On the Use of Commit Messages for Corrective Software Maintenance: A Systematic Mapping Study

Syful Islam
syful.islam@telecom-paris.fr
LTCI, Télécom Paris, Institut Polytechnique de Paris
Palaiseau, France

Stefano Zacchiroli
stefano.zacchiroli@telecom-paris.fr
LTCI, Télécom Paris, Institut Polytechnique de Paris
Palaiseau, France

Abstract

Corrective maintenance is crucial to ensure the quality of software, thereby improving reliability and user experience. In a version control system (VCS), developers write commit messages to document their changes and support later maintenance. Therefore, the utilization of commit messages to accomplish corrective maintenance has become a common practice among software engineering practitioners and researchers. Still, to this day, no secondary study has mapped the research landscape of how commit messages have been used in corrective software maintenance.

We present a systematic mapping study of 97 primary sources published between 2004 and May 2025, where we examine the goals, potential utilization of source code artifacts along with commit messages, methodologies, stakeholders, and the key findings about their influence on corrective maintenance.

Our analysis reveals a growing interest in the usage of commit messages to perform corrective maintenance tasks, in particular for bug analysis and bug fix identification goals. Surprisingly few studies address other themes such as automated program repair, security development practices, etc. We find that the software artifacts most used in combination with commit messages are commit "diffs" and that repository mining, together with natural language processing (NLP) and artificial intelligence/machine learning (AI/ML) are the methodological foundations of studies in this field. Among stakeholders considered in previous studies, developers play the most important role in shaping corrective maintenance practices.

Key findings in previous studies about commit messages establish their significant role in corrective maintenance, due to the fact that they carry crucial information helpful for stakeholders to understand and improve the code base through the software evolution process. Often, though, commit messages lack important information and are not enough to convey the intent of code changes to future readers. Therefore, developers should be aware of commit message contextual richness while committing code changes in VCS.

CCS Concepts

• General and Reference → Surveys and Overviews; **• Software and its Engineering → Software Maintenance**; **• NLP → Software Artifacts**.

This work is licensed under a Creative Commons Attribution 4.0 International License.
EASE 2026, Glasgow, Scotland, United Kingdom
© 2026 Copyright held by the owner/author(s).
ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Keywords

commit messages, corrective software maintenance, systematic mapping study

ACM Reference Format:
Syful Islam and Stefano Zacchiroli. 2026. On the Use of Commit Messages for Corrective Software Maintenance: A Systematic Mapping Study. In *Proceedings of The 30th International Conference on Evaluation and Assessment in Software Engineering (EASE 2026).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Software maintenance encompasses a broad set of activities (such as implementing new features, fixing bugs, enhancements, etc.) performed on software products to ensure their quality and long-term sustainability in the market [9]. According to Swanson [14], these activities are classified into three categories, namely corrective, adaptive and perfective. Among them, corrective maintenance deals with modification of a software product performed after delivery to correct discovered bugs in source code [5, 14]. Corrective maintenance derives its importance not only from the huge costs it induces but also in the way it affects customer relations and ultimately, the revenue of the software company [2, 4, 10].

During corrective maintenance, developers need to understand the code changes enough to analyse problems, identify the bugs and determine how they should be fixed without breaking anything [16]. In VCS, commit messages play a crucial role in facilitating collaborative software development and might be the only source of information left for future developers to understand what changes were made and why [8, 15, 19]. Therefore, the usage of commit messages to accomplish corrective maintenance has become a common practice within the software engineering community.

To improve corrective software maintenance, researchers investigate the feasibility of commit message usage in combination with various artifacts such as code changes and apply different techniques, namely repository mining, NLP, AI/ML, and manual inspection. These techniques are used in various areas such as bug analysis, prediction, program repair, patch identification, etc. However, the landscape of commit message usage in corrective maintenance remains fragmented, with various goals and methodologies being employed. Researchers have conducted numerous empirical studies and propose various tools, yet little work has been done to comprehensively map this field and synthesize the findings about how commit messages influence corrective maintenance. To fill this knowledge gap, we conducted and present in this work a systematic mapping study that provides an overview of historical trends, assesses the role of commit messages in corrective maintenance, and identify areas where further research is required.

Paper structure. Section 2 details the methodology of our study. We focus on fundamental research questions, such as where and when the research was published, and the goals that drive researchers to engage in corrective software maintenance using commit messages. In addition, we examine the potential utilization of diverse source code artifacts along with commit messages, their usage patterns, methodology, stakeholder perspectives, and the key findings about how commit messages influence corrective maintenance activities. We followed a systematic methodology with defined inclusion/exclusion criteria, screening steps, backward snowballing, validated and refined via discussion between authors. This led to the selection of 97 primary studies, which were then categorized based on various dimensions related to our research questions. Section 3 presents our main results, which can be summarized as:

- There is consistent and growing research interest in the usage of commit messages to address corrective maintenance tasks, with a spike in the 2022–2023 years.
- The major goals of corrective maintenance pursued by mining commit messages are bug analysis and bug fix identification; surprisingly few studies address others such as automated program repair, security development practices, etc.
- The most dominant recurring source code artifact usage pattern is {commit messages, code changes} within the software engineering community where repository mining serves as the methodological foundation, often supported by {NLP, AI/ML} and occasionally complemented by expert manual inspection.
- Among human stakeholders considered in previous work, developers appeared most frequently in our primary sources, indicating their central role and influence in the corrective software maintenance strategies.
- Our findings reveal that commit messages play a significant role in corrective maintenance activities since they carry crucial information helpful for stakeholders (i.e., developer, maintainer, and researcher) to understand and improve code base through software evolution process. However, commit messages often lack necessary information and are not always sufficient to represent the intent of code changes.

In Section 4, we present a set of key recommendations for stakeholders such as developers, maintainers, and researchers supported by our primary sources. Before concluding (Section 6), we discuss threats to the validity of our study in Section 5.

Contributions and Data Availability. This paper presents (i) the first secondary study on the usage of commit messages for corrective software maintenance activities, and (ii) a publicly available replication package [6] containing the dataset of 97 analyzed papers and a structured taxonomy categorizing their key characteristics. All code and data used in this study are included in the package.

2 Methodology

To conduct this study, we adopt the methodology proposed by Petersen et al. [12, 13] to ensure a systematic and reproducible approach for selecting and analyzing relevant articles. The methodology is organized around three main phases: developing research questions (Section 2.1), selection of primary sources (Section 2.2), and data collection (Section 2.3).

2.1 Research questions

To systematically assess the role of commit messages in corrective software maintenance, according to scientific works published so far, we stated the research questions (RQ) listed below, together with their rationales:

- RQ_1 *Where and when was the study published?*
  Rationale: grasping the evolution over time and macro-fields of studies that rely on commit messages.
- RQ_2 *What were the goals of the study?*
  Rationale: mapping the ultimate, often unstated, reasons that drive researchers to engage in corrective software maintenance using commit messages.
- RQ_3 *What software artifacts were used in the study?*
  Rationale: grasping the potential utilization of diverse source code artifacts in corrective software maintenance, along with commit messages.
- RQ_4 *What research methodology was used in the study?*
  Rationale: understanding which methodological techniques are needed to benefit from commit messages for corrective maintenance needs.
- RQ_5 *Which human stakeholders' perspectives were considered in the study?*
  Rationale: understanding which stakeholders are involved is essential to locate who can later *improve* corrective software maintenance practices. Answer to this RQ will also help in identifying potential biases and gaps in empirical evidence, as well as opportunities to improve collaboration and knowledge transfer.
- RQ_6 *What were the key findings of the study, about how commit messages influence corrective software maintenance?*
  Rationale: understanding the key findings from studies provides overview of how commit messages influence corrective maintenance, and also best practices for practitioners.

