Exploring Conversational Agents for Developer Guidance on Software Dependability Issues

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

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

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

Abstract

Software dependability is a critical non-functional requirement that encompasses a system's ability to remain available, reliable, maintainable, robust, and resilient in the presence of faults. Dependability requirements are essential in almost every software project. Exception handling plays a central role in operationalizing these attributes and serves as the means to detect, report and handle failures; yet it remains under-supported by tools and often under-emphasized in developer workflows. In this context, this work empirically investigates recurring exception handling issues in software projects. Then, it further investigates how conversational agents, particularly those powered by Large Language Models (LLMs), can support dependable software development. With this aim, we evaluated the effectiveness of LLM-based assistants in real-world software engineering contexts: (i) answering technical questions of developers, and (ii) automatically detecting dependability-related issues and generating candidate fixes for developer review. To this end, four complementary studies were conducted in the context of open-source and closed-source systems. DevMentorAI, a retrieval-augmented conversational agent, was evaluated through a case study involving Mozilla developers in the PDF.js project. DevMentorAI demonstrated superior or similar performance compared to humans in many scenarios. Then, another comparative study in Mozilla Firefox showed that retrieval-augmented models produce more helpful and comprehensive responses than generic LLMs and human experts. Finally, we designed and evaluated a multi-agent architecture that extends conversational assistance by proactively identifying and repairing dependability issues. The approach was evaluated through case studies in three industrial software projects from different domains, with assessments performed by the projects' own developers. Our findings revealed that LLM-driven agents can generate effective, explainable, and reviewable code changes that align with developers' expectations. When grounded in project-specific knowledge, they can effectively support the identification and resolution of software dependability issues, advancing both research and practice in AI-assisted software engineering. These studies are reported in premier vehicles in software engineering and specific themes. The agent systems were used by industry partners.

Keywords

Software Dependability; Developer Assistance; Large Language Models; Conversational Agents

1 Introduction

Software dependability is a critical concern in modern software engineering. The dependability of a software system is the ability to deliver its services that can justifiably be trusted [1]. This non-functional requirement (NFR) encompasses a system's ability to remain robust, maintainable, and resilient in the presence of failures [26]. The operational and financial consequences of poor dependability are far from negligible. For instance, the 2024 CrowdStrike outage alone caused an estimated $5.4 billion in losses across the 500 largest U.S. companies, with healthcare, banking, and airlines among the hardest hit [32]. Achieving dependability means satisfying one or more of the interrelated attributes: (i) availability (service continuity), (ii) reliability (correct service over time), (iii) safety (avoiding catastrophic faults), (iv) confidentiality (protecting sensitive data), (v) integrity (consistent system state), and (vi) maintainability (cost-effective evolution). These attributes are pursued through four complementary engineering strategies: fault prevention, fault removal, fault forecasting, and fault tolerance [1, 26]. Nowadays, software dependability requirements are essential to satisfy in almost all industrial-strength software projects.

In this vein, the exception-handling mechanism is a principal vehicle for achieving dependability and, more explicitly, *fault tolerance*: it serves as the primary means of achieving program robustness by capturing abnormal states, managing errors, and enabling recovery or controlled degradation [1, 26]. As such, it enables the enactment of dependability attributes at the code level. It shapes recovery (reliability) and graceful degradation (availability), state rollback and resource cleanup (integrity), sanitization of error messages and stack traces (confidentiality), failing safely so that faults do not leave the system in an exploitable state (security), and the clarity of error propagation paths and handling structures (maintainability). Yet decades of research show that designing exception-related code is inherently difficult. Shah *et al.* [37] found that the quality of exception code in industrial systems is often substandard despite its importance. Less-experienced developers struggle to use these mechanisms, leading to the proposal of a dedicated "exception engineer." Poor choices – e.g., empty or generic exception catch blocks – propagate invalid states and produce silent failures. They frequently co-occur with broader architectural problems rather than remaining localized [30].

This exposes a central paradox that motivates the work: *although dependability is among the most important attributes in many projects, the implementation of its sub-attributes – such as robustness and security – is often the least tested and least trustworthy part of the system.* A key reason is the shortage of appropriate automated assistance for *detecting and resolving* these issues. Compounding this, we found that existing tools and techniques for addressing exception handling tend to center on problems that are far from the most frequent in practice [10]. Static analyzers and automated-repair frameworks for exception handling focus on control-flow violations and evident bugs [10]. Recurring activities performed

by developers concerning exception handling, such as refining exception handlers and improving error messages, remain largely unsupported [10]. These reinforce dependability problems in software projects. Another empirical finding was that robustness is referenced far less frequently than maintainability, security, and performance in PRs' discussions [40], remaining a poorly-addressed problem in modern development.

These difficulties were consistently observed in open-source software (OSS) projects, where development is highly distributed and collaborative. Design decisions in these systems emerge through asynchronous channels such as issue trackers and PRs [19]. Incomplete and dispersed documentation, architectural drift, and high contributor turnover make it hard to keep non-functional requirements satisfied. Many contributors in these projects lack a comprehensive understanding of the system's design [23, 40, 41]. As a result, dependability problems gradually accumulate. Prior studies on exceptions [3, 5, 6, 45] classify problems and solutions but rarely analyze the *collaborative processes*, i.e., how developers propose, review and discuss exception-related contributions. This gap limits our understanding of how these practices are socially constructed and technically validated in real collaborative projects.

Recent advances in LLMs and conversational agents offer a promising avenue for delivering the context-aware, timely assistance these settings demand. They can help developers understand complex code, answer technical questions, and even generate or review code changes [2, 28]. Their effectiveness, however, may depend on rich project context, including requirements, coding standards, architectural rationale, and the current state of the codebase [13, 24]. This makes it difficult to produce actionable and safe recommendations [33, 35] and observe how their outputs are perceived and validated by human developers [13]. To be effective, LLM-powered tools must bridge the gap between theoretical capabilities and the practical needs of dependable software development.

In this context, this work investigates how conversational assistants based on LLMs can support dependable software development by addressing three interconnected gaps:

- **G1 – Understanding exception-handling practices in OSS.** Prior research focuses mainly on language mechanisms, defect analysis, or best practices [20, 29, 37], leaving open how exception handling is coordinated through collaborative development, such as PRs, and whether specialized "exception engineers" dominate (or not) this work in practice.
- **G2 – Evaluating project-aware developer assistance.** The general-purpose LLMs show potential for assisting software development tasks [28]. However, their effectiveness may be inconsistent or limited by a lack of project-specific grounding. It may also be prone to hallucination or verbosity. Retrieval-Augmented Generation (RAG) can inject relevant context [24], yet software-engineering evaluations rarely compare against strong human or non-RAG baselines.
- **G3: From information support to actionable code improvements.** Most automated-repair and multi-agent approaches are evaluated on generic bug datasets. They rarely generate traceable, reviewable patches for real-world closed-source systems while preserving developer oversight.

