

In this section, we discuss related work across three main subsections: the use of LLMs for code tasks and as judges, the benchmarks developed to evaluate such capabilities, and the application of LLMs to dependability problems.

2.1 LLM for Code and LLM-as-a-judge

Large Language Models (LLMs) have recently been employed across several software engineering tasks, including code synthesis, program repair, and test generation, and more recently as judges to automatically evaluate the correctness and quality of generated outputs. The Standard evaluation metrics for such activities, such as *CodeBLEU* [22], rely on human-written references to compute similarity and often fail to capture semantic correctness. To overcome these limitations, researchers have explored the *LLM-as-a-Judge* paradigm, where models assess responses directly based on quality and correctness [29].

The *LLM-as-a-Judge* paradigm has been explored in the last years. Tripathi et al. [25] investigate prompting strategies for this paradigm, comparing pairwise evaluation — where the judge selects the better of two responses — against pointwise evaluation — where each response is scored independently. Their findings show that pairwise protocols are more vulnerable to manipulation via distractor features, with preferences flipping in approximately 35% of cases compared to only 9% for pointwise scoring, suggesting that the choice of feedback protocol should be guided by evaluation objectives and dataset characteristics. Complementarily, McAleese et al. [16] incorporated structured critiques into the judgment process to surface and mitigate biases that arise when LLMs evaluate their own or peers' outputs.

2.2 Benchmarks to Evaluate LLMs for Code

Despite recent advances, a key limitation remains in the benchmarks used to evaluate LLMs on coding tasks and as judges. Most available benchmarks suffer from limited diversity, questionable quality, and insufficient scale. Widely adopted datasets such as *HumanEval* [3, 20] and the *Mostly Basic Python Problems Dataset* [1, 9], while broadly used, fall short of capturing the complexity of real-world software engineering scenarios. Du et al. [7] reinforce this concern by evaluating LLMs on code generation at the class level, highlighting the need for more realistic and complex assessments. Efforts to address this gap have led to more challenging benchmarks. Jimenez et al. [13] introduced *SWE-bench*, encompassing real-world GitHub issues for assessing LLM capabilities beyond

simple code generation. However, Ramos et al. [21] underscore the need for careful benchmark selection and robust evaluation metrics to reliably assess LLM capabilities in automated program repair, as models may exhibit significant memorization of widely used benchmarks, while newer models show more limited leakage. Finally, such benchmarks remain restricted to open-source repositories and rely exclusively on unit test approval for correctness verification, leaving questions of industrial applicability.

A parallel line of work focuses specifically on benchmarks for the *LLM-as-a-Judge* paradigm. Specialized approaches such as *LLM4-PatchCorrect* [31] and *CriticGPT* [16] attempt to tailor models for code judging, but remain constrained in coverage and robustness. Jiang et al. [12] substantially extend prior work by introducing *CodeJudgeBench*, a large-scale benchmark covering three fundamental coding tasks: *code generation*, *code repair*, and *unit test generation*. Nevertheless, these benchmarks share the same fundamental limitation: they evaluate LLM capabilities in isolation from real development workflows, leaving the question of how well such judges perform in industrial, dependability-critical settings unanswered.

2.3 LLM for Dependability Problems

Previous studies have investigated the use of LLMs for dependability problems. Using automated program repair (APR), Sobania et al. [24] showed that ChatGPT's bug-fixing performance is competitive with deep learning-based approaches, outperforming traditional repair techniques. Meng et al. [17] systematically analyzed the strengths and weaknesses of complex LLM-based agent systems on SWE-bench Verified, evidencing that reasoning ability, symbol-level fault localization, and patch verification are the key differentiators among top performers. On the other hand, Xia et al. [27] challenge the assumption that such complexity is necessary, showing that a simple three-phase pipeline (localize, repair, and validate) without complex tooling can match or outperform sophisticated agents on SWE-bench Lite at a fraction of the cost.

Regarding *maintainability*, Wu et al. [26] combine LLMs with static analysis tools (e.g., *iSMELL*) to improve code-smell detection and refactoring. For *reliability and robustness*, Wu et al. [14] propose XRFix, an LLM-based repair framework that targets performance bugs in Extended Reality (XR) applications, combining static analysis for detection with prompt-based LLM repair across different granularity levels, achieving a fix rate of 67.3% on a curated dataset of 104 real-world bugs from 23 open-source XR projects. Zhang et al. [28] propose *Seeker*, a multi-agent framework that targets exception safety by locating weak exception paths and suggesting repairs. In *security*, Zhou et al. [30] survey rapid progress in vulnerability detection and patching, and Zibaeirad et al. [32] introduce *VulnLLMEval* to probe model limits on real-world repositories.

These advances push dependability detection and fixing forward, although most evaluations still rely on open-source benchmarks rather than being embedded in industrial closed-source workflows. We address this gap by targeting real industrial closed source systems and treating *dependability* as the core, all-encompassing objective. Our agent pipeline goes beyond judging code: it *detects*, *validates*, and *fixes* dependability issues, with developers reviewing the outcomes. We also employ an *LLM-as-a-Judge* to automatically evaluate fixes' quality, what is an approach well-studied for general code correctness but still uncommon in dependability pipelines [12, 29]. This shifts the focus from isolated benchmark evaluation to LLM-based support for issue-level detection and resolution in real development practices. In particular, because we do not evaluate on public benchmark datasets, we mitigate the threat of data contamination and memorization that recalling benchmark answers as highlighted by Ramos et al. [21]. The goal is not only to assess LLM capability, but also to demonstrate the feasibility of an all-encompassing and deployable pipeline that proactively detects and repairs dependability problems based on issue descriptions.

3 Study Design

In this section, we describe the study design. We begin by introducing the proposed pipeline designed to automatically detect and address dependability issues in software systems. Next, we outline our research questions that will guide the evaluation of the pipeline's effectiveness and relevance. We then explain the data collection process, the manual assessments performed by developers, and finally, the methods used for data analysis.

3.1 Architecture for Automatic Detection and Resolution of Dependability Issues

We designed and implemented a multi-agent pipeline that takes the path to a software repository as input and automatically analyzes it to detect and propose solutions for issues that may compromise software dependability. The overall architecture of the pipeline is depicted in Figure 1.

The proposed pipeline, as depicted in Figure 1, consists of four interconnected agents, each relying on the output of the previous one. The first agent identifies potential dependability issues in the source files. The second agent then validates these detected issues, filtering out false positives and unnecessary noise. Notably, we employed models from different companies for detection/validation and fix/judgment in an effort to mitigate bias. The third agent generates fixes in the target code. Finally, the last agent evaluates whether the fixes effectively address the identified issues.