2.2 Selection of primary sources

Figure 1 illustrates the protocol applied for selecting our primary sources. Initially, we formulated automatic query search criteria to be used on selected popular online libraries and executed them. After deduplication and noise removal of obtained results, we applied inclusion and exclusion criteria, titles and abstracts screening, reviewing full content of the sources, and conducted backward snowballing phase to obtain the final set of primary sources.

2.2.1 Step 1: Automatic query search. To identify the key terms for our search query, we adopted the PICO framework [7], commonly used in systematic reviews, which stands for "Population, Intervention, Comparison, and Outcome". This approach formulates research questions and structure as outlined by Petersen et al. [12]. Our systematic mapping study focused on the Population and Intervention components. The Population component in our study refers to software domain context entities. We defined this using the terms: software, repository, project, application, code, github. The Intervention component pertains to the classification processes

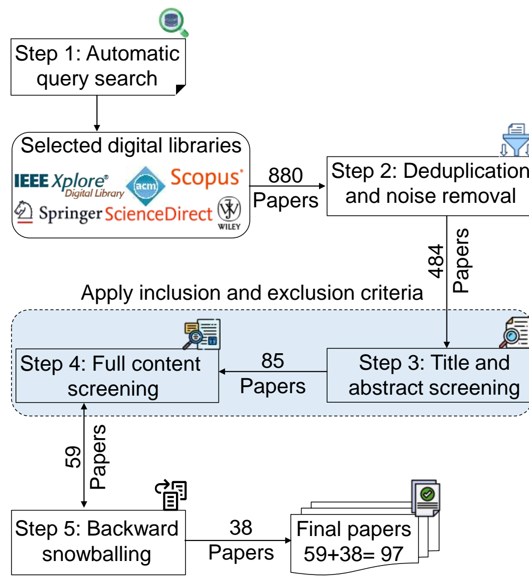

Figure 1: Protocol used for the selection of primary sources.

```
{Title:((commit OR commits) AND (message OR messages))
OR
Abstract:((commit OR commits) AND (message OR messages))}
AND
Abstract:(<domain context terms>)
```

Figure 2: Query template applied on the selected digital libraries. This query searches for the singular or plural form of "commit" and "message" words in either title or abstract and any domain context terms (i.e., "software", "repository", "project", "application", "code", "github") in abstract to keep the search highly relevant.

applied to these entities. We defined this using singular and plural forms of the terms: commit and message.

We did not include the comparison component because our study does not involve comparing different methods or interventions against each other. Our objective is to map the existing research on corrective software maintenance as a whole rather than to evaluate or contrast specific approaches. Similarly, we also excluded the outcome component because we are not focusing on specific measurable outcomes from commit message usage in corrective maintenance activities such as accuracy metrics. In summary, our aim is to identify and categorize the following characteristics of existing studies: data sources, methodologies, and key findings about commit message usage in corrective maintenance activities.

Figure 2, presents the search query template, adapted to the syntax and semantics of selected bibliographic database. This query was applied exclusively to the title and abstract of research articles. We also added domain context terms in query because "commit or commits" and "message or messages" keywords are very common

Table 1: Inclusion and exclusion criteria.

Parameters	Inclusion	Exclusion
Contribution content	Articles that includes contributions related to corrective maintenance	Articles that do not include this feature in the contribution
Language	Article written in English	Article not written in English
Source type	Must be peer reviewed	Not peer reviewed such as tutorials, editorial notes, abstract, technical reports, thesis, book chapter etc.
Accessibility	Article available in full text	Article not available in full text
Extension	If multiple publications of the same study exist presenting the same analysis, the latest version will be included	Articles for which a newer or more complete version exists

in the human psychology literature such as "No one commits suicide: Textual analysis of ideological practices"; searching for them in article title and abstract returned large amounts of non-relevant articles during our iterations on the search query. Thus, our approach ensures high relevance of the retrieved articles without compromising the breadth of the search.

We conducted automatic query search on several popular online digital libraries i.e., IEEE, ACM, Scopus, Springer, ScienceDirect, and Wiley similar to a previous study by Balla et al. [3]. These selected digital libraries represent the major repositories available in the field of software engineering. For ScienceDirect database, we applied query to the title, abstract, and keywords, as the database natively supports built-in filtering across these fields. In addition, given that Springer database does not support complex queries, we adjusted our search strategy using phrases (i.e.,"commit message" OR "commit messages" OR "commits") on article title only to maintain the core focus of the query. To reduce non-relevant articles from Springer database, we compared different search phrase combinations, evaluating the noise ratio in the returned results. Furthermore, while executing query, the whole database was searched instead of specifying time frame aiming to build a more comprehensive dataset on the target topic for systematic mapping study. Thus, we obtained a total of 880 papers from initial automatic query on the selected digital libraries.

2.2.2 Step 2: De-duplication and noise removal. In this step, our goal is to ensure that the collected papers are both unique and highly relevant to mapping study. To achieve this, we first remove duplicates based on paper titles and DOIs. We then manually inspect the remaining studies to catch any duplicates that may persist due to noise in the article metadata. This filtering ensures that every study we include offers a distinct contribution to the knowledge base. Through this process, we have identified 484 candidate papers for title and abstract screening.

2.2.3 Step 3: Title and abstract screening. In this step, we performed screening of the retrieved papers based on title and abstract, as recommended by Petersen et al. [13]. This step was performed by the first author, who read titles and abstracts, and applied inclusion/exclusion criteria listed in Table 1. Out of the 484 total papers retained thus far, 399 papers were rejected, leaving 85 potentially-relevant papers for full content screening.

Table 2: Data collection scheme.

Research Question	Dimension	Scale
RQ1	Year	Time-based scale
RQ1	Venue	Single-choice scale
RQ2	Goal	Single-valued open scale
RQ3	Type of source code artifacts used with commit message	Multi-valued open scale
RQ4	Type of research methodologies	Multi-valued open scale
RQ5	Type of stakeholders	Multi-valued closed scale
RQ6	Key findings about commit message	Multi-valued open scale

2.2.4 Step 4: Full content screening. In this step, we applied the inclusion/exclusion criteria as listed in Table 1 to the full content of the papers obtained from step 3. The task was collaboratively performed by two authors (i.e., 1st and 2nd author). This task began by discussing and aligning on systematic approach to identify the contribution content and ultimately agreeing to assess whether commit messages were mined as part of the studied methodology and were aligned with corrective maintenance. We did not measure Kappa agreement because the task was conducted collaboratively through continuous discussion and consensus-building, making statistical inter-rater reliability measurement unnecessary [11].