The overarching goal of this work is to promote the strengthening of software dependability through developer support by addressing these three gaps above. To this end, we provide contributions in the following complementary research directions: (i) characterizing how exception-related PRs are coordinated in OSS; (ii) assessing whether conversational agents can assist developers with technical questions; (iii) comparing general-purpose and retrieval-augmented models with human responses; and (iv) exploring how conversational agents can proactively enhance dependability through automated detection and remediation of code issues. These investigations span OSS ecosystems and closed-source systems, enabling comprehensive evaluation across development environments.

Each of these contributions and other ones related to this Doctoral research are reported in papers for premier international [8, 10, 13, 30, 31] and national [15, 17] conferences and for journals [16, 31]. These papers were submitted and published in *top vehicles in the Software Engineering field or reference vehicles of specialized themes associated with our research*, such as ICSE, AIware@FSE, SCAM, MSR, SBES, SBQS and Proceedings of the ACM/FSE journal. The produced agent systems *were actually used in projects of the industry partners. We are also currently working on extensions of our previous studies in partnership with industrial collaborators that observed the aforementioned gaps in their routines.* The replication packages, such as [9, 11, 12], were also made available through our research and reported in the aforementioned publications or in the PhD thesis of the first author [14]. In this present paper, we provide an overview of our mixed-methods approach, our four complementary studies, and the key contributions of the conducted Doctoral research.

2 Related Work

We organize related work into three thematic areas: exception-handling practices (G1), LLM and RAG-based developer assistance (G2), and automated repair of dependability issues (G3).

Exception handling as error recovery in software systems. Exception handling connects internal errors to observable behavior and is a central means to ensure program robustness. Shah *et al.* [37] characterized exception-handling practices in industry through developer interviews, reporting dissatisfaction with language-imposed mechanisms (e.g., exception-handling constructs in Java) and limited organizational support. Given the difficulties of addressing exception handling, they proposed the role of an *exception engineer* in software teams. In follow-up work, they observed that novices improperly use exceptions mainly for debugging while experts integrate them properly into their normal routine flows [38].

Afterwards, Nakshatri *et al.* [29] compared Java practice against the *Effective Java* guidelines [4] and found that developers frequently disregard checked exceptions and favor higher-level exception classes. Sawadpong *et al.* [36] reported that exception-handling code in Eclipse had a defect density roughly three times higher than the overall codebase. Ebert *et al.* [20, 21] classified 220 exception-handling bugs in Eclipse and Tomcat. Their survey of 154 developers revealed that only 27% of organizations had formal practices for implementing exception-handling mechanisms. Cacho *et al.* [5, 6] studied how exception-handling behavior evolves and analyzed the trade-offs between robustness and maintainability. Viviani *et al.* [40] manually classified PR discussions and identified robustness as a frequent but under-prioritized design topic. Our prior work

[10] extends this line by systematically analyzing exception-related PRs in OSS and contrasting them with non-exception PRs to expose their social and procedural dynamics.

LLMs in software engineering. LLMs can generate, summarize, and transform code and natural language. A growing body of work examines their use in software engineering. Dakhel *et al.* [28] evaluated GitHub Copilot's ability to produce correct, efficient, and reproducible solutions, with mixed results relative to human-written code. Sallou *et al.* [35] highlighted concerns around closed-source models, data leakage, and reproducibility. Balfroid *et al.* [2] found that LLM-generated onboarding "code tours" often remained vague or merely echoed documentation. Pudari and Ernst [33] reported that Copilot frequently fails to follow idiomatic best practices. Across these studies, a common thread is that general-purpose models operate without project-specific grounding: they produce plausible but generic output, unaware of a system's conventions, architecture, or rationale. This motivates approaches that supply such context explicitly, which we examine next.

RAG for project-specific support. Retrieval-Augmented Generation (RAG) enhances LLM capabilities by grounding responses in external, contextually relevant sources such as project documentation and code [22, 24, 43]. In software engineering contexts, Ibtasham *et al.* [24] demonstrated that RAG can achieve 70% relevance in release management tasks, while Finsaas and Maksim [22] improved technical support accuracy through relevance feedback and multi-agent collaboration. Applications to large-scale onboarding and developer Q&A, however, remain underexplored. Correia *et al.* [13] conducted the first systematic comparison of RAG-enhanced LLM responses and human answers for newcomer questions in an OSS project (PDF.js), with their in-house system outperforming human responses in 8 of 14 cases, matching in 3, and underperforming in 3; that study was nonetheless limited by its small scale, single evaluator, and narrow project scope. A follow-up study [8] addressed those limitations by comparing responses from a general-purpose LLM, a RAG-enhanced LLM built on an open-source framework (Cognita) [7], and human developers, using 52 real-world questions from Firefox evaluated by eight experienced engineers along helpfulness, comprehensiveness, conciseness, and overall preference. This design isolates RAG's contribution and yields actionable insights into when retrieval is necessary and the context to include.

Automated repair of dependability issues. LLM-based repair has advanced rapidly. Sobania *et al.* [39] report bug-fixing performance competitive with prior deep-learning approaches, while Meng *et al.* [27] and Xia *et al.* [42] debate whether agentic complexity is warranted on SWE-bench. These evaluations share a correctness criterion, namely unit test approval on open-source benchmarks [25], that is blind to the handler refinements and error-message improvements developers perform most often [10], and that carries a memorization risk for widely used datasets [34]. Closest to our setting, Zhang *et al.* [44] target exception safety via a multi-agent framework that detects fragile code, locates weak exception paths, and suggests repairs across open-source repositories. We differ in setting, entry point, and correctness criterion: we target industrial closed-source systems, take issue descriptions rather than code paths as the starting point, and treat developer acceptance under review as the measure of a successful fix.