One of the primary challenges in developing a pipeline for detecting and proposing code changes is defining the prompts for each agent. This process involves several iterations of prompt engineering. We applied well-established strategies from the prompt engineering literature. These strategies included clearly defining the role of the model, specifying the task, and providing one-shot examples in the expected JSON output format [10, 15, 23]. These approaches offer several advantages. By defining the role, we can focus the model's perspective, reducing irrelevant or overly creative responses. Specifying the task clearly enhances clarity and ensures that the output remains reliable and focused. Additionally, by providing examples of the expected output, we minimize variability in the responses, which improves the model's precision and machine-readability of the results.

It is important to note that during our architectural design phase, GPT-5 was released, and we investigated the use of this new model in our study. This exploration highlighted the importance of employing effective prompting techniques. Unlike previous models, the GPT-5 API does not allow for strict output constraints or temperature tuning, which means there is less control at the parameter level. As a result, strategies such as role specification and providing

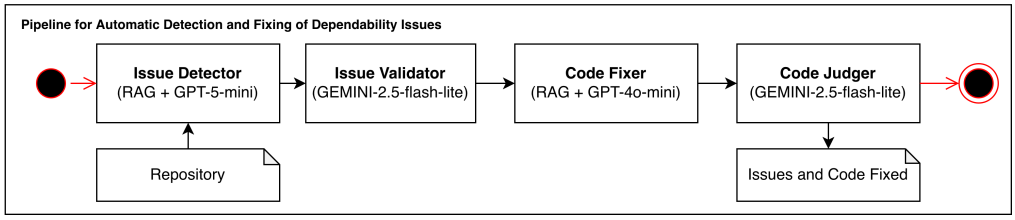

Figure 1: Pipeline for automatic detection and resolution of dependability issues.

format examples became essential for effectively guiding the responses. In the following subsections, we discuss each agent and its characteristics. The full prompts presented in following subsections are available in our supplementary material [6].

3.1.1 Issue Detector. The processing begins with a repository folder, which is the input to the analysis. The first step involves an agent we call *Issue Detector*, implemented as a combination of Retrieval-Augmented Generation (RAG) mechanism and the GPT-5-mini model. We applied the Cognita framework to implement our RAG mechanism, locally hosted (Local-RAG). Initially, we created a collection in the Cognita framework. This collection basically consists of computing the embeddings for a set of documents (all the files in the analyzed repository) to input as context in queries to LLM models. Once the collection is created, Cognita enables the execution of queries to the models through an API, automatically forwarding the context retrieved from the collection. So, one file in the repository at a time, we performed queries (i.e., through the Local-Rag) to identify possible dependability issues. The output of this process is a set of potential dependability issues with the issue title, description, file, impact and the affected source code.

The Frame 1 shows the structure used for the prompt at this stage. In our experiments with various model configurations, we discovered that GPT-5-mini, although slower in response than GPT-4o-mini, consistently yielded more accurate and comprehensive detections. The extra time taken during the detection step was justified by the improvements in the quality of identified issues.

Issue Detector

Role You are a senior Python developer specialized in highly dependable systems [...]

Task Perform a thorough code review of the code file below, focused on detect dependability issues [...]

File: {file content}

Output: Return a JSON with:

```
{"title": "Handle missing config file ...",
 "location": "settings.py",
 "issue": "Fails if file absent ...",
 "impact": "Startup crash"}
```

Context {retrieved content}

Frame 1: Issue Detector Prompt Template

3.1.2 Issue Validator. The second stage in our pipeline is the *Issue Validator*, which is powered by *Gemini-2.5-Flash-Lite*. We chose a model from a different organization to avoid possible biases in conducting detection and validation with models coming from the same architecture family. The *Issue Validator* received as input the issue title, description, location, and affected code. The Frame 2 presents the prompt structure delivered to the model. Using this prompt, we asked the model to evaluate whether the issue was truly related to dependability. The role of the *Issue Validator* was to check the *Issue Detector*'s findings and to filter out false positives. By doing so, it ensures that only issues with strong evidence and clear dependability implications progress to the fixing stage.

Issue Validator

Role You are a senior Python developer specialized in highly dependable systems. [...]

Task Critically evaluate whether a reported issue is a valid dependability issue. Reject style-only concerns [...] or negligible impacts. [...]

Issue: title, description, location and affected code.

Output: Return a JSON with:

```
{"description": "This issue is a valid...",
 "classification": "Confirm"}
```

Frame 2: Issue Validator Prompt Template

3.1.3 Code Fixer. After the *Issue Validator* completed double validation, it passed the identified dependability issues to the *Code Fixer*. The information was structured as follows: issue title, description, and affected code. By using our Local-RAG and the GPT-4o-mini model, we tasked the agent with generating a solution for the specified problem in the affected code. The prompt design for this stage is illustrated in Frame 3. We evaluated various configurations, including GPT-5-mini, and determined that GPT-4o-mini provided the best balance between latency and quality for this phase. The objective of the Code Fixer is to minimize "creative" modifications and to leverage repository context to maintain coding conventions while enhancing reliability, integrity, and maintainability.

Code Fixer

Role You are a senior Python developer specialized in highly dependable systems [...]

Task Think critically, step by step: evaluate the provided dependability issue and [...] fix the code [...]

Reported Issue: title, description and affected code.

Output: A JSON with two fields [...]

```
{"description": "The create method ...",
 "fixed_code": "def create(self,..."}
```

Context {retrieved content}

Frame 3: Code Fixer Prompt Template

3.1.4 Code Judger. The final stage, the *Code Judger* was also powered by Gemini-2.5-Flash-Lite. At this point, we already have a candidate solution for the identified issues, so we instruct the agent to assess whether this candidate adequately addresses the problems. The judger evaluates the issue title, description, the affected code, and the proposed fix, ultimately issuing a final verdict of either "Confirm" or "Reject," the solution, accompanied by a brief rationale that highlights the specific changes made to the code. A condensed version of the judger's instructions can be found in Frame 4. Finally, any issue that successfully passes through all four stages is considered a recommendation for improving the system's dependability, while those that do not are discarded.

Code Judger

Role You are a senior Python developer specialized in highly dependable systems. [...]

Task Critically evaluate step by step whether the reported issue was completely fixed [...]

Reported Issue: title, description and affected code.

Proposed Solution: fixed code.

Output: Return a JSON with:

```
{"description": "This issue is a valid...",
 "classification": "Confirm"}
```

Frame 4: Code Judger Prompt Template

3.2 Research Questions

To evaluate the proposed pipeline, we defined two research questions that address both the technical effectiveness of our pipeline and its practical relevance from the perspective of developers.