Based on the assessment conditions, the first author manually annotated all 85 papers obtained from step 3. The annotation process allowed the first author to mark papers as “may be” in case of uncertainty about their inclusion, prompting a second evaluation by the second author. After mitigating the uncertain cases, we accepted 59 papers relevant to our target study.

2.2.5 Step 5: Backward snowballing. To improve retrieval of relevant studies missed in the previous steps, we incorporated backward snowballing as an additional step. This process was applied to the 59 papers accepted during the selection of primary sources in step 4. The snowballing phase followed established methodology used for evaluating primary sources: we began with 59 direct studies and manually checked the references of each study iteratively. These were processed through three sequential steps: (i) removal of duplicate papers, (ii) document filtering through “commit message” keyword searching and (iii) full-content screening. During full-content screening, we found that some studies incorporated commit messages as part of their methodology but did not highlight this in the title or abstract, making it necessary to include these studies in the primary sources. After three successful backward snowballing iterations, we were not able to identify any new papers, indicating that the process had fully converged. Thus, we identified 38 additional studies, bringing the total number of primary sources to 97.

2.3 Data collection

The process of data collection aims to obtain thorough information from the primary studies corresponding to the scales, as outlined in Table 2. The data collection protocol follows standard rules and practices commonly employed in software engineering literature reviews [7, 20]. This involves a series of steps conducted by both authors in three collaborative iterations.

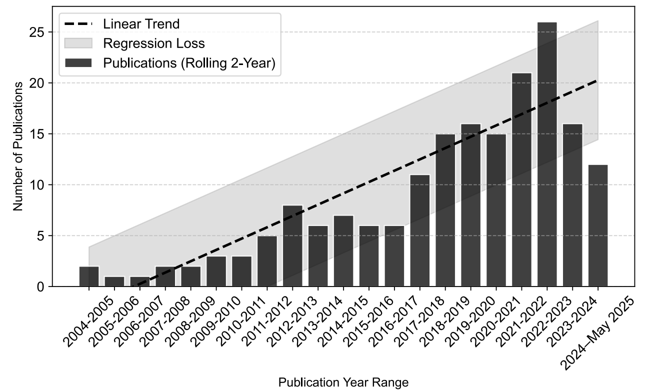

Figure 3: Year-wise publication (rolling 2-year) trend.

During initial iteration, both authors reviewed the same 25 sample papers from the 97 primary sources. They reviewed the full text of the papers and extracted the information according to predefined research question dimensions and conducted an initial classification. Some challenges arise where certain data items are missing, poorly defined, or require inference from context. Such instances are flagged for further investigation in subsequent iterations.

In the second iteration, authors cross-reference and merge their individual data summaries into an integrated table. Unclear data items are revisited and resolved through discussions and further research together. For instance, a category labeled “check together” is added for the evaluation procedure due to a lack of justification in some studies, requiring additional research to determine the subject system domains and ensure that their interpretations of the different dimensions and scales were aligned.

In the third iteration, the first author proceeded to read and extract the information from the remaining papers at a flexible yet systematic pace to maintain methodological rigor and held weekly meetings with second author to resolve any arising issues. Once the information was extracted from all papers, the evaluators met again to homogenize the vocabulary used for the open scales following an open card sorting process [20]. For each open scale, the evaluators listed and consolidated the information extracted from the papers. For instance, for the scale “goal of research”, we consolidated the extracted terms “bug prediction”, “just-in-time defect prediction and localization” under the common term “bug identification and localization”. Once the exhaustive list of values for each scale was established, the first author applied them to all papers. For certain open scales, we further analyzed recurring patterns in the data, grouping them into distinct themes to provide additional depth and structure to the analysis. For instance, for the scale “Type of source code artifacts used”, we grouped all values related to source code artifacts in version control system (e.g., code changes, issue tracking, vulnerability information from security advisory, etc).

Finally, the results are summarized into integrated table statistics, count the number of studies for each technique associated with data items, and group similar techniques into broader categories. Both authors participate in comprehensive discussions to finalize data classifications for the study. These classifications and patterns

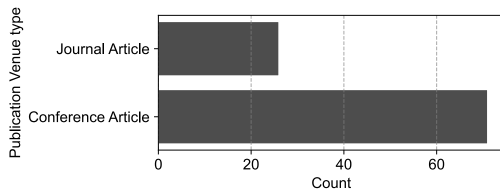

Figure 4: Publication venue types.

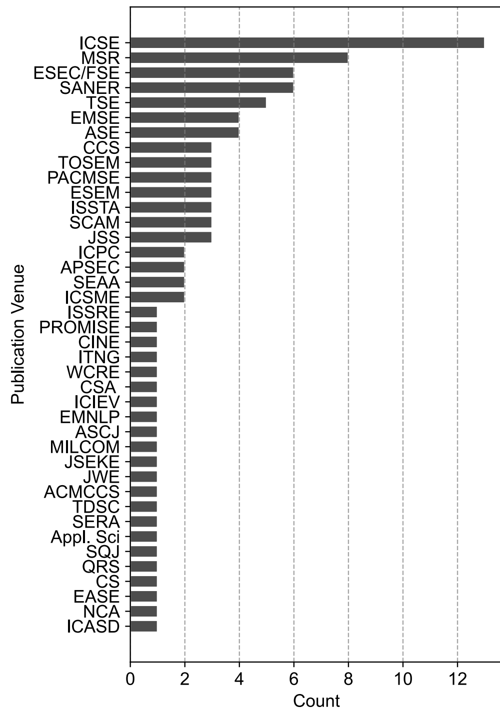

Figure 5: Publication venues.

are further discussed in Section 3, providing a comprehensive interpretation of the findings.

3 Results

In this section, we review and analyze the information extracted from 97 primary sources following the scales and dimensions identified for each research question in Table 2. For each dimension, the number of studies matching it is indicated in parentheses. The raw data, analysis scripts, and plots discussed in this section are available from the reproducibility package [6].

3.1 RQ_1 *Where and when was the study published?*

Year of publication. Figure 3 shows the analyzed primary sources span over two decades, from 2004 to May 2025. To analyze publication trend, we used a two-year span instead of a single year because the publication process commonly takes multiple years, making one-year counts an unreliable measure of output. Thus, our trend

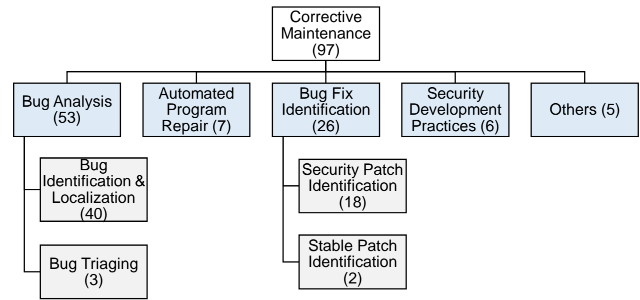

Figure 6: Taxonomy of research goals where the goal categories are clustered under four major themes (plus Others).

analysis indicates that the number of publications increases over time, with notable peaks in 2022-2023 and a decline in 2024–May 2025. The latter decrease is common in secondary studies and is most likely due to incomplete indexing of recent publications by the bibliographic database.