3 Research Questions

The present paper is structured as a compilation of four studies, each guided by a research question (RQ):

- **RQ1.** *How do PR dynamics and contributor roles differ between exception-PRs and non-exception PRs in OSS?* This question characterizes the social and procedural dynamics of exception-related contributions, how they are reviewed, and the roles contributors assume, establishing where exception work concentrates and how context-aware tools can help. This question is directly addressed by a study reported in a SCAM 2024 paper [10]. This question emerged from our previous exploratory studies performed along the PhD research mainly in collaboration with another PhD student at that time, which are co-authored and reported in a MSR 2023 paper [30] and in a journal/FSE paper (Proceedings of the ACM Journal, a volume dedicated to FSE 2024 articles).
- **RQ2.1.** *To what extent can a project-aware conversational assistant match or surpass human-provided answers to technical questions?* This question evaluates DevMentorAI's responses against those provided by human respondents in the Mozilla PDF.js project. It examines the assistant's ability to provide comprehensive and contextually grounded answers by leveraging project documentation and source code, while identifying the conditions in which human expertise remains advantageous, such as nuanced debugging, personal experience, and undocumented workarounds. We also assess how answers affect expert's satisfaction in terms of accuracy, clarity, relevance, and completeness. This question was directly addressed by our study published at AIware at FSE 2024 [13]. This question was inspired by the former study, demanded by our interaction with Mozilla Corporation (along the sandwich program of the first author), and also inspired in a preliminary study involving machine learning projects reported in SBQS 2020 [15].
- **RQ2.2.** *How do RAG-based answers compare with those from a general-purpose LLM and human-provided answers in terms of helpfulness, comprehensiveness, conciseness, and expert preference?* This question compares the answers of a project-aware RAG assistant, a general-purpose LLM, and human respondents in the Mozilla Firefox ecosystem. It examines whether grounding responses in up-to-date project data improves coverage and contextual accuracy, while characterizing the associated trade-off in conciseness. We also identify the circumstances in which human responses remain competitive, particularly for narrow-scope questions, and where general-purpose LLMs overlook project-specific nuances. This question is addressed by our study published at the main track of ICSE 2026 [8]. This question emerged from our previous study and collaboration with Mozilla Corporation.
- **RQ3.** *How do developers perceive the ability of a multi-agent architecture to detect and fix software dependability issues?* This question shifts the focus from question answering to the identification of dependability issues and the generation of actionable code fixes. We assess developers' perceptions of the relevance of the detected issues, the effectiveness of the proposed fixes, and the acceptability of the generated patches

within existing code-review practices. We also examine the factors that influence the acceptance of these recommendations, including contextual alignment, issue severity, and the necessity of proposed validation checks. This question originates from our study accepted at the research track of SBES 2026 [17]. The extended version of this study is reported in an article recently submitted to a journal.

4 Methodology

This work pursues its overarching goal through four complementary empirical studies, each guided by a distinct research question and employing tailored methodological approaches. The work adopts a mixed-methods approach spanning both observational and experimental studies, grounded in three types of systems: (i) open-source Java projects from the Apache ecosystem (Study 1); (ii) large open-source projects (Mozilla Firefox and PDF.js) in Studies 2 and 3; and (iii) industrial closed-source Python systems from a Brazilian R&D institute (Study 4). This diversity ensures that findings are grounded in realistic settings and can be triangulated across development environments.

For quantitative analyses, we employ statistical hypothesis testing (Mann-Whitney tests, paired Wilcoxon tests) complemented by effect-size measures (Cliff's delta) to assess both statistical significance and practical importance. For qualitative analyses, we use systematic manual classification with multiple validators, collaborative conflict resolution, and open coding to distill patterns and insights from unstructured data (pull request discussions, expert evaluations, developer feedback).

The remainder of this section details the design of each study. Study 1 characterizes exception-related pull requests in open-source Java projects (Section 4.1). Studies 2 and 3 evaluate project-aware conversational assistance, first comparing a retrieval-augmented agent against human answers in PDF.js (Section 4.2) and then against a general-purpose LLM and humans in Firefox (Section 4.3). Finally, Study 4 assesses a multi-agent pipeline that detects and repairs dependability issues in industrial systems (Section 4.4).

4.1 Study 1: Exception-Related Pull Requests

Research Question. RQ1 addresses how PR dynamics and contributor roles differ between exception-PRs and non-exception PRs in open-source systems, and what exception-related aspects are most frequently addressed by developers.

Data Collection and Repositories. We mined 988 exception-related pull requests (exception-PRs) from three Apache Java projects: Apache Druid (126 exception-PRs, 1.30% of 9,675 total PRs), Apache Pulsar (294, 2.07% of 14,198), and Apache Flink (568, 2.49% of 22,728). Projects were selected according to criteria including: (i) repository size (>5,000 commits), (ii) active development (commits within one month of data collection), (iii) ongoing PR discussions, and (iv) diverse application domains.

Identification Heuristic. We designed a conservative heuristic to identify exception-PRs with high precision. A PR qualifies if it satisfies three conditions: (C1) the PR title contains keywords such as 'exception', 'throw', or 'catch' (or derivatives); (C2) the PR description contains the same keywords; and (C3) the PR was not opened by a bot. This heuristic achieved 90.7% precision when manually validated on a random sample of 280 PRs.

Metrics and Manual Validation. For each PR, we collected metrics including the number of reviews, review requests, and comments. For each developer, we computed ratios of pull requests, reviews, review requests, and comments relative to their activity in exception-PRs and non-exception-PRs respectively.

The manual validation process was performed by eight experienced developers who each evaluated 70 PRs. Validators answered three questions per PR: (Q1) What was the PR's aim? (Q2) Is this PR exception-related? (Q3) Which exception aspects were touched? Disagreements on Q1 and Q2 were resolved through collaborative review by a third evaluator. For Q3, responses were consolidated by merging both validators' answers.

Data Analysis. For RQ1, we conducted Mann-Whitney tests to compare PR metrics (reviews, comments, review requests) between exception-PRs and non-exception-PRs, with Cliff's delta to quantify effect sizes. For developer behavior, we used paired Wilcoxon tests to compare developer metrics within the same population. We tabulated the frequency of exception aspects (error representability, error recoverability, error detectability, error propagation, and cleanability) and analyzed their co-occurrence using contingency tables. We also examined how these aspects were distributed across PR purposes: improvement, bug fix, new feature, test, and other.

4.2 Study 2: Conversational Agents

Research Question. RQ2.1 aims at evaluating to what extent a project-aware conversational assistant can match human-provided answers to technical questions in real-world development contexts.

System Design. DevMentorAI is a retrieval-augmented generation (RAG) conversational assistant designed to answer questions using project-specific artifacts. It ingests source code, technical documentation, wiki pages, and web pages; converts supported content into a standardized representation; splits it into semantically coherent chunks; and stores vector representations for retrieval. For each question, the system retrieves relevant project context and uses it, together with the conversation history, to prompt GPT-3.5 Turbo (4K context window) to generate an answer. Its modular architecture supports integration with multiple artifact sources and communication platforms.

Case Study Setup. We conducted a case study of Mozilla's PDF.js project. We exported the project's public Matrix chat history up to January 16, 2024, obtaining 3,492 messages. Two researchers inspected the messages to identify technical questions with suitable human answers, excluding unanswered questions, non-technical questions, questions requiring external resources or additional thread context, and follow-up exchanges. This process yielded 14 question–answer pairs. For each question, we started a new DevMentorAI conversation and generated one response. A PDF.js expert developer, with 15 years of programming experience and four years on the project, independently assessed the human and DevMentorAI answers on a five-point satisfaction scale (0 = not at all satisfied; 4 = completely satisfied).