RQ_1: *How effective is a multi-agent LLM pipeline at improving software dependability?* In this first research question, we aim to explore the effectiveness of our multi-agent pipeline in detecting dependability issues. We measure effectiveness in two dimensions: the extent to which the detected issues cover the main dependability aspects, and how experienced developers rate these detected issues. First, we analyze issues' descriptions generated by the pipeline and categorize them according to the nature of the dependability concerns. Each issue is mapped to attributes such as reliability, maintainability, confidentiality, safety, availability, and integrity. This analysis provides a clearer understanding of the pipeline's detection capabilities, highlighting both its strengths and limitations. Next, we assess effectiveness from the perspective of experienced developers in three dimensions: relevance, adequacy of the proposed solution, and acceptance of the suggested code changes. Through this evaluation, we examine whether the issues highlighted by the pipeline are perceived as contextually relevant, whether the proposed fixes are seen as effective resolutions to the problems, and whether the resulting code would be acceptable in a review process.

RQ_2: *How do developers assess the quality and relevance of issues and fixes generated by the pipeline?* We pose this question to identify the technical factors that either strengthen or weaken the mergeability and actionability of the pipeline's outputs in practice. Our goal is to understand developers' perceptions of these outputs. While structured evaluations offer a quantitative perspective, they often do not capture practitioners' views on the usefulness, adequacy, and trustworthiness of identified issues and proposed solutions. By collecting qualitative feedback from developers who maintain code, we can gain insights into whether the pipeline's contributions align with real-world development practices. This approach can also highlight situations in which our pipeline's outputs are more likely to be integrated into existing workflows.

3.3 Data Collection

Our data collection process begins with the identification of target projects for dependability analysis (see Figure 2). We prioritize projects that allow us to validate dependability issues through expert code evaluation. To accomplish this, we collaborate with private development organizations based in Brazil. We have identified and established three dependability-intensive software systems with these partners. All of these systems are written in Python and are currently in production for government institutions in Brazil. Additional information about these systems can be found in Table 1.

Table 1: Target Systems by Type and Domain.

System	Type	Domain
S_1	REST/API	Back-end for an Advanced Process Control Tool in industry.
S_2	REST/API	Back-end for a Process Improvement Tool integrated with a Government System.
S_3	REST/API	Back-end for a Conversational Agent-based Tool integrated with a Government System.

Withing these systems we have a total of 285 code files available for analysis: 183 files from system S_1, 74 files from system S_2, and 28 files from system S_3. Given the computational and financial costs associated with LLM inference, we restricted our evaluation to a subsample of the available source code files. To ensure this analysis was representative, we applied random sampling to select a subset of files from each system. The sample size (n) for each system was calculated using a 95% confidence level, which allowed us to obtain a statistically valid subset of files. We sampled 64 files from S_1, which is approximately 35% of its total files. From S_2, we selected 43 files, representing about 58% of its total. Finally, from S_3 we considered 22 files (78.6%). Altogether, this process resulted in a sample of 129 files, nearly half (45%) of all available files across the systems. The second and third columns in Table 2 summarize the total files in the systems and the sampled ones, respectively.

Table 2: Issue Distribution by System and Pipeline Stage.

System	Files	Sampled Files	Issue Detector (# issues detected)	Issue Validator (# issues validated)	Code Fixer (# issues fixed)	Code Judger (# issues validated)
S_1	183	64	107	97	97	70
S_2	74	43	82	79	79	54
S_3	28	22	41	39	39	25

We then executed the pipeline on the sampled files. For S_1, the pipeline identified 70 issues across 21 of 64 files (~33% of the sampled files). For S_2, it detected 54 issues across 15 of 43 files (~35%), and for S_3, 25 issues across 8 of 22 files (~36%). This stage yielded 149 issues along with their candidate solutions. Table 2 summarizes the number of issues identified for each system on each pipeline stage.

For our data collection purposes, the pipeline generates a structured spreadsheet where each row captures the complete details of an analyzed issue. The columns include: *title*, *location*, *affected code*, *description*, *impact*, *label*, *validator justification*, *validator classification*, *fixed code*, *fix description*, *judger justification*, and *judger*

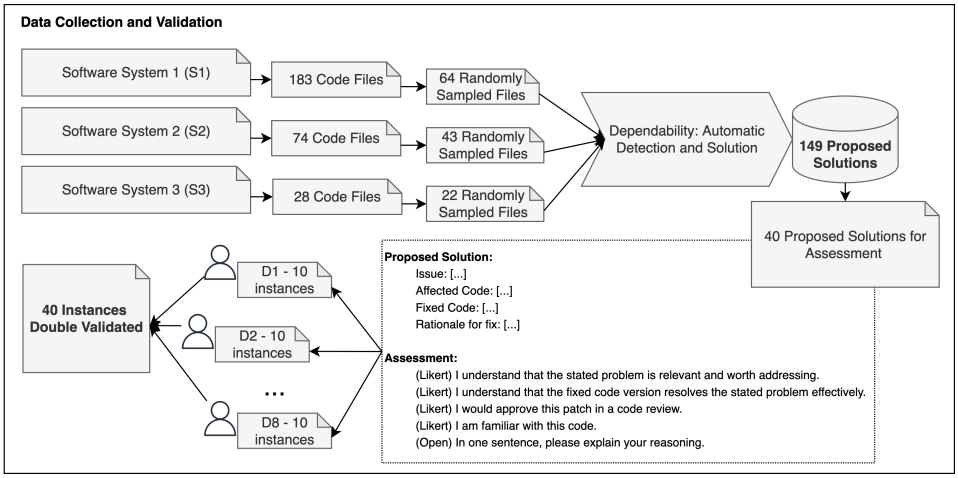

Figure 2: Data collection and validation.

classification. Together, these fields form a detailed record enabling, for example, automated pull request generation with actionable, evidence-based suggestions to improve code reliability, integrity, and maintainability.

Following, we validate the issues and solutions with the developers who worked on these systems. To avoid overwhelming them with the need to manually validate all solutions, we randomly selected a subset of 40 solutions, which constitutes approximately 27% of the dataset. This selection aimed to strike a balance between comprehensive coverage and practicality, ensuring sufficient diversity while keeping the validation workload manageable.

3.4 Manual Validation Process

As shown in Figure 2, each of the 40 randomly selected solutions was double-validated by independent developers. We engaged eight developers, D_1 to D_8, with each one validating 10 solutions. In practice, this meant that every solution was reviewed by two different developers, ensuring cross-checking and reducing individual bias. In total, this process produced 80 individual evaluations (40 solutions $\times$ 2 reviewers). Table 3 shows more details about the developers' background.

Table 3: Developers Background.