Overall, these observations suggest a growing and sustained research interest of exploiting commit messages for corrective software maintenance: the topic remains active and evolving.

Venue. Publication venues are quite varied, reflecting the nature of research on corrective maintenance and the shared interests of various communities. As shown in Figure 4, the research community published their works more through conferences (71) than journals (26). While research papers are published across a wide range of venues, most works are published in leading software engineering international conferences like ICSE (13), MSR (8), ESEC/FSE (6), and SANER (6), as well as in the top journals like TSE (5), EMSE (4), TOSEM (3), and JSS (3) reflecting a particularly strong contribution from the empirical software engineering community as shown in Figure 5. However, other venues appear only once or twice in the dataset, showcasing the broad distribution of studies throughout the publication landscape. This suggests that using commit messages for corrective maintenance is highly relevant to software engineering domains and concentrated in top conferences and journals.

3.2 RQ_2 *What were the goals of the study?*

Since our focus is on the use of commit messages for corrective maintenance, it is essential to understand which goals researchers pursue with them. In this research question, we particularly review the specific goals of executing research on corrective maintenance. Our analysis led us to identify eight categories of corrective maintenance activities as shown in Table 3.

Figure 6 illustrates the frequency of papers related to each corrective software maintenance category. We observed that the goals motivating to undertake research activities can be clustered under four major themes: bug analysis (53), automated program repair (7), bug fix identification (26), and security development practices (6). Among these, the bug analysis category comprises two subcategories—bug identification and localization, and bug triaging—while the bug fix identification category includes subcategories security patch identification and stable patch identification. The majority of the papers fall under bug identification and localization

Table 3: Research goal categories and their relevant papers from primary source.

Categories	Goals	List of Papers
Bug analysis	Analysis of compiler bug propagation, dormant bugs vs non-dormant bugs, atom of confusion, reverted commits, security vulnerabilities, root causes of program bug, defect proneness, relationships between stability and bug-proneness of clones, correlation between commits social characteristics and bugginess, etc.	[30, 35, 41, 58, 76, 77, 98, 103, 109, 110]
Bug identification and localization	Just-in-time defect prediction and localization, vulnerable dependency prediction, impact of tangled code changes on defect prediction models. In addition, this category also include optimization related topics such as investigation of “unfair, imbalanced” datasets effect on prediction performance, impact of misclassifications of bug reports in prediction, quality of attributes in predicting defect, heuristic to improve fault prediction, improve tracing of defects through new SZZ algorithm, evaluate commits that introduce bug, searching code that is semantically identical to given buggy code and assessing practitioner beliefs about software defect prediction, etc.	[21, 29, 31–34, 36, 38, 40, 45–50, 56, 57, 59, 60, 62, 63, 71, 74, 75, 78, 84–87, 89, 90, 92, 93, 101, 104, 106, 108, 111, 112, 116]
Bug triaging	Assessing bug report and recommending the most appropriate developer for bug resolution.	[23, 26, 53]
Automated program repair	Improving patch quality and correctness, vulnerability comprehension in automated program repair and thereby reducing manual efforts.	[22, 28, 44, 66, 94, 96, 97]
Bug fix identification	Identifying commits that fix bugs. In addition, this category also include optimization related topics such as improving classification of bug fix commits, analyze data quality that have influence on bug fixing process, relation of bug fix and commit size, etc.	[24, 27, 67, 70, 95, 105]
Security patch identification	Security patch identification/localization, hidden security fix pattern identification, commits that fix bugs with security implications and alert users to silent fixes, characterize the nature of security fixes in comparison to other non-security bug fixes, etc.	[42, 52, 61, 72, 73, 81–83, 88, 91, 99, 100, 102, 107, 113–115, 117]
Stable patch identification	Identifying stable patches that can be used for long term.	[64, 65]
Security development practice [Human Aspect]	Understanding why developers introduce vulnerabilities, determining where developers document security concerns, informativeness of security commit message, defining systematic scheme for identifying security vocabularies, creation of a convention for security commit messages that structure and integrate information about vulnerabilities.	[25, 43, 68, 69, 79, 80]
Others	Goals that do not fit above categories such as understanding relation between developer sentiment and software bugs, dependency management practice in response to bug, etc.	[37, 39, 51, 54, 55]

(40), revealing a strong emphasis on enhancing software reliability through early stage bug identification during software development. The next dominant subcategory is security patch identification (18), which aims to identify hidden security fixes and alert users about their nature. However, other categories such as automated program repair (7), security development practices (6), bug triaging (3), and stable patch identification (2) received relatively less attention compared to the dominant categories.

3.3 *RQ$_3$ What software artifacts were used in the study?*

Our analysis of what software artifacts were used in the reviewed studies, together with commit messages, identified 11 types of artifacts:

- *Code changes:* includes diff/patch/code changes of a commit, lines of code (LOC)/churn per commit within diff, etc.
- *Issue tracking:* includes information extracted from issue/bug tracking systems, such as GitHub issues, Jira tickets, Bugzilla reports, etc.
- *Developer metrics:* includes number of commits by author in past, number of files committed by commit author in the past, co-changed files, number of switches in projects by the author, ownership of files, work experience, and membership status such as owner, subscriber, watcher, stranger, etc.
- *Vulnerability information:* includes information about security vulnerabilities, in general from security advisories such as NVD, SNYK, GitHub, GitLab, reserved CVEs, etc.
- *Pull requests:* includes data and metadata from pull and merge requests such as description, discussions, status, code changes (even when not merged in the target repository at the end), etc.
- *Developer discussions:* includes content from question and answer web sites like stack overflow, mailing list, etc.
- *Community feedback:* includes feedback from specific community through surveys, and in some cases also in-person workshops.
- *Release notes:* includes information extracted from the release notes of specific software versions or packages.
- *Code comments:* includes code comments extracted from source code.

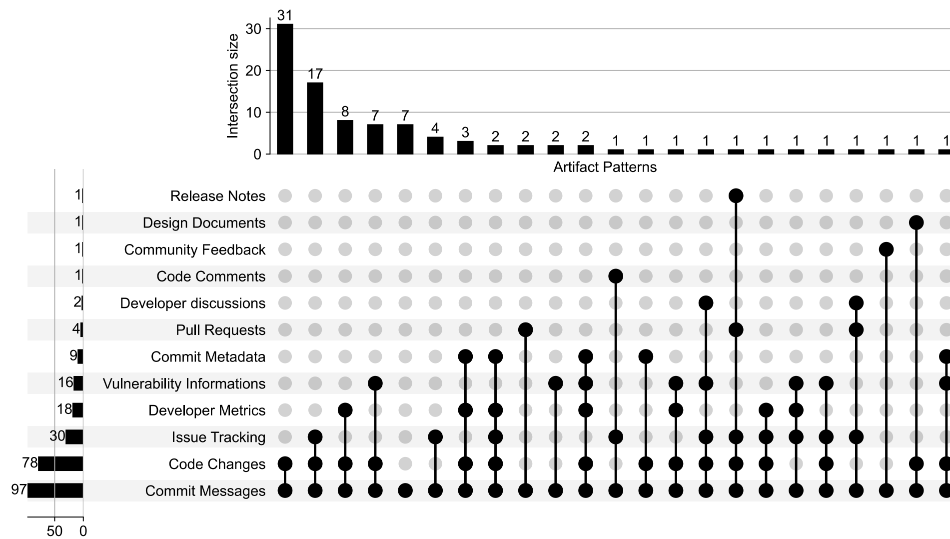

Figure 7: Frequency and co-occurrence of software artifacts used to support corrective software maintenance activities.