Analysis. We compared DevMentorAI's answers to human answers through the expert's satisfaction judgments, categorizing outcomes as: DevMentorAI outperforms humans, matches human quality, or underperforms. For each case, we analyzed the factors contributing to success or failure (e.g., documentation coverage, code relevance, accuracy, actionability, clarity).

4.3 Study 3: Comparing RAG, LLMs, Humans

Research Question. RQ2.2 examines how experts assess answers from a RAG-enhanced model, a general-purpose LLM, and human contributors to real technical questions from the Firefox developer community. The assessment considers helpfulness, comprehensiveness, conciseness, and preference in practice.

Study Setup. We collected publicly available conversations from three Mozilla Matrix chat rooms focused on Firefox development. After preprocessing and systematic filtering, we selected 52 technical question–answer pairs for evaluation. We compared answers from three sources: (i) the original human answer posted in Matrix; (ii) an answer generated by GPT-4o; and (iii) an answer generated by GPT-4o enhanced with RAG. For the RAG condition, we adapted the open-source Cognita framework to ingest Firefox source code and project documentation from the public `mozilla/gecko-dev` repository. Relevant code, documentation, wiki, and web-content chunks were retrieved based on semantic similarity to each question and appended to the GPT-4o prompt. Both agents used the same prompt, which requested a direct, one-paragraph answer consistent with developer-chat conventions.

Expert Evaluation. Eight Mozilla engineers evaluated the answers. Each engineer assessed ten question–answer sets: six assigned only to that evaluator and four assessed by all eight engineers to support inter-rater agreement analysis. For each question, the three answers were anonymized, presented in random order, and their sources were not disclosed. Evaluators first assessed each answer independently using binary judgments for three attributes: (i) helpfulness, i.e., whether the answer provided meaningful assistance; (ii) comprehensiveness, i.e., whether it included the essential elements needed to address the question; and (iii) conciseness, i.e., whether it conveyed the necessary information without unnecessary detail. They then selected the most helpful, most comprehensive, and most concise answer among the three alternatives, as well as the answer they would prefer to receive in practice. Evaluators could also provide open-ended feedback on aspects of the answers.

Analysis. We calculated the proportion of answers from each source judged helpful, comprehensive, and concise, and recorded how often each source was selected as most helpful, most comprehensive, most concise, and preferred in practice. We used pairwise Fisher's Exact Tests to compare the distributions of attribute judgments across answer sources. For the four commonly assessed questions, we used Fleiss' Kappa to examine inter-rater agreement. We also analyzed the evaluators' open-ended feedback to identify perceived strengths, limitations, and gaps in the answers.

4.4 Study 4: A Multi-Agent Architecture for Dependability Issue Detection and Repair

Research Question. RQ3 investigates how developers perceive the ability of a multi-agent LLM-based architecture to proactively detect and repair software dependability issues, and whether such outputs are suitable for code review workflows.

Multi-Agent Pipeline. The pipeline receives a software repository as input and comprises four sequential agents. The *Issue Detector* uses retrieval-augmented generation (RAG) and GPT-5-mini to identify potential dependability issues in individual source files. The *Issue Validator*, powered by Gemini-2.5-Flash-Lite, assesses whether each detected issue is a valid dependability concern and filters false positives. The *Code Fixer* uses RAG and GPT-4o-mini to generate a candidate repair based on the validated issue and repository context. Finally, the *Code Judger*, also powered by Gemini-2.5-Flash-Lite, assesses whether the proposed repair adequately addresses the issue and returns a *Confirm* or *Reject* decision. The agents use role-specific prompts, one-shot examples, and structured outputs. Only issues that pass all four stages are retained as recommendations.

Case Study Setup. We evaluated the pipeline using three closed-source, production Python systems developed by private organizations in Brazil for government institutions. The systems comprised 285 source-code files. To balance representativeness and inference costs, we randomly sampled files from each system at a 95% confidence level, resulting in 129 files (64, 43, and 22 files, respectively). Running the pipeline on this sample produced 149 issues with candidate solutions. We randomly selected 40 solutions (27% of the 149 outputs) for developer validation.

Developer Assessment. Eight professional developers participated in the validation. Each selected solution was independently reviewed by two developers; each developer reviewed ten solutions, yielding 80 individual evaluations. For each solution, developers received the identified issue, the affected code fragment, the proposed fixed code, and a brief fix rationale. They rated four statements on a five-point Likert scale, from 1 (*strongly disagree*) to 5 (*strongly agree*): (i) the identified problem is relevant and worth addressing; (ii) the fixed code effectively resolves the stated problem; (iii) they would approve the patch in a code review; and (iv) they are familiar with the code and context of the solution. Developers also provided a one-sentence open-ended justification.

Analysis. For each solution, we aggregated the two developers' Likert-scale evaluations by calculating the mean score for issue relevance, fix effectiveness, code-review approval, and code familiarity. We visually inspected differences between paired evaluations to identify cases of reviewer agreement and disagreement. We then used open coding to classify issue descriptions and mapped each issue to a dependability dimension, including reliability, maintainability, confidentiality, safety, availability, integrity, and security. Finally, we analyzed the developers' written justifications using open and axial coding to identify positive and negative rationales for accepting or rejecting the proposed issues and fixes.

5 Results and Discussions

This section presents the results of four complementary studies that address the work's three research gaps. Together, they aim to *strengthen software dependability through developer support*, using exception handling as a concrete entry point. Exception handling operationalizes at a code level, reliability and availability through graceful degradation and recovery; integrity through rollback and resource cleanup; confidentiality through error-message and stack-trace sanitization; and maintainability through clear propagation paths and handlers. The studies progress from *understanding* exception work (RQ1), to *assisting* developers with project-aware answers (RQ2.1–RQ2.2), to *acting* through automated detection and repair of dependability issues (RQ3).

Table 1: Aspects modified by the 254 exception-PRs. A PR may touch several aspects, so # sums to more than 254.

	Occurrence		PR aim (%)		
Exception aspect	**#**	**%**	**Bug fix**	**Improv.**	**Other**
External error representability	108	42.52	19.44	**77.78**	2.78
Recoverability – effective handling	92	36.22	**48.91**	46.74	4.35
Error detectability	60	23.62	40.00	**56.67**	3.33
Internal error representability	52	20.47	36.54	**63.46**	0.00
Recoverability – program continuity	52	20.47	**55.77**	42.31	1.92
Other (non-exception-related code)	50	19.69	**68.00**	26.00	6.00
Error propagation	42	16.54	26.19	**69.05**	4.76
Recoverability – cleanability	19	7.48	36.84	**63.16**	0.00

5.1 RQ1: Revealing Error-Handling Practices

We mined 46,601 pull requests from three Apache Java projects: Druid, Pulsar, and Flink. A conservative keyword heuristic identified 988 exception-related PRs (2.12%). Eight experienced developers double-validated a sample of 280 PRs, confirming 254 cases (90.7% precision) and supporting the population-level analysis.