Developer	Education Level	Experience (years)	Experience in Company	Languages
D1	Master's	7	4	Python, Javascript, Java, C#
D2	Master's	10	2	Python
D3	Undergraduate	5	3	Python, Javascript, C/C++
D4	Undergraduate	2	1	Python, Matlab, C, C++
D5	Undergraduate	6	4	C, C++, C#, Java, Python, SQL, JavaScript, TypeScript
D6	Undergraduate	1	1	Python
D7	Master's	15	5	Ruby, JavaScript, Python
D8	Undergraduate	7	1	Python, Javascript

We designed a structured evaluation form. For each proposed issue and solution, developers were presented with:

(1) **Contextual information** – including the description of the identified issue, the affected code fragment, the automatically generated fixed code, and a brief rationale explaining the fix.
(2) **Closed questions** – in these four questions, we employed a Likert scale with five points, ranging from *(1) Strongly Disagree* to *(5) Strongly Agree*. The statements were:
   - "I understand that the stated problem is relevant and worth addressing."
   - "I understand that the fixed code version resolves the stated problem effectively."
   - "I would approve this patch in a code review."
   - "I am familiar with the code and context of this solution."
(3) **An open-ended field** (optional) – provided after each issue/solution, where developers were encouraged to justify their ratings in one sentence."

Each question in the evaluation form was carefully designed to capture a specific dimension of the developers' assessment of the proposed fixes. The first statement, "*I understand that the stated problem is relevant and worth addressing*", aims to assess whether the evaluator agrees that the identified issue is meaningful and requires attention, thereby validating the significance of the problem detected by the pipeline. The second statement, "*I understand that the fixed code version resolves the stated problem effectively*", focuses on the correctness and adequacy of the proposed solution, prompting reviewers to consider whether the automated fix addresses the root cause of the problem without introducing regressions. The third statement, "*I would approve this patch in a code review*", serves as a holistic judgment, integrating multiple aspects such as problem relevance, fix adequacy, and code quality into a realistic scenario of code review. Finally, the fourth statement, "I am familiar with the code and context of this solution", was included to capture potential variation in confidence and contextual understanding, enabling us to interpret the other answers in light of the evaluator's familiarity with the underlying codebase. By integrating structured Likert-scale ratings with open-ended textual justifications, the validation process captured both quantifiable assessments and contextual insights into developer reasoning, enabling a more comprehensive understanding of their evaluation criteria.

3.5 Data Analysis

To answer our research questions, we conducted a multi-step analysis of the issues and fixes produced by the pipeline. We examined how effectively the pipeline identified and addressed dependability concerns and how developers assessed the quality and relevance of

those outputs. We combined quantitative and qualitative analysis to provide a view of the pipeline's technical and practical impact.

3.5.1 Quantitative Analysis. We started by converting the Likert scale evaluations into numerical scores ranging from *(1) Strongly Disagree* to *(5) Strongly Agree*. For each of the 40 solutions, we aggregated the responses from the two evaluators by calculating their mean values. This resulted in a single score for each solution across four evaluation dimensions: relevance of the problem, effectiveness of the fix, likelihood of approval in a code review, and familiarity with the code. Aiming at identifying patterns of agreement and disagreement between reviewers, we plotted the data and visually inspected any divergences. We reviewed cases where the two evaluators had notable discrepancies in their assessments, as these instances indicate solutions that received mixed judgments regarding their dependability or appropriateness.

3.5.2 Qualitative Analysis. Initially, the issue descriptions were systematically analyzed using an open coding approach. Each description was carefully examined, and relevant codes were assigned to capture the nature and impact of the identified problem. Based on these codes, we further mapped each issue to one dependability dimension, such as integrity, reliability, maintainability, or security. In this way, we provide a structured view of the types of concerns addressed by the pipeline. This process enabled us to assess not only the technical relevance of the detected issues but also the range of dependability attributes covered. To further understand the pipeline's output characteristics, we computed the frequency of issues associated with each dimension, offering insight into the distribution of dependability concerns detected across the systems.

Aditionally, the open-ended justifications provided by developers were also systematically analyzed using an open coding approach. Each comment was coded to capture the underlying rationale supporting or rejecting the pipeline's proposed issue or fix. These codes were then grouped into broader categories through axial coding, distinguishing between positive rationales (supporting the relevance and adequacy of the solution) and negative rationales (highlighting reasons for rejection or criticism). Positive codes encompassed, for example, explicit mentions of **improved validation**, **strengthened security**, or **reliable error handling**. Negative codes often reflected concerns about **incomplete fixes**, l**ow-severity issues**, or **misalignment with project context**. By combining this categorical analysis with the quantitative Likert-scale ratings, we obtained a more comprehensive view of developer evaluations, combining measurable agreement levels with nuanced reasoning behind acceptance or rejection.

4 Results and Discussions

In general, the use of the pipeline to address dependability issues produced predominantly positive results. In the three closed-source systems analyzed, the developers recognized the relevance of the identified problems and the effectiveness of the fixes. The experts mainly agreed on the recommendations as clear improvements in security, data integrity, and system reliability. When disagreements occurred, they were more about contextual assumptions than technical validity. In general, our findings suggest that the pipeline offers valuable recommendations based on dependability, although human oversight and contextual knowledge remains crucial.

4.1 Effectiveness to Detect Dependability Issues

Improving software dependability is a complex challenge that involves not only generating technically sound and context-aware solutions, but also identifying critical issues. Our first research question (RQ_1) investigates whether a multi-agent pipeline can effectively contribute to this goal by automatically identifying problems that impact key dependability attributes.

To address RQ_1, we conducted a qualitative analysis of the pipeline's output to understand the dimensions of dependability associated with the detected issues. As detailed in the Study Design section (Section 3), we employed an open coding process on the textual descriptions of the identified issues, followed by a categorization process that linked each issue to a specific dependability dimension (e.g., reliability, maintainability). The results of this analysis are presented in Table 4, which displays the distribution of the identified issues across dependability dimensions, the number of occurrences out of 40 issues, with examples from the coding process.