- *Design documents:* includes team members document on design decision.
- *Commit metadata:* includes commit information other than messages and diffs (already covered in other categories above), such as timestamps, commit hash identifiers, authors, etc.

As shown in Figure 7, code changes (78) artifacts emerged as the most used, followed by issue tracking information (30), developer metrics (18), and vulnerability information (16). This dominance suggests that code changes play a central role in shaping the observed research patterns in artifact usage while executing corrective maintenance related studies—together with commit messages, of course. Artifact issue tracking contents, while less prevalent than code changes, frequently co-occur with code changes, indicating a potential interdependence between the two (and commit messages). Artifact developer metrics and vulnerability information from security advisories, though less dominant overall, appear to complement the roles of both commit messages and code changes in specific contexts. The most frequently observed combination pattern within the dataset is {commit message, code changes} followed by {commit message, code changes, issue tracking}, which recurred across most patterns as shown in Figure 7. This recurring pattern highlights the significance of their joint application in achieving desired research goals in corrective maintenance tasks. Conversely, the isolated use of commit messages appeared less common, implying a lower effectiveness in supporting corrective maintenance activities by themselves. Artifacts such as code comments and design documents were observed less frequently and often appeared in specialized scenarios, indicating their evolving relevance within the broader framework.

3.4 RQ_4 *What research methodology was used in the study?*

As shown in Figure 8, our analysis revealed six primary methodology types: repository mining, NLP, AI/ML, manual inspection, community feedback, and controlled experiment. Among these, repository mining (89) emerged as the most dominant, followed by NLP (82) and AI/ML (72) techniques. The prevalence of repository mining indicates its broad applicability and perceived effectiveness across corrective maintenance related research settings. NLP and AI/ML techniques often appeared in conjunction with repository mining, suggesting their combined use provides complementary strengths and enhances the overall rigor of research outcomes. Manual inspection, while less frequent, was typically employed in studies requiring deeper analytical or multi-dimensional investigation on data. The combination { Repository mining, NLP, AI/ML} is the most common methodological pattern, reinforcing the view that integrating these approaches yields more comprehensive insights. In contrast, community feedback and controlled experiments were applied less frequently, reflecting their limited usage for corrective maintenance research contexts. Overall, the observed patterns collectively highlight a structured hierarchy, where repository mining serves as the methodological foundation, often supported by NLP and AI/ML, occasionally complemented by manual inspection.

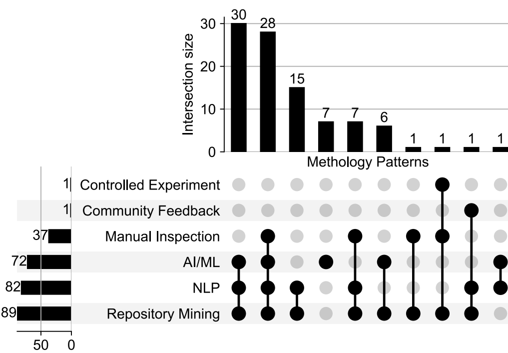

Figure 8: Methodology types and their usage patterns.

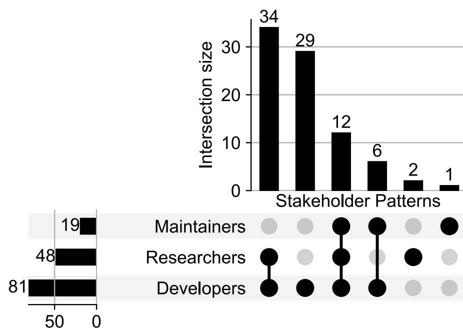

Figure 9: Stakeholder types and their mention patterns.

3.5 *RQ_5 Which human stakeholders' perspectives were considered in the study?*

Our analysis revealed three primary stakeholder types explicitly mentioned across the reviewed primary studies: developer, maintainer and researcher. Below, we define each stakeholder similar to previous work [53] as follows:

- *Developer:* is mainly responsible for building software through development activities such as writing and modifying source code, fixing bug, testing components, etc. In addition, handling security responsibilities such as implementing security controls are integrated part of development role [17]. Therefore, security engineers and developers are considered as the same person and used interchangeably.
- *Maintainer:* is mainly responsible for the governance of software project such as crucial decision making, reviewing contributions and merging, bug triaging, ensure code quality and consistency, etc.
- *Researcher:* is someone who investigates software-related phenomena using scientific methods to produce new knowledge, tools, or validated theories.

As shown in Figure 9, developers (81) appeared most frequently, indicating their central role in corrective software maintenance. Researchers (48) are next, often considered alongside developers, suggesting the importance of applied research and collaboration among researchers and practitioners. Maintainers (19) were considered last, typically in studies emphasizing stability and security-focused perspectives, indicating their responsibility for maintaining system security and compliance. Overall, the findings indicate clear dominance of developers, underscoring how attention is often given to stakeholders who play the most critical role in shaping decisions and improving corrective maintenance practices.

3.6 *RQ_6 What were the key findings of the study, about how commit messages influence corrective software maintenance?*

The key findings of the previous studies about how commit messages influence corrective maintenance are mapped into four major themes. These are explained as follows.

In bug analysis context, commit messages play a vital role in tasks such as bug analysis, prediction, and triaging. Previous studies reported that commit messages convey important semantic information that helps machine learning models to better understand code changes and improve Just-in-time defect prediction accuracy [21, 57]. Integrating commit messages with code change enhances bug detection performance, whereas the absence of either commit messages or code changes significantly reduces model accuracy. In addition, detailed and high-quality messages allow models to identify bugs more efficiently, while short or noisy ones negatively impact prediction outcomes [30, 38, 45, 63, 87, 89, 106, 108]. Moreover, studies on bug analysis also reported that commit messages often lack sufficient details [33, 76]. In bug triaging, the linguistic features of commit messages align closely with bug reports, helping to assign issues to the right developers and improving triaging accuracy [23, 26, 53]. Overall, informative and structured commit messages enhance both prediction accuracy and the interpretability of automated tools in managing software bugs.

In automated program repair context, commit messages contribute valuable insights into the intent and context behind code changes. Well-structured commit messages help large language models (LLMs) learn to produce better patches, effectively linking natural language description with code semantics [22, 94]. Studies on automated program repair also reported that commit message quality correlates with correct patch identification and recurring linguistic patterns within commit messages provide features that improve automated analysis and repair [44]. Overall, high quality commit messages not only document developer intent but also act as a rich data source to strengthen the precision and interpretability of automated program repairing techniques.