Exception work is routine, not specialized. Across the three projects, Mann–Whitney tests found no practically meaningful differences in review effort, measured by the number of reviews, comments, and review requests, between exception-PRs and non-exception-PRs. The only statistically significant comparison was comment count in Druid ($p = 0.033$), but its Cliff's delta was negligible. Developer-level analysis offers a more nuanced view. Contributors in Druid and Pulsar were significantly more likely to *initiate* exception-PRs ($p = 0.008$ and $p = 0.006$), whereas Flink showed no such tendency ($p = 0.160$); ratios for reviewing, requesting, and commenting did not differ. Thus, although some projects may encourage developers to initiate exception-related work, these changes receive the same review attention as other contributions. This finding challenges the long-standing notion of a dedicated "exception engineer" [37]: exception handling is integrated into ordinary feature work rather than assigned to specialists.

Developers refine, not merely fix. Manual characterization of the 254 confirmed PRs (Table 1) shows that improvements (55.12%) outnumber bug fixes (40.55%), contrasting with the literature's near-exclusive focus on exception *defects*. New features are rare (0.79%) because Java exception declarations typically ship with the functional code they protect; consequently, robustness work is embedded in feature PRs and remains invisible to a title/body heuristic. The most frequently modified aspect is *external error representability*, namely how errors appear in logs, traces, and user-facing messages, which is addressed in 42.52% of PRs; 77.8% of these changes are improvements rather than fixes. Effective error handling (handler logic) follows at 36.22%. The remaining aspects appear less often and span a wide range — detectability (23.62%), internal representability and program continuity (20.47% each), error propagation (16.54%), and cleanability (7.48%).

Aspects are entangled, and so is the work. Exception-PRs rarely touch a single concern in isolation. Developers routinely combine representability with recovery, or detectability with recovery and representability, in one PR, mirroring the interdependence of the underlying mechanism: a clearer message (representability) is meaningless if the wrong handler runs (recovery) on an undetected condition (detectability). This entanglement is itself a finding: tool support that treats handler logic, control flow, and message quality as separate problems is misaligned with the actual work.

A plausible lifecycle explanation connects these observations. Under deadline pressure, developers may first stabilize control flow with broad, generic handlers. They later refine this initial solution by narrowing exception types, clarifying messages, and aligning recovery behavior with the project's fault-tolerance policy. This iterative cycle is largely unsupported by current tools: static analyzers and automated-repair frameworks focus on control-flow violations and overt bugs, offering little support for the incremental, usability-oriented refinement that dominates real exception work.

Contribution (C1). This study reframes exception handling as *routine, proactive, multi-aspect, and socially integrated* work. It also exposes a concrete mismatch: developers invest heavily in error representability and message clarity, where tool support is limited. This motivates the following studies' shift from reactive debugging to proactive, context-aware developer assistance.

5.2 RQ2: Assessing Conversational Assistance

We address RQ2.1 and RQ2.2 in two complementary studies. Having located *where* assistance is needed, Studies 2 and 3 ask whether conversational agents grounded in project artifacts can deliver it. DevMentorAI, a retrieval-augmented (RAG) assistant ingesting PDF.js documentation and source, was evaluated by the project's expert on 14 real developer questions. It *outperformed* the human answer in 8 questions, *tied* in 3, and was beaten in only 3; in 5 of its 8 wins the margin was at least two scale points. Its advantage came from grounded answers that surfaced project features and tools human respondents sometimes omitted or only briefly described. Humans remained stronger on questions requiring lived experience, nuanced debugging, or undocumented workarounds.

Study 3 extended the comparison to Mozilla Firefox, evaluating a RAG assistant, GPT-4o, and human contributors on 52 questions from three Matrix channels. Eight Mozilla engineers assessed the answers for helpfulness, comprehensiveness, and conciseness (Table 2). When engineers picked the single best answer per attribute, RAG was chosen most helpful in 46.5% of questions (vs. 30.2% human, 23.2% GPT) and most comprehensive in 55.8% (vs. 20.9% human, 23.2% GPT). Conciseness was the one attribute where RAG lost: GPT was judged most concise (35.7%), ahead of humans (33.3%) and RAG (30.9%). When asked which answer they would most want to *see in practice*, engineers chose RAG in 39.5% of cases, ahead of humans (34.8%) and GPT (25.5%).

Table 2: Percent of questions where the source was the *best* per attribute, and the source engineers prefer in practice.

Criterion (best source)	**Human**	**GPT-4o**	**RAG**
Helpfulness	30.2%	23.2%	**46.5%**
Comprehensiveness	20.9%	23.2%	**55.8%**
Conciseness	33.3%	**35.7%**	30.9%
Preferred in practice	34.8%	25.5%	**39.5%**

Two-sided Fisher's exact tests sharpen these proportions. RAG is statistically *more comprehensive* than GPT ($p = 0.040$, $OR = 0.393$) and shows no statistically significant difference from human experts on any attribute (helpfulness $p = 0.808$; comprehensiveness $p = 0.534$; conciseness $p = 1.000$). GPT, by contrast, was significantly

less helpful than humans ($p = 0.017$, $OR = 3.215$), and the GPT-vs-RAG helpfulness gap was borderline in RAG's favor ($p = 0.054$). A correlation analysis explains why this matters in practice: an answer's perceived helpfulness predicts whether engineers would adopt it ($r = 0.84$), comprehensiveness somewhat less ($r = 0.76$), and conciseness least of all ($r = 0.51$). Brevity is desirable but secondary; engineers trade verbosity for correctness and coverage.

Grounding and its fragility. The qualitative coding made the trade-off clear. When RAG retrieved current project information, it gave a precise answer, such as identifying `NS_ConvertUTF16toUTF8` for converting an `nsString` to an `nsCString`, without unnecessary detail. Without that grounding, GPT sometimes produced convincing but incorrect answers, including a non-existent `./mach build –enable-release` flag and the false claim that an API returned a promise. RAG was not immune to this problem: when its documentation was outdated, it confidently recommended a deprecated API. In practice, RAG is only as reliable as the project knowledge it retrieves. Keeping that knowledge current is therefore essential.

Contribution (C2). Grounding LLMs in project artifacts produces answers that experts more often judge helpful, comprehensive, and preferable in practice than generic LLM or human answers, although sometimes at the cost of conciseness. This establishes the feasibility of agent-based developer support and its key requirement: current, project-specific documentation. In practice, such support can help answer recurring maintainer queries and ease newcomer onboarding.