Table 4: Frequency of Detected Dependability Issues

Category	# of Issues	Relevant Codes
Integrity	13	missing serializer-level validation; nullable constraint mismatch; conflicting model constraints; …
Reliability	10	KeyError due to missing request context; brittle dependency on DRF runtime; no rollback on DB failure; …
Maintainability	8	missing language localization in exceptions; undesirable side effects on import; unhandled log level values; …
Security	5	unauthenticated user assignment risk; refresh token exposed in API; unsafe filename interpolation; …
Confidentiality	3	sensitive user fields exposed in nested serializer; log exposes user email; PII risk in API response; …
Availability	1	missing DB connection closure; potential connection leak; resource exhaustion over time; …

The most frequent dependability dimension was *integrity*, with 13 detected issues, followed by *reliability*, with 10. This distribution reflects the pipeline's strength in identifying problems related to data consistency, input validation, and predictable system behavior—concerns that are often explicitly manifested in the code. Typical examples include missing serializer-level checks, constraint mismatches, and improper error handling. *Maintainability* accounted for 8 issues, such as unlocalized exception messages, log inconsistencies, and side effects, which are generally detectable through structural or syntactic patterns. Although less common, the pipeline also flagged *security* (5 issues) and *confidentiality* (3 issues) problems, including exposed tokens and Personal Identification Information leaks. *Availability* was the least represented dimension, with only one issue identified.

The results indicate that the pipeline has the potential to detect issues that significantly impact all aspects of dependability, although it identifies fewer problems in some categories. This lower frequency is expected, as concerns such as security often require contextual understanding that static code representations may not fully capture. Additionally, the absence of availability issues likely reflects the inherent limitations of static analysis in identifying problems that generally arise under dynamic runtime conditions, such as resource leaks or concurrency bottlenecks. Overall, the findings show that the pipeline effectively addresses a wide range of dependability concerns, especially in the areas of integrity, reliability, and maintainability. However, there are opportunities to improve its coverage of more contextual or runtime dimensions, such as availability and security.

To further address RQ_1, we conducted human-centered evaluations in which expert developers reviewed the pipeline's outputs. Table 5 presents the average Likert-scale ratings given by these evaluators on four dimensions: relevance of the issue, effectiveness of the fix, approval of the patch, and familiarity with the code. Overall, the results indicate a generally positive evaluation, suggesting

that the pipeline is capable of producing outputs that developers find meaningful, accurate, and worthy of review.

Table 5: Mean Ratings and Likert Scale Interpretation

Question	Mean	Likert Interpretation
I understand that the stated problem is relevant and worth addressing.	4.47	Agree to Strongly Agree
I understand that the fixed code version resolves the stated problem effectively.	4.35	Agree to Strongly Agree
I would approve this patch in a code review.	4.16	Agree
I am familiar with this code.	4.00	Agree

We achieved the highest rating on the *relevance of the issue* dimension, with a mean score of 4.47. This value reflects a response trend between agree and strongly agree for the statement: "*I understand that the stated problem is relevant and worth addressing*". Figure 3 shows the individual ratings ($R1$ and $R2$) assigned by each pair of evaluators across the 40 assessed issues, with consistently high agreement. This result suggests that developers systematically recognized the issues identified by the pipeline as meaningful and worthy of attention across the evaluated systems. These findings highlight the ability of the pipeline to detect dependability concerns that align with professional judgment and established notions of relevance in real-world software engineering contexts.

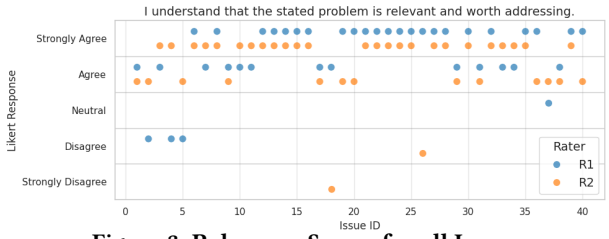

Figure 3: Relevance Scores for all Issues.

The second-highest mean score, 4.35, corresponds to the *effectiveness of the fix* dimension. This indicates a consistent trend toward agreement or strong agreement with the statement: "*I understand that the version of the fixed code effectively solves the stated problem*". As shown in the scatter plot for this dimension (Figure 4), ratings were generally favorable across reviewers. These results suggest that the proposed fixes were perceived as both technically adequate and contextually appropriate, reflecting the pipeline's capacity to generate meaningful improvements.

Figure 4: Fixing Scores for all Issues.

The statement "*I would approve this patch in a code review*" received a mean score of 4.16, which still falls within a clearly positive range. This dimension, depicted in the corresponding scatter plot (Figure 5), shows that most evaluators expressed agreement, indicating a general willingness to endorse the fixes through standard review practices. These results reflect not only confidence in the technical correctness of the generated code but also its practical acceptability from the perspective of experienced developers engaged in real-world development workflows.

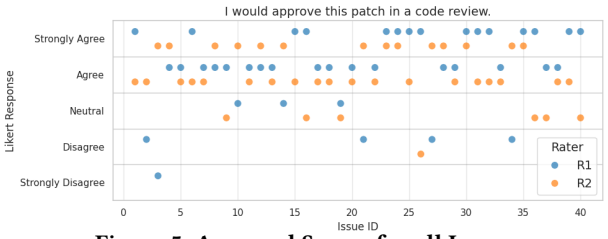

Figure 5: Approval Scores for all Issues.

Finally, as a control dimension, we assessed the *familiarity of developers* with the specific code under review. This dimension received a mean score of 4.00, which also falls within the agree range. The scatter plot for this dimension (Figure 6) shows slightly more variation among individual ratings, likely due to the fact that, although all evaluators were actively involved in the overall project, not all had contributed to the specific modules under review. Nonetheless, the overall score suggests a sufficient level of confidence in understanding the code, thereby reinforcing the credibility and contextual grounding of the evaluators' assessments across the other dimensions.

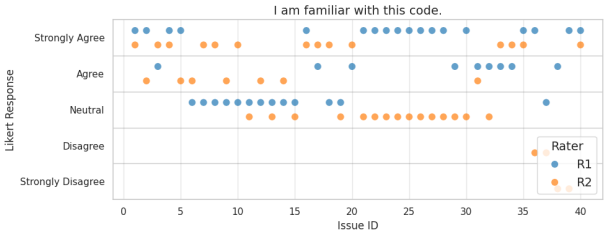

Figure 6: Familiarity with Code Scores for all Issues.

The findings indicate that the pipeline generates valuable outputs: developers consistently rated the flagged issues as relevant, the fixes as effective, and the patches as reviewable. However, these ratings mainly reflect the perceived quality of the identified items, rather than the overall coverage of the dependability space. In other words, positive ratings do not determine which types of concerns the pipeline is most or least sensitive to, nor do they indicate its recall across various dimensions.

To address this limitation, we analyzed the pipeline's detection profile through our coding analysis (see Table 4). This analysis reveals that the pipeline is more effective at detecting code-local integrity, reliability, and maintainability issues, while it shows comparatively weaker coverage of context- and runtime-dependent categories, such as availability and security. This distinction highlights the strengths and limitations of the pipeline and suggests opportunities for expanding its coverage to include more contextual and dynamic behaviors.