In bug fix identification context, commit messages are essential, but many of them lack informativeness, limiting the automated detection accuracy [24, 105, 107]. Commit messages that are clear and descriptive, with fix-related keywords or issue references, enable tools to distinguish bug-fix commits from regular updates and support efficient maintenance tracking [27, 70]. For security patch identification, well-crafted messages containing security-related keywords such as CVE ids combined with code diffs are especially useful in recognizing vulnerability-fixing commits and significantly improve model accuracy [72, 73, 88, 113, 117]. Overall, precise and detailed commit messages with specific vocabulary enhance the automation and reliability of identifying bug fix related commits across various contexts.

Security development practice related studies, reported that commit messages often fail to provide sufficient security-related details, limiting their utility for vulnerability assessment [68, 80]. Most commit messages lack explicit references to security issues, relying on implicit domain vocabulary instead [69]. Well-structured and informative messages that explicitly describe security implications can facilitate automated vulnerability detection and human understanding [43, 79]. Beyond security, other studies show that commit messages capture developer sentiment, and rationale for code changes, which can influence software quality and team collaboration [54, 55]. Overall, improving the linguistic richness, structure, and developers apparent sentiment (as detectable by sentiment analysis) in commit messages can strengthen both technical and human aspects of software development.

4 Recommendations

Building on the results above, we present key recommendations for stakeholders (such as developers, maintainers, and researchers) to improve commit message quality and, eventually, corrective software maintenance.

Developers are recommended to follow strict rules and lexicons when writing commit messages, ensuring semantic alignment with code changes [34, 45, 106]. In addition, their written messages should preserve essential natural-language patterns and technical terminologies, as they serve as a bridge between code-level changes and natural language understanding [44]. For bug fix identification purposes, developers are recommended to include explicit rationale and contextual explanations in commit messages to make maintenance more efficient [27]. Besides, keeping commit messages concise with separate commit-info, commit-subject, and commit-body can improve automated understanding and accuracy of patch identification [115]. For better security patch identification, developers should avoid hiding the nature of security vulnerability in commit message [73]. Besides, messages should clearly describe the type of security issue addressed by including references (like CVE or bug IDs) to facilitate easier linking of fixes [91, 99, 117]. For developers, these recommendations can help understand necessary technical knowledge to write better commit messages and supporting later corrective software maintenance.

Maintainers should be conscious that the quality of commit messages will directly impact later corrective maintenance tasks. When starting a new project, they should provide explicit guidelines and possible explanations (e.g., by providing good examples and better documentation) to write commit messages, ensuring semantic alignment with code changes. For instance, a univocal language can be used to identify the type of operation committed by developers, such as bug fix, refactoring, etc. [34, 45, 106]. Recent conventions about how to write commits, like Conventional Commits [1], can help in this respect. In addition, maintainers should be proactive in reviewing commits made by newbies carefully and ensure that code changes are accompanied by well-structured and detailed messages [89]. For maintainers, such understandings help support cost-effective maintenance and evolution of their software projects.

Researchers should standardize templates, official guidelines, and best practices for writing commit messages, as these make it easier for automated tools and human analysts to detect bugs [79]. They should encourage developers to adopt best development practice (such as security), through highlighting how informative commit messages with structured and consistent formatting can facilitate bug assessment [80]. For instance, a study by [69] suggested that combining standard security keywords (e.g., "encryption," "authentication") with project-specific vocabulary further improves precision in identifying security-relevant changes. In building automated classification, researchers are advised to be careful during pre-processing of commit message in order to avoid overlooking acronyms such as VRSN (version), RLS (release), etc [70]. To further improve bug classification performance and reduce the false positive rate, researchers should focus on enriching commit messages with supplementary information, such as test data, issue tracking contents, etc [48]. Moreover, researchers should investigate possible ways to reduce manual verification effort, as previous studies further suggest that even when automated classifiers identify security commits, manual expert verification remains essential for ensuring precision [81]. Beyond these, other studies recommend that researchers should also explore the sentiment and emotional cues in commit messages to understand developer performance and team collaboration dynamics [54].

5 Threats to validity

Like all secondary studies, threats to validity can be raised by the research methodology applied by this systematic mapping study. We have mitigated such threats by following the systematic review protocol proposed by Wohlin et al. [18].

Threats to internal validity may be introduced by having inappropriate classification and interpretation of the studies. We have limited this by conducting three iterations of paper reviews by the two authors collaboratively. In addition, unclear data items are revisited and resolved through discussions and further research during the search.

Threats to construct validity may be introduced by inadequate capture of relevant literature for the target study. Our initial search string does not include *specific* terms related to corrective maintenance (e.g., "defect", "repair", "bug", etc.), but rather retrieves studies broadly related to commit messages. This choice was due to the difficulty of being comprehensive with specific terms; short of which the initial seed of papers might have been imbalanced, favoring some concepts over others. In addition, we do not consider the popularity of conference and journal venues as a selection criterion, which may result in the inclusion of a few studies from less widely recognized venues; on the other hand, this choice protects against venue-related biases.

Threats to external validity may restrict the generalizability of the revealed statistics and the justification of future research opportunities. Since our study was focused on peer-reviewed publications, some relevant industrial or unpublished work (e.g., gray literature) may have been missed. Furthermore, despite leveraging multiple digital libraries and following a rigorous protocol to craft search queries, we might still have missed relevant studies. The inclusion of a backward snowballing step mitigates this risk to some extent, but complete coverage is never guaranteed in any systematic review of the literature. This, together with evidence from best practices in corrective software maintenance, has provided rich sources for us to discuss the results and the future research opportunities identified.

6 Conclusion

In this paper, we conducted a systematic mapping study on the usage of commit messages in corrective software maintenance accompanied by a meta-analysis to answer our research questions. After a comprehensive search that follows a systematic series of steps and assessing the quality of the studies, 97 papers were identified.

Based on the data extracted from these selected papers, we derived a comprehensive synthesis on the state-of-the-art of the use of commit messages for corrective software maintenance tasks. We observed a consistent and growing interest in the usage of commit messages to resolve issues of corrective maintenance over the last two decades, with a marked increase in 2022-2023. Although

bug analysis is the dominant goal, relatively few studies address other themes such as automated program repair, security development practices, etc. We have observed {commit messages, code changes} to be the most dominant artifact co-usage pattern. In terms of methodologies: repository mining, followed by NLP, and AI/ML techniques are the most popular ones. Our analysis demonstrates that commit messages play a significant role in corrective maintenance activities since they carry crucial information helpful for stakeholders (i.e., developer, maintainer, and researcher) to understand and improve the code base through the software evolution process. However, commit messages often lack necessary information and are not always sufficient to represent the intent of code changes.

Among the gaps, we have observed in the literature, three stand out: (1) the need to close the gap between stated goals of corrective maintenance and stakeholders' needs; (2) a thorough study to investigate the trade-offs between benefits and drawbacks of following official guidelines in writing commit messages, to identify what an ideal project VCS history—including commit messages—would look like; (3) interviewing or surveying stakeholders to further allow for triangulation of our findings to understand the impact of commit message quality on corrective maintenance tasks.

Data availability

A complete reproducibility package for the work described in this paper is publicly archived and available from Zenodo [6].

Acknowledgments

This work was supported by France Agence Nationale de la Recherche (ANR), program France 2030, reference ANR-22-PTCC-0001. The authors would like to thank Théo Zimmermann for insightful discussions and comments on an earlier version of this paper.