5.3 RQ3: Automated Detection and Repair

The final study moves from answering questions to proposing fixes. We evaluated a four-agent pipeline on three closed-source systems in production for Brazilian government institutions. Its stages run without human intervention, but it never applies its own patches — every output enters code review as a proposal, so developer acceptance, not test approval, is our criterion for a successful fix.

From raw detections to review-ready fixes. We ran the pipeline on 129 sampled files, representing 45% of the three codebases and selected using a 95% confidence level. The successive stages filtered 230 initial detections into 215 validated issues and, ultimately, 149 issue–fix pairs accepted by the Code Judger. To understand whether these recommendations would make sense to practitioners, eight professional developers independently reviewed a random subset of 40 issues. Each issue was assessed by two developers, resulting in 80 evaluations on a five-point Likert scale.

The results in Table 3 are encouraging. Developers generally agreed that the detected issues were relevant (4.47/5) and that the proposed fixes addressed them effectively (4.35/5). More importantly, they also agreed that they would approve the suggested patches in a code review (4.16/5), indicating that the outputs were not merely plausible in isolation but often fit developers' practical expectations. Reviewers also reported familiarity with the code and its context (4.00/5), supporting these assessments.

Table 3: 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

Where it works, and where it does not. The detection profile in Table 4 is concentrated on concerns that can be identified from local code context. Integrity (32.5%), reliability (25%), and maintainability (20%) account for more than three quarters of the detected issues. Security (12.5%), confidentiality (7.5%), and particularly availability (2.5%) appear less often. This pattern reflects an important boundary of the approach: availability problems and many security vulnerabilities depend on runtime behavior or broader system context, which static source code alone cannot reliably reveal.

One successful case illustrates the kind of issue the pipeline handles well. A model contained a `deleted_at` field, suggesting soft deletion, but its default `delete()` operation still permanently removed records. The Code Fixer introduced a `soft_delete()` method and a `DocumentQuerySet` for filtering active rows. The developers unanimously accepted this change because it made the intended data-retention policy explicit.

Table 4: Dependability dimensions of the 40 human-validated issues (descending by frequency; total = 40).

	Integrity	Reliability	Maintainability	Security	Confidentiality	Availability
Issues	13	10	8	5	3	1
Share	32.5%	25.0%	20.0%	12.5%	7.5%	2.5%

Disagreement reveals the human boundary. When the two reviewers disagreed about a fix, the difference usually reflected contextual judgment rather than a simple question of technical correctness. Four recurring concerns emerged: *incomplete or misaligned fixes*, such as bilingual error handling that conflicted with an English-by-default design; *low-severity issues*, where reviewers assigned different levels of risk to the same flaw; *uncertain parameter validation*, when defensive checks seemed either prudent or redundant; and *security and robustness concerns*, such as removing a refresh token from a payload without addressing the underlying vulnerability. These judgments require human understanding of severity, architectural impact, and root causes. They remain an important role for human reviewers.

Contribution (C3). A multi-agent pipeline can detect, validate, and repair real dependability issues, producing review-ready fixes. The evidence positions it as a specialized reviewer that augments human oversight rather than as an autonomous repair tool: it validates the feasibility of unsupervised detection and patch generation while delimiting where human expertise remains irreplaceable.

5.4 Conversational Agents for Dependability

Together, the findings of our four studies can be synthesized as follows. Study 1 identifies where current tooling falls short: exception work is routine, refinement-oriented, and often focused on error representability, an area that existing tools rarely support. Studies 2 and 3 show that project-grounded conversational agents can provide useful assistance in this space, matching or exceeding human answers when they draw on current project knowledge. Study 4 extends this support from answering questions to proposing repairs that developers can assess in review, while retaining human oversight.

Three conclusions follow from this evidence. *(i)* Exception handling is practical and distributed work, not a specialist activity; dependability tools should therefore fit ordinary development and

review workflows. *(ii)* Trustworthy assistance depends on project grounding rather than model scale alone. In Firefox, RAG produced answers comparable to human responses, while in closed-source systems it enabled context-aware fixes that generic models could not derive from public knowledge alone. *(iii)* Multi-agent systems can support the detection and repair of dependability issues when their outputs are transparent and subject to human validation. Their limits remain clear, however: runtime behavior and architectural context cannot be fully recovered from static code alone.

Taken together, the studies identify where automated support can add value and what it requires: current documentation, project-specific context, and human review. They provide evidence that conversational agents can support the identification and resolution of software dependability issues. They also point to the next steps: architecture-aware retrieval, runtime-informed detection, and feedback-learning agents integrated into PR workflows.

6 Threats to Validity

Internal validity. The heuristic used to identify exception-related PRs may have missed relevant changes. We designed it to align with the study objectives and used double validation and collaborative conflict resolution to reduce error and bias. In the multi-agent study, random file sampling may not fully represent each codebase; we mitigated this risk through 95% confidence-level sampling and recorded reviewers' familiarity with the assessed code. Developers assessed only issue–fix pairs that had already passed the Issue Validator and the Code Judger, so the reported ratings characterize the quality of the pipeline's retained output rather than its precision or recall: pairs discarded at those stages were never shown to a human, leaving false negatives unmeasured. In Study 2, a single expert assessed all answers, so satisfaction judgments reflect one perspective; Study 3 addresses this with eight independent evaluators.

External and construct validity. The studies cover selected projects: three Apache Java systems, Mozilla PDF.js and Firefox, and three industrial Python systems developed by private organizations in Brazil. This provides depth but limits generalization across languages, ecosystems, organizations, and regions. Further evaluation across technology stacks and domains is needed. Helpfulness, comprehensiveness, and conciseness do not capture every dimension of answer quality. The one-paragraph response requirement, while consistent with developer-chat conventions, may also favor concise answers. Our definition of dependability was grounded in established literature and mapped to recognized dimensions; multiple researchers cross-checked the open coding to reduce subjectivity.

Conclusion validity. Some of our evaluations rely on expert judgments and restricted samples, including forty solutions assessed by eight developers, which limits statistical power; non-significant comparisons should therefore be read as absence of evidence rather than evidence of equivalence. We emphasized effect sizes and patterns consistent across dimensions. PDF.js and Firefox are public repositories that may appear in the training data of the evaluated models, which could favor the general-purpose baseline in Study 3 and understate the contribution of retrieval. Finally, LLM outputs are non-deterministic and depend on model versions that evolve over time, but detailed methods and open replication packages support reproducibility.

7 Conclusion: Impact and Implications

As described here, our findings led to research implications and actionables, such as the production of agent systems. The direct impact of our produced agent systems are our previous industry partners as well as industry and academia stakeholders accessing all our replication packages. The beneficiaries of our research are essentially all software project teams that need to meet dependability requirements along issue resolution in software maintenance and evolution. In particular, almost every software program has exception handling behavior, the primary means to achieve fault tolerance and software dependability as a whole (Section 1).