4.2 Quality of Generated Fixes

In the second research question (RQ_2, we aimed to understand how expert developers qualitatively assess the fixes generated by the pipeline. While previous analyses primarily used structured Likert-scale ratings, this section delves into the open feedback provided by evaluators, offering deeper insight into their perceptions of the proposed solutions. Table 6 presents the resulting categories, organized by whether the comments supported (Positive) or rejected (Negative) the proposed solutions, along with descriptions and examples of representative codes.

Table 6: Open Coding Categories Derived from Expert Feedback on Proposed Solutions

Suggestion Acceptance	Category	Description	Representative Codes
Positive	Fix Security and Privacy	Strengthens confidentiality, prevents leaks, and improves overall system security.	critical security flaw; removes PII from logs; fix ensures password complexity
Positive	Fix Validation and Data Integrity	Ensures input correctness, schema consistency, and clearer error feedback.	serializer validation improves feedback; data integrity risk; PositiveIntegerField enforces constraints
Positive	Fix Effectiveness and Reliability	Delivers efficient fixes that solve problems and enhance reliability.	simple and effective fix; correct rollback usage; improved observability
Negative	Incomplete or Misaligned Fixes	Represents fixes seen as partial, generic, or contextually unsuitable.	incomplete solution; generic fix criticized; fix lacks context
Negative	Low Severity Issue	Covers low-priority concerns or fixes adding little practical benefit.	minor issue; not serious issue; issue unlikely due to current design
Positive	Bug Recognition and Relevance	Highlights expert agreement on the existence and relevance of bugs.	valid issue acknowledged; bug acknowledged; context-aware bug
Negative	Uncertain Paremeter Validation	Questions the necessity or redundancy of certain validation checks.	manual validation needed in some cases; parameter assumed present; unnecessary extra checks
Positive	Fix Approval	Indicates expert acceptance of fixes, sometimes with conditions.	approval without reservations; approval with suggestion; testing needed before approval
Negative	Security and Robustness Concerns	Points to lingering risks, leaks, or added maintenance burden.	exception message removed; requires future maintenance

4.2.1 Agreement on Experts Evaluations. In several cases, experts consistently recognized both the relevance of the identified issues and the adequacy of the proposed fixes. These instances illustrate situations where the pipeline's suggestions aligned well with established development practices and expert expectations.

A representative example was the case of the soft delete mechanism (Issue 40). The model contained a *deleted_at* field, which implied an intended strategy for soft deletion, but lacked the necessary supporting methods to enforce it. The default delete() operation still triggered hard deletion, creating the risk of inconsistent data handling if developers alternated between calling .delete() and manually setting *deleted_at*. Experts agreed that this represented a relevant dependability issue, as it could compromise data retention policies and introducing inconsistencies in the state of the database.

The proposed fix introduced a *soft_delete*() method, replacing the default *delete*() to prevent hard deletions, and added a *DocumentQuerySet* with a helper method to filter active records. Experts consistently evaluated this fix as effective and aligned with expected practices. The open coding categories most associated with this consensus were Fix Validation and Data Integrity (ensuring consistency in how deletions are applied) and Fix Effectiveness and Reliability (a solution that directly addressed the identified risk).

This pattern of agreement highlights that the pipeline was particularly successful in cases where the identified problem corresponded to well-understood dependability concerns (e.g., consistency and data integrity) and where the proposed fix provided a clear, contextually appropriate mechanism. In such cases, expert evaluations converged, reinforcing the potential of automated pipelines to deliver fixes that meet developers' expectations.

4.2.2 Disagreements on Experts Evaluations. Although most of the evaluations demonstrated convergence between experts in terms of the relevance of the detected issues and the suitability of the proposed solutions, several cases revealed disagreements. These divergences highlight how contextual knowledge, the design of each system, and differing interpretations shape developers' judgments.

First, some disagreements arise from contextual misalignment. For example, in the multilingual exception handling case (Issue 2), one evaluator viewed the lack of structured serialization and bilingual support as a minor issue, categorizing the fix as an improvement in error messages. In contrast, the second expert argued that the solution failed to understand the original design choice of using English as the default language, consequently introducing unnecessary complexity. This situation falls under the category of "Incomplete or Misaligned Fixes," where the pipeline's proposals, although technically valid, were deemed contextually unsuitable.

Second, disagreements were also linked to different severity assessments. For instance, in serializers, the access to the request attribute assumes that the attribute exists (Issue 3); one developer minimized the likelihood of failure in practice, considering it an edge case. The second insisted that explicitly checking for request presence is important for defensive programming. Similarly, the *deleted_at* field manager (Issue 5) was seen by one expert as unlikely to cause failures because the field was guaranteed in the current model. In contrast, the other considered the check a relevant safeguard. These cases align with the "Low Severity Issue" category, where the same flaw is acknowledged but weighted differently depending on the perceived risk.

Other disagreements derived from uncertainties in parameter validation. In Issue 18, one attribute is not assigned during object creation, leading to a problem because the attribute is read-only. One developer emphasized that leaving the attribute unset introduced inconsistencies, while another argued that the context was missing since the attribute was always populated upstream. Similarly, in Issue 28, related to batch ID validation, opinions diverged between those who valued stricter upfront checks and those who considered them redundant or misapplied. These instances illustrate the "Uncertain Parameter Validation" category, where experts questioned whether additional validations were necessary.

Finally, disagreements were also tied to different interpretations of security and robustness concerns. In the refresh token exposure case (Issue 34), one evaluator endorsed the fix as a necessary security measure, whereas the other criticized it as incomplete, since removing the token from the payload did not fully resolve the underlying vulnerability. Comparable situations appeared in database-related fixes (Issues 22, 27, 28), where one evaluator focused on improved reliability through validation or resource management, while the other emphasized potential side-effects, such as premature connection pool closure or suppressed errors. These patterns correspond to the "Security and Robustness Concerns" and "Incomplete or Misaligned Fixes" categories.

Overall, the disagreements show that expert evaluations were not simply binary judgments of correctness but were mediated by trade-offs among security, robustness, contextual relevance, and practical severity. The open coding results (Table 6) capture this diversity: positive categories (e.g., Fix Validation and Data Integrity,

Fix Effectiveness and Reliability) often clashed with negative categories (e.g., Incomplete or Misaligned Fixes, Low Severity Issue), depending on how experts weighed contextual constraints and long-term dependability. These findings underscore that the acceptance of automated fixes depends not only on their technical soundness but also on how they align with existing practices, assumptions, and risk perceptions in the target systems.

5 Threats to Validity

This section discusses potential threats to the validity of our study and the measures we took to mitigate them.