References

[1] [n. d.]. Conventional Commits. https://www.conventionalcommits.org/en/v1.0.0/. Last Accessed: December 10, 2025.

[2] Maria Alaranta and Stefanie Betz. 2012. Knowledge Problems in Corrective Software Maintenance–A Case Study. In *2012 45th Hawaii International Conference on System Sciences*. IEEE, 3746–3755.

[3] Stefano Balla, Thomas Degueule, Romain Robbes, Jean-Rémy Falleri, and Stefano Zacchiroli. 2025. Automatic Classification of Software Repositories: a Systematic Mapping Study. In *International Conference on Evaluation and Assessment in Software Engineering (EASE 2025)*.

[4] Shariq Aziz Butt, Acosta-Coll Melisa, and Sanjay Misra. 2022. Software product maintenance: A case study. In *International Conference on Computer Information Systems and Industrial Management*. Springer, 81–92.

[5] Gerardo Canfora and Aniello Cimitile. 2001. Software maintenance. In *Handbook of Software Engineering and Knowledge Engineering: Volume I: Fundamentals*. World Scientific, 91–120.

[6] Syful Islam and Stefano Zacchiroli. 2026. Reproducibility package for: On the Use of Commit Messages for Corrective Software Maintenance: A Systematic Mapping Study. https://doi.org/10.5281/zenodo.18324248. Last Accessed: March 16, 2026.

[7] Barbara Kitchenham, O Pearl Brereton, David Budgen, Mark Turner, John Bailey, and Stephen Linkman. 2009. Systematic literature reviews in software engineering–a systematic literature review. *Information and software technology* 51, 1 (2009), 7–15.

[8] Jiawei Li and Iftekhar Ahmed. 2023. Commit message matters: Investigating impact and evolution of commit message quality. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 806–817.

[9] James Martin and Carma L McClure. 1983. *Software Maintenance: The Problems and Its Solutions.* Prentice Hall Professional Technical Reference.

[10] Mohammad Mehdi Morovati, Amin Nikanjam, Florian Tambon, Foutse Khomh, and Zhen Ming Jiang. 2024. Bug characterization in machine learning-based systems. *Empirical Software Engineering* 29, 1 (2024), 14.

[11] Cliodhna O'Connor and Helene Joffe. 2020. Intercoder reliability in qualitative research: Debates and practical guidelines. *International journal of qualitative methods* 19 (2020), 1609406919899220.

[12] Kai Petersen, Robert Feldt, Shahid Mujtaba, and Michael Mattsson. 2008. Systematic mapping studies in software engineering. In *12th international conference on evaluation and assessment in software engineering (EASE)*. BCS Learning & Development.

[13] Kai Petersen, Sairam Vakkalanka, and Ludwik Kuzniarz. 2015. Guidelines for conducting systematic mapping studies in software engineering: An update. *Information and software technology* 64 (2015), 1–18.

[14] E Burton Swanson. 1976. The dimensions of maintenance. In *Proceedings of the 2nd international conference on Software engineering*. 492–497.

[15] Yingchen Tian, Yuxia Zhang, Klaas-Jan Stol, Lin Jiang, and Hui Liu. 2022. What makes a good commit message?. In *Proceedings of the 44th International Conference on Software Engineering*. 2389–2401.

[16] A Marie Vans, Anneliese von Mayrhauser, and Gabriel Somlo. 1999. Program understanding behavior during corrective maintenance of large-scale software. *International Journal of Human-Computer Studies* 51, 1 (1999), 31–70.

[17] Song Wang and Nachiappan Nagappan. 2021. Characterizing and understanding software developer networks in security development. In *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 534–545.

[18] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, Anders Wesslén, et al. 2012. *Experimentation in software engineering*. Vol. 236. Springer.

[19] Pengyu Xue, Linhao Wu, Zhongxing Yu, Zhi Jin, Zhen Yang, Xinyi Li, Zhenyu Yang, and Yue Tan. 2024. Automated commit message generation with large language models: An empirical study and beyond. *IEEE Transactions on Software Engineering* (2024).

[20] Thomas Zimmermann. 2016. Card-sorting: From text to themes. In *Perspectives on data science for software engineering*. Elsevier, 137–141.

Primary sources

[21] Manar Abu Talib, Ali Bou Nassif, Mohammad Azzeh, Yaser Alesh, and Yaman Afadar. 2024. Parameter-efficient fine-tuning of pre-trained code models for just-in-time defect prediction. *Neural Computing and Applications* 36, 27 (2024), 16911–16940.

[22] Toufique Ahmed and Premkumar Devanbu. 2023. Better patching using llm prompting, via self-consistency. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1742–1746.

[23] Bader Alkhazi, Andrew DiStasi, Wajdi Aljedaani, Hussein Alrubaye, Xin Ye, and Mohamed Wiem Mkaouer. 2020. Learning to rank developers for bug report assignment. *Applied Soft Computing* 95 (2020), 106667.

[24] Idan Amit and Dror G Feitelson. 2021. Corrective commit probability: a measure of the effort invested in bug fixing. *Software Quality Journal* 29, 4 (2021), 817–861.

[25] Gábor Antal, Márton Keleti, and Péter Hegedűs. 2020. Exploring the security awareness of the python and javascript open source communities. In *Proceedings of the 17th International Conference on Mining Software Repositories*. 16–20.

[26] Farhan Rahman Arnob, Rubel Hassan Mollik, Pooja Goyal, and Renee Bryce. 2025. Bug Triaging Based on Transformer Models Utilizing Commit Messages. In *International Conference on Information Technology-New Generations*. Springer, 354–366.

[27] Adrian Bachmann and Abraham Bernstein. 2010. When process data quality affects the number of bugs: Correlations in software engineering datasets. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*. IEEE, 62–71.

[28] Jiaqi Bai, Long Zhou, Ambrosio Blanco, Shujie Liu, Furu Wei, Ming Zhou, and Zhoujun Li. 2021. Jointly Learning to Repair Code and Generate Commit Message. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. 9784–9795.

[29] Lingfeng Bao, Xin Xia, Ahmed E Hassan, and Xiaohu Yang. 2022. V-SZZ: automatic identification of version ranges affected by CVE vulnerabilities. In *Proceedings of the 44th international conference on software engineering*. 2352–2364.

[30] Jacob G Barnett, Charles K Gathuru, Luke S Soldano, and Shane McIntosh. 2016. The relationship between commit message detail and defect proneness in java projects on github. In *Proceedings of the 13th International Conference on Mining Software Repositories*. 496–499.

[31] Pamela Bhattacharya, Marios Iliofotou, Iulian Neamtiu, and Michalis Faloutsos. 2012. Graph-based analysis and prediction for software evolution. In *2012 34th International conference on software engineering (ICSE)*. IEEE, 419–429.

[32] Christian Bird, Adrian Bachmann, Eirik Aune, John Duffy, Abraham Bernstein, Vladimir Filkov, and Premkumar Devanbu. 2009. Fair and balanced? bias in bug-fix datasets. In *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of*

Software Engineering. 121–130.