Our research used exception handling as a lens for studying how developer support can strengthen software dependability. Three lessons emerged. First, *exception handling is everyday work, but tools lag behind practice*: although exception-related PRs are a small share of contributions, they often refine error representation and handler behavior, forms of incremental and usability-oriented work that static analyzers and repair frameworks largely overlook. Second, *project grounding matters more than model scale*: access to current documentation, source code, and project links enabled agents to outperform generic models and, in several cases, human answers. This is especially important in closed-source settings, where generic models cannot access proprietary artifacts. Third, *conversational AI can move from answering to acting*: a multi-agent pipeline generated patches that developers were generally willing to review when they were explainable, context-aware, and subject to human oversight.

These findings show that project-grounded conversational agents can assist developers in identifying and resolving multi-attribute dependability issues. We significantly advanced the state-of-the-art with these findings. As reported in our papers [8, 10, 17], previous research has focused only on investigating techniques or agents to resolve a very strict set of specific problems of a particular dependability attribute, such as certain security vulnerabilities or specific maintainability smells, not involving industry projects and actual developers, as our research covered. Moreover, these existing works do not assess how LLM-based agents can depart from real dependability problems reported upfront as natural-language issue descriptions. Much of the existing work does not fully represent the multifaceted nature of dependability in complex, industrial systems.

Artifact Availability

The research artifacts supporting this work, including source code, datasets, experimental materials, and replication packages, are publicly available. Artifacts associated with *Study 1* are available in [11], those for *Study 2* in [12], for *Study 3* in [9], and for *Study 4* in [18].

Acknowledgements

This work was supported by the National Science Foundation grants 2236198, 2247929, 2303042 and 2303043. Also, CAPES: 88887.899310 /2023-00, 88887.900069/2023-00 and 88887.915794/2023-00. CNPq: 381697/2026-6, 140770/2021-6, 141180/2021-8, 140771/2021-2, 315711 /2020-5, 141276/2020-7 and 141054/2019-0. FAPERJ: 200.510/2023. 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. Finally, we thank the Mozilla engineers and Brazilian industry partners for the collaboration along the studies.

References

[1] 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

[2] Martin Balfroid, Benoît Vanderose, and Xavier Devroey. 2024. Towards LLM-Generated Code Tours for Onboarding. In *Workshop on NL-based Software Engineering (NLBSE'24)*.

[3] Eiji Adachi Barbosa, Alessandro F. Garcia, and Simone Diniz Junqueira Barbosa. 2014. Categorizing Faults in Exception Handling: A Study of Open Source Projects. In *2014 Brazilian Symposium on Software Engineering*. IEEE Computer Society, 11–20. doi:10.1109/SBES.2014.19

[4] Joshua Bloch. 2018. *Effective Java* (3rd ed.). Addison-Wesley.

[5] Nélio Cacho, Eiji Adachi Barbosa, Juliana Araujo, Frederico Pranto, Alessandro Garcia, Thiago Cesar, Eliezio Soares, Arthur Cassio, Thomas Filipe, and Israel Garcia. 2014. How Does Exception Handling Behavior Evolve? An Exploratory Study in Java and C# Applications. In *IEEE International Conference on Software Maintenance and Evolution*. 31–40. doi:10.1109/ICSME.2014.25

[6] Nélio Cacho, Thiago César, Thomas Filipe, Eliezio Soares, Arthur Cassio, Rafael Souza, Israel Garcia, Eiji Adachi Barbosa, and Alessandro Garcia. 2014. Trading robustness for maintainability: an empirical study of evolving C# programs. In *36th International Conference on Software Engineering*. ACM, 584–595. doi:10.1145/2568225.2568308

[7] Cognita. [n. d.]. Cognita. https://github.com/truefoundry/cognita. (Accessed on 06/05/2026).

[8] 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 48th IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE/ACM.

[9] João Correia, Daniel Coutinho, Marco Castelluccio, Caio Barbosa, Rafael de Mello, Anita Sarma, Alessandro Garcia, Marco Gerosa, and Igor Steinmacher. 2026. Replication Package: A Comparison of Conversational Models and Humans in Answering Technical Questions: the Firefox Case. https://github.com/RESHAPELab/answering-technical-questions-firefox. Accessed: 2026-06-05.

[10] 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), in conjunction with International Conference on Software Maitenance and Evolution (ICSME 2024)*. 143–153. doi:10.1109/SCAM63643.2024.00023

[11] 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. Replication Package: On the Investigation of Exception Pull Request Characteristics: Exploring the Apache Ecosystem. (2024). https://github.com/opus-research/robustness-prs-replication Accessed on: 2026-06-05.

[12] João Correia, Morgan C. Nicholson, Daniel Coutinho, Caio Barbosa, Marco Castelluccio, Marco Gerosa, Alessandro Garcia, and Igor Steinmacher. 2024. Replication package: Unveiling the Potential of a Conversational Agent in Developer Support: Insights from Mozilla's PDF.js Project. https://github.com/RESHAPELab/DevMentorAI-PDF.js-replication. Accessed: 2026-06-05.

[13] João Correia, Morgan C. Nicholson, Daniel Coutinho, Caio Barbosa, Marco Castelluccio, Marco Gerosa, Alessandro Garcia, and Igor Steinmacher. 2024. Unveiling the Potential of a Conversational Agent in Developer Support: Insights from Mozilla's PDF.js Project. In *Proceedings of the 1st ACM International Conference on AI-Powered Software, Foundations of Software Engineering (FSE) Conference* (Porto de Galinhas, Brazil) *(AIware 2024)*. Association for Computing Machinery, New York, NY, USA, 10–18. doi:10.1145/3664646.3664758

[14] João Lucas Correia. 2025. *Exploring Conversational Agents for Developer Guidance on Software Dependability Issues.* PhD Thesis. Pontifical Catholic University of Rio de Janeiro, Informatics Department, Rio de Janeiro, Brazil. https://www.maxwell.vrac.puc-rio.br/conteudo_simples_indexacao.php?nrSeq=74447

[15] João Lucas Correia, Juliana Alves Pereira, Rafael Maiani de Mello, Alessandro Garcia, Baldoino Fonseca, Márcio Ribeiro, Rohit Gheyi, Marcos Kalinowski, Renato Cerqueira, and Willy Tiengo. 2020. Brazilian Data Scientists: Revealing their Challenges and Practices on Machine Learning Model Development. In *19th Brazilian Symposium on Software Quality, SBQS 2020, São Luís, Brazil, December, 2020*, Davi Viana and Marcelo Schots (Eds.). ACM, 10. doi:10.1145/3439961.3439971

[16] João Lucas Marques Correia, Daniel Coutinho, Alessandro Garcia, Rafael Maiani de Mello, Wesley K. G. Assunção, Caio Barbosa, Nélio Cacho, and Thaís Vasconcelos Batista. 2026. A Conversational Multiagent Architecture for Resolving Issues on Multi-Attribute Dependability. *submitted to a journal* (2026), 1–20.