Internal Validity. Our random sampling strategy for selecting files from each system may introduce bias if the sampled files are not representative of the codebase characteristics. To mitigate this threat, we used statistical sampling with a 95% confidence level, ensuring a representative subset across all three systems. Additionally, the developers who evaluated the issues were all from the same organization and had varying levels of familiarity with the codebases under analysis. While developers actively work on the systems they evaluated, they were not necessarily reviewing code they had personally authored or maintained. This factor could introduce bias, particularly in how confidently they assessed the issues. To account for this, we included a familiarity dimension in our evaluation form as a control measure, allowing us to measure the extent to which each developer felt confident in their experience.

External Validity. The study focused exclusively on Python systems, which may limit the generalizability of our findings to other programming languages. The results provide evidence of the pipeline's effectiveness within this context. However, additional validation is needed to determine whether similar results are maintained across other technology stacks and scenarios. We validated 40 issues out of 149 (27%). This dataset may not capture the full spectrum of issues' characteristics. While this sample size was chosen to balance comprehensiveness with practical constraints, it may restrict our findings. Also, all evaluated systems came from three closed-source organizations in Brazil. The organizational culture, practices, and quality standards may have influenced developer evaluations in ways that may not apply to other organizations.

Construct Validity. The definition of dependability may not align with how other researchers or practitioners conceptualize this construct. We addressed this by grounding our definition in established literature and mapping detected issues to dependability dimensions (reliability, security, integrity, maintainability, availability, confidentiality). The Likert-scale questions we used to assess developer perceptions may not fully capture the nuanced judgments in code review scenarios. We complemented these quantitative measures with open-ended qualitative feedback to provide insights into developer reasoning. Moreover, our open coding approach for categorizing issues into dependability dimensions may have introduced subjectivity. We mitigated this by having multiple researchers review the coding scheme and ensuring consistency.

Conclusion Validity. With only 40 evaluated solutions and 8 developers, our study may have limited statistical power to detect small but meaningful differences in developer perceptions. We addressed this by focusing on effective sizes and confidence intervals instead of relying on statistical tests. Finally, we conducted multiple analyses across different dimensions and evaluation criteria, which increases the risk of Type I errors. We acknowledge this limitation and interpret our findings conservatively, focusing on patterns that are consistent across multiple dimensions.

6 Conclusion

This paper presented a comprehensive evaluation of a multi-agent LLM pipeline designed to automatically detect and fix software dependability issues in industrial systems. Through a study involving eight experienced developers from Brazilian industry settings, we demonstrated that automated approaches can effectively identify meaningful dependability concerns and generate actionable solutions that align with professional development practices.

Our evaluation revealed consistently positive developer perceptions across all evaluation dimensions. Developers strongly agreed that the identified problems were relevant and worth addressing (mean score of 4.47 out of 5), and they recognized the effectiveness of the proposed fixes (4.35 out of 5). The high approval rating for code review scenarios (4.16 out of 5) validates the practical applicability of the pipeline's outputs in real-world development workflows. The qualitative analysis showed that the pipeline is particularly effective at detecting problems related to *integrity*, *reliability*, and *maintainability*, while also identifying *security* and *confidentiality* concerns. While most evaluations of issues and solutions showed agreement, disagreements came from four main areas: incomplete fixes, low severity issues, uncertain parameter validation, and security concerns. These disagreements demonstrate that human oversight and domain knowledge are essential for validating automated solutions.

Our work makes three key contributions: first, we demonstrated that a multi-agent LLM pipeline can effectively identify and fix real dependability issues in industrial systems; second, through a study with 8 professional developers, we revealed how practitioners evaluate automated outputs in practice; and third, we showed that the pipeline performs robustly across all dependability attributes. These findings carry practical implications, indicating that automated dependability analysis tools can be integrated into existing development workflows—particularly for surfacing issues related to data integrity, system reliability, and code maintainability. Future work should explore alternative pipeline architectures, evaluate effectiveness across diverse technology stacks and application domains, and investigate methods to deepen contextual understanding for more nuanced dependability concerns.

Artifact Availability

To support reproducibility, the artifacts utilized and generated in this paper are publicly available at [6]. The replication package includes the datasets and full pipeline source code, organized as a notebook for easy execution and replication. Confidential data, including proprietary code artifacts, has been suppressed.

Acknowledgments

This work was partially supported by INES.IA (National Institute of Science and Technology for Software Engineering Based on and for Artificial Intelligence) www.ines.org.br, CNPq grant 408817/2024-0. Also, CNPq: 310391/2025-3, 140770/2021-6, 141180/2021-8, 140771/2021-2, 315711/2020-5, 141276/2020-7, 141054/2019-0. FAPERJ: 200.510/2023, 211.033/2019, 202.621/2019. Finally, CAPES/Proex: 88887.373933/2019-00.

References

[1] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. Program Synthesis with Large Language Models. *arXiv preprint arXiv:2108.07732* (2021).

[2] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. 2004. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing* 1, 1 (2004), 11–33. doi:10.1109/TDSC.2004.2

[3] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating Large Language Models Trained on Code. *arXiv preprint arXiv:2107.03374* (2021).

[4] João Correia, Daniel Coutinho, Marco Castelluccio, Caio Barbosa, Rafael de Mello, Anita Sarma, Alessandro Garcia, Marco Gerosa, and Igor Steinmacher. 2026. A Comparison of Conversational Models and Humans in Answering Technical Questions: the Firefox Case. In *Proceedings of the IEEE/ACM 48th International Conference on Software Engineering (ICSE '26)*. IEEE.

[5] João Correia, Daniel Coutinho, Alessandro Garcia, Rafael de Mello, Caio Barbosa, Anderson Oliveira, Wesley K. G. Assunção, Juliana Alves Pereira, Igor Steinmacher, Marco Gerosa, Jairo Souza, and Johny Arriel. 2024. On the Investigation of Exception Pull Request Characteristics: Exploring the Apache Ecosystem. In *2024 IEEE International Conference on Source Code Analysis and Manipulation (SCAM)*. 143–153. doi:10.1109/SCAM63643.2024.00023

[6] João Correia, Daniel Coutinho, Alessandro Garcia, Rafael De Mello, Wesley K. G. Assunção, Caio Barbosa, Nélio Cacho, and Thais Batista. 2026. Supplementary Material: Dataset and Promps of Validated Dependability Issues. https://github.com/opus-research/developers_perception_dependability.

[7] Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2024. Evaluating Large Language Models in Class-Level Code Generation. In *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering (ICSE 2024)*. IEEE Computer Society, Lisbon, Portugal, 982–994. doi:10.1145/3597503.3639219

[8] András Földvári and András Pataricza. 2021. Semi-automated model extraction from observations for dependability analysis. In *2021 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 1–6.