[33] Peter Bludau and Alexander Pretschner. 2022. PR-SZZ: How pull requests can support the tracing of defects in software repositories. In *2022 IEEE international conference on software analysis, evolution and reengineering (SANER)*. IEEE, 1–12.

[34] Maria Caulo and Giuseppe Scanniello. 2019. On the Use of Commit Messages to Support the Creation of Datasets for Fault Prediction: an Empirical Assessment. In *2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, 193–198.

[35] Tse-Hsun Chen, Meiyappan Nagappan, Emad Shihab, and Ahmed E Hassan. 2014. An empirical study of dormant bugs. In *Proceedings of the 11th Working Conference on Mining Software Repositories*. 82–91.

[36] Xiang Chen, Hongling Xia, Wenlong Pei, Chao Ni, and Ke Liu. 2023. Boosting multi-objective just-in-time software defect prediction by fusing expert metrics and semantic metrics. *Journal of Systems and Software* 206 (2023), 111853.

[37] Yang Chen, Andrew E Santosa, Ang Ming Yi, Abhishek Sharma, Asankhaya Sharma, and David Lo. 2020. A machine learning approach for vulnerability curation. In *Proceedings of the 17th International Conference on Mining Software Repositories*. 32–42.

[38] Jiwon Choi, Taeyoung Kim, Duksan Ryu, Jongmoon Baik, and Suntae Kim. 2023. Just-in-time defect prediction for self-driving software via a deep learning model. *Journal of Web Engineering* 22, 2 (2023), 303–326.

[39] Filipe Roseiro Cogo, Gustavo A Oliva, and Ahmed E Hassan. 2019. An empirical study of dependency downgrades in the npm ecosystem. *IEEE Transactions on Software Engineering* 47, 11 (2019), 2457–2470.

[40] Beyza Eken, RiFat Atar, Sahra Sertalp, and Ayşe Tosun. 2019. Predicting defects with latent and semantic features from commit logs in an industrial setting. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*. IEEE, 98–105.

[41] Jon Eyolfson, Lin Tan, and Patrick Lam. 2011. Do time of day and developer experience affect commit bugginess?. In *Proceedings of the 8th Working Conference on Mining Software Repositories*. 153–162.

[42] Nitzan Farhi, Noam Koenigstein, and Yuval Shavitt. 2025. PatchView: Multi-modality detection of security patches. *Computers & Security* 151 (2025), 104356.

[43] Kelsey R Fulton, Daniel Votipka, Desiree Abrokwa, Michelle L Mazurek, Michael Hicks, and James Parker. 2022. Understanding the how and the why: Exploring secure development practices through a course competition. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 1141–1155.

[44] Amiao Gao, Zenong Zhang, Simin Wang, Liguo Huang, Shiyi Wei, and Vincent Ng. 2025. Teaching AI the Why and How of Software Vulnerability Fixes. *Proceedings of the ACM on Software Engineering* 2, FSE (2025), 2006–2029.

[45] Yuxiang Guo, Xiaopeng Gao, Zhenyu Zhang, Wing Kwong Chan, and Bo Jiang. 2023. A study on the impact of pre-trained model on Just-In-Time defect prediction. In *2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security (QRS)*. IEEE, 105–116.

[46] Zhaoqiang Guo, Shiran Liu, Xutong Liu, Wei Lai, Mingliang Ma, Xu Zhang, Chao Ni, Yibiao Yang, Yanhui Li, Lin Chen, et al. 2023. Code-line-level bugginess identification: How far have we come, and how far have we yet to go? *ACM Transactions on Software Engineering and Methodology* 32, 4 (2023), 1–55.

[47] Kim Herzig, Sascha Just, Andreas Rau, and Andreas Zeller. 2013. Predicting defects using change genealogies. In *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 118–127.

[48] Kim Herzig, Sascha Just, and Andreas Zeller. 2013. It's not a bug, it's a feature: how misclassification impacts bug prediction. In *2013 35th international conference on software engineering (ICSE)*. IEEE, 392–401.

[49] Kim Herzig, Sascha Just, and Andreas Zeller. 2016. The impact of tangled code changes on defect prediction models. *Empirical Software Engineering* 21, 2 (2016), 303–336.

[50] Thong Hoang, Hoa Khanh Dam, Yasutaka Kamei, David Lo, and Naoyasu Ubayashi. 2019. Deepjit: an end-to-end deep learning framework for just-in-time defect prediction. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 34–45.

[51] Thong Hoang, Hong Jin Kang, David Lo, and Julia Lawall. 2020. Cc2vec: Distributed representations of code changes. In *Proceedings of the ACM/IEEE 42nd international conference on software engineering*. 518–529.

[52] Thong Hoang, Julia Lawall, Yuan Tian, Richard J Oentaryo, and David Lo. 2019. Patchnet: Hierarchical deep learning-based stable patch identification for the linux kernel. *IEEE Transactions on Software Engineering* 47, 11 (2019), 2471–2486.

[53] Md Kamal Hossen, Huzefa Kagdi, and Denys Poshyvanyk. 2014. Amalgamating source code authors, maintainers, and change proneness to triage change requests. In *Proceedings of the 22nd International Conference on Program Comprehension*. 130–141.

[54] Syed Fatiul Huq, Ali Zafar Sadiq, and Kazi Sakib. 2019. Understanding the effect of developer sentiment on fix-inducing changes: An exploratory study on github pull requests. In *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 514–521.

[55] Syed Fatiul Huq, Ali Zafar Sadiq, and Kazi Sakib. 2020. Is developer sentiment related to software bugs: An exploratory study on github commits. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 527–531.

[56] Tian Jiang, Lin Tan, and Sunghun Kim. 2013. Personalized defect prediction. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Ieee, 279–289.

[57] Yuze Jiang, Beijun Shen, and Xiaodong Gu. 2025. Just-in-time software defect prediction via bi-modal change representation learning. *Journal of Systems and Software* 219 (2025), 112253.

[58] Matthieu Jimenez, Yves Le Traon, and Mike Papadakis. 2018. Enabling the continous analysis of security vulnerabilities with vuldata7. In *18th IEEE International Working Conference on Source Code Analysis and Manipulation*.

[59] Matthieu Jimenez, Renaud Rwemalika, Mike Papadakis, Federica Sarro, Yves Le Traon, and Mark Harman. 2019. The importance of accounting for real-world labelling when predicting software vulnerabilities. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 695–705.

[60] Sunghun Kim, E James Whitehead, and Yi Zhang. 2008. Classifying software changes: Clean or buggy? *IEEE Transactions on software engineering* 34, 2 (2008), 181–196.

[61] Frank Li and Vern Paxson. 2017. A large-scale empirical study of security patches. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2201–2215.

[62] Jingyue Li and Michael D Ernst. 2012. CBCD: Cloned buggy code detector. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 310–320.

[63] Jingyu Liu, Jun Ai, Minyan Lu, Jie Wang, and Haoxiang Shi. 2023. Semantic feature learning for software defect prediction from source code and external knowledge. *Journal of Systems and Software* 204 (2023), 111753.

[64] Rongkai Liu, Heyuan Shi, Shuning Liu, Chao Hu, Sisheng Li, Yuheng