[17] João Lucas Marques Correia, Daniel Coutinho, Alessandro Garcia, Rafael Maiani de Mello, Wesley K. G. Assunção, Caio Barbosa, Nélio Cacho, and Thaís Vasconcelos Batista. 2026. Developers' Perceptions of Automated Detection and Resolution of Software Dependability Issues. In *Proceedings of the 40th Brazilian Symposium on Software Engineering (SBES)* (São Paulo, Brazil).

[18] João Lucas Marques Correia, Daniel Coutinho, Alessandro Garcia, Rafael Maiani de Mello, Wesley K. G. Assunção, Caio Barbosa, Nélio Cacho, and Thaís Vasconcelos Batista. 2026. Research Artifacts for "Developers' Perceptions of Automated Detection and Resolution of Software Dependability Issues". https://github.com/opus-research/developers_perception_dependability. Research artifact and replication package. Accessed: 2026-06-05.

[19] Laura Dabbish, Colleen Stuart, Jason Tsay, and Jim Herbsleb. 2012. Social coding in GitHub: transparency and collaboration in an open software repository. In *Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work* (Seattle, Washington, USA) *(CSCW '12)*. Association for Computing Machinery, New York, NY, USA, 1277–1286. doi:10.1145/2145204.2145396

[20] Felipe Ebert, Fernando Castor, and Alexander Serebrenik. 2015. An exploratory study on exception handling bugs in Java programs. *Journal of Systems and Software* 106 (2015), 82–101. doi:10.1016/j.jss.2015.04.066

[21] Felipe Ebert, Fernando Castor, and Alexander Serebrenik. 2020. A Reflection on "An Exploratory Study on Exception Handling Bugs in Java Programs". In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 552–556. doi:10.1109/SANER48275.2020.9054791

[22] Mats Finsås and Joachim Maksim. 2024. *Optimizing RAG Systems for Technical Support with LLM-based Relevance Feedback and Multi-Agent Patterns.* Master's thesis. NTNU.

[23] Felipe Fronchetti, David C. Shepherd, Igor Wiese, Christoph Treude, Marco Aurélio Gerosa, and Igor Steinmacher. 2023. Do CONTRIBUTING Files Provide Information about OSS Newcomers' Onboarding Barriers?. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (San Francisco, CA, USA) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 16–28. doi:10.1145/3611643.3616288

[24] Md Saleh Ibtasham, Sarmad Bashir, Muhammad Abbas, Zulqarnain Haider, Mehrdad Saadatmand, and Antonio Cicchetti. 2025. ReqRAG: Enhancing Software Release Management through Retrieval-Augmented LLMs: An Industrial Study. In *Requirements Engineering: Foundation for Software Quality: 31st International Working Conference, REFSQ 2025, Barcelona, Spain, April 7–10, 2025, Proceedings* (Barcelona, Spain). Springer-Verlag, Berlin, Heidelberg, 277–292. doi:10.1007/978-3-031-88531-0_20

[25] 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

[26] J.C. C. Laprie, A. Avizienis, and H. Kopetz. 1992. *Dependability: Basic Concepts and Terminology.* Springer-Verlag, Berlin, Heidelberg.

[27] 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

[28] Arghavan Moradi Dakhel, Vahid Majdinasab, Amin Nikanjam, Foutse Khomh, Michel C. Desmarais, and Zhen Ming (Jack) Jiang. 2023. GitHub Copilot AI pair programmer: Asset or Liability? *Journal of Systems and Software* 203 (2023), 111734. doi:10.1016/j.jss.2023.111734

[29] Suman Nakshatri, Maithri Hegde, and Sahithi Thandra. 2016. Analysis of exception handling patterns in Java projects: an empirical study. In *13th International Conference on Mining Software Repositories*. ACM, 500–503. doi:10.1145/2901739.2903499

[30] 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

[31] 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. *Proceedings of the ACM on Software Engineering Journal - Foundations on Software Engineering Conference (FSE 2024)* 1, FSE, Article 24 (July 2024), 22 pages. doi:10.1145/3643750

[32] Parametrix Solutions, Inc. 2025. CrowdStrike's Impact on the Fortune 500. https://www.parametrixinsurance.com/crowdstrike-outage-impact-on-the-fortune-500. Accessed: 2025-09-19.

[33] Rohith Pudari and Neil A Ernst. 2023. From copilot to pilot: Towards AI supported software development. *arXiv preprint arXiv:2303.04142* (2023).

[34] 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

[35] June Sallou, Thomas Durieux, and Annibale Panichella. 2023. Breaking the silence: the threats of using llms in software engineering. *arXiv preprint arXiv:2312.08055* (2023).

[36] Puntitra Sawadpong, Edward B. Allen, and Byron J. Williams. 2012. Exception Handling Defects: An Empirical Study. In *14th International Symposium on High-Assurance Systems Engineering*. 90–97. doi:10.1109/HASE.2012.24

[37] Hina Shah, Carsten Görg, and Mary Jean Harrold. 2008. Why do developers neglect exception handling?. In *4th International Workshop on Exception Handling*. ACM, 62–68. doi:10.1145/1454268.1454277

[38] Hina Shah, Carsten Gorg, and Mary Jean Harrold. 2010. Understanding Exception Handling: Viewpoints of Novices and Experts. *IEEE Transactions on Software Engineering* 36, 2 (2010), 150–161. doi:10.1109/TSE.2010.7

[39] 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

[40] Giovanni Viviani, Michalis Famelis, Xin Xia, Calahan Janik-Jones, and Gail C. Murphy. 2021. Locating Latent Design Information in Developer Discussions: A Study on Pull Requests. *IEEE Transactions on Software Engineering* 47, 7 (2021), 1402–1413. doi:10.1109/TSE.2019.2924006

[41] Giovanni Viviani, Calahan Janik-Jones, Michalis Famelis, Xin Xia, and Gail C. Murphy. 2018. What design topics do developers discuss?. In *26th Conference on Program Comprehension*. ACM, 328–331. doi:10.1145/3196321.3196357

[42] 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

[43] Zhentao Xu, Mark Jerome Cruz, Matthew Guevara, Tie Wang, Manasi Deshpande, Xiaofeng Wang, and Zheng Li. 2024. Retrieval-Augmented Generation with Knowledge Graphs for Customer Service Question Answering. *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval* (2024). https://api.semanticscholar.org/CorpusID:269449459

[44] 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

[45] Hao Zhong and Hong Mei. 2018. Mining repair model for exception-related bug. *Journal of Systems and Software* 141 (2018), 16–31. doi:10.1016/j.jss.2018.03.046