[9] Google Research. [n. d.]. MBPP: Mostly Basic Python Problems (dataset and tasks). https://github.com/google-research/google-research/tree/master/mbpp. Accessed: 2025-09-16.

[10] Jia He, Mukund Rungta, David Koleczek, Arshdeep Sekhon, Franklin X Wang, and Sadid Hasan. 2024. Does Prompt Formatting Have Any Impact on LLM Performance? arXiv:2411.10541 [cs.CL] https://arxiv.org/abs/2411.10541

[11] IEEE Systems and Software Engineering Standards Committee. [n. d.]. *IEEE982:2024 – IEEE Standard for Measures of the Software Aspects of Dependability*.

[12] Hongchao Jiang, Yiming Chen, Yushi Cao, Hung-yi Lee, and Robby T. Tan. 2025. CodeJudgeBench: Benchmarking LLM-as-a-Judge for Coding Tasks. *arXiv preprint arXiv:2507.10535* (2025).

[13] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-World GitHub Issues? arXiv:2310.06770 [cs.CL] https://arxiv.org/abs/2310.06770

[14] Wu Jingwen, Hanyang Guo, Hong-Ning Dai, and Xiapu Luo. 2025. XRFix: Exploring Performance Bug Repair of Extended Reality Applications with Large Language Models. doi:10.1145/3744916.3773120

[15] Aobo Kong, Shiwan Zhao, Hao Chen, Qicheng Li, Yong Qin, Ruiqi Sun, Xin Zhou, Enzhi Wang, and Xiaohang Dong. 2024. Better Zero-Shot Reasoning with Role-Play Prompting. arXiv:2308.07702 [cs.CL] https://arxiv.org/abs/2308.07702

[16] Nat McAleese and Maja Trębacz. 2024. Finding GPT-4's mistakes with GPT-4. url=https://openai.com/index/finding-gpt4s-mistakes-with-gpt-4/.

[17] Xiangxin Meng, Zexiong Ma, Pengfei Gao, and Chao Peng. 2025. An Empirical Study on LLM-based Agents for Automated Bug Fixing. arXiv:2411.10213 [cs.SE] https://arxiv.org/abs/2411.10213

[18] Anderson Oliveira, João Correia, Leonardo Sousa, Wesley K. G. Assunção, Daniel Coutinho, Alessandro Garcia, Willian Oizumi, Caio Barbosa, Anderson Uchôa, and Juliana Alves Pereira. 2023. Don't Forget the Exception! : Considering Robustness Changes to Identify Design Problems. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. 417–429. doi:10.1109/MSR59073.2023.00064

[19] Anderson Oliveira, João Correia, Wesley K. G. Assunção, Juliana Alves Pereira, Rafael de Mello, Daniel Coutinho, Caio Barbosa, Paulo Libório, and Alessandro Garcia. 2024. Understanding Developers' Discussions and Perceptions on Non-functional Requirements: The Case of the Spring Ecosystem. *Proc. ACM Softw. Eng.* 1, FSE, Article 24 (July 2024), 22 pages. doi:10.1145/3643750

[20] OpenAI. [n. d.]. HumanEval: Hand-Written Evaluation Set — Code for "Evaluating Large Language Models Trained on Code". https://github.com/openai/human-eval. MIT License. Accessed: 2025-09-16.

[21] Daniel Ramos, Claudia Mamede, Kush Jain, Paulo Canelas, Catarina Gamboa, and Claire Le Goues. 2025. Are Large Language Models Memorizing Bug Benchmarks? arXiv:2411.13323 [cs.SE] https://arxiv.org/abs/2411.13323

[22] Shuo Ren, Daya Guo, and et al. 2020. CodeBLEU: a Method for Automatic Evaluation of Code Synthesis. In *Proceedings of the 2020 International Conference on Learning Representations (ICLR)*.

[23] Connor Shorten, Charles Pierse, Thomas Benjamin Smith, Erika Cardenas, Akanksha Sharma, John Trengrove, and Bob van Luijt. 2024. StructuredRAG: JSON Response Formatting with Large Language Models. arXiv:2408.11061 [cs.CL] https://arxiv.org/abs/2408.11061

[24] Dominik Sobania, Martin Briesch, Carol Hanna, and Justyna Petke. 2023. An Analysis of the Automatic Bug Fixing Performance of ChatGPT. (May 2023), 23–30. doi:10.1109/APR59189.2023.00012

[25] Tuhina Tripathi, Manya Wadhwa, Greg Durrett, and Scott Niekum. 2025. Pairwise or Pointwise? Evaluating Feedback Protocols for Bias in LLM-Based Evaluation. arXiv:2504.14716 [cs.LG] https://arxiv.org/abs/2504.14716

[26] Di Wu, Fangwen Mu, Lin Shi, Zhaoqiang Guo, Kui Liu, Weiguang Zhuang, Yuqi Zhong, and Li Zhang. 2024. iSMELL: Assembling LLMs with Expert Toolsets for Code Smell Detection and Refactoring. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ACM. doi:10.1145/3691620.3695508

[27] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2025. Demystifying LLM-Based Software Engineering Agents. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE037 (June 2025), 24 pages. doi:10.1145/3715754

[28] Xuanming Zhang, Yuxuan Chen, Yuan Yuan, and Minlie Huang. 2025. Towards Exception Safety Code Generation with Intermediate Representation Agents Framework. arXiv:2410.06949 [cs.SE] https://arxiv.org/abs/2410.06949

[29] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. 2023. Judging LLM-as-a-judge with MT-bench and Chatbot Arena. In *Proceedings of the 37th International Conference on Neural Information Processing Systems* (New Orleans, LA, USA) *(NIPS '23)*. Curran Associates Inc., Red Hook, NY, USA, Article 2020, 29 pages.

[30] Xin Zhou, Sicong Cao, Xiaobing Sun, and David Lo. 2025. Large Language Model for Vulnerability Detection and Repair: Literature Review and the Road Ahead. 31 pages. doi:10.1145/3708522

[31] Xin Zhou, Bowen Xu, Kisub Kim, DongGyun Han, Hung Huu Nguyen, Thanh Le-Cong, Junda He, Bach Le, and David Lo. 2024. Leveraging Large Language Model for Automatic Patch Correctness Assessment. *IEEE Transactions on Software Engineering* 50, 11 (2024), 2865–2883. doi:10.1109/TSE.2024.3452252

[32] Arastoo Zibaeirad and Marco Vieira. 2024. VulnLLMEval: A Framework for Evaluating Large Language Models in Software Vulnerability Detection and Patching. arXiv:2409.10756 [cs.SE] https://arxiv.org/abs/2409.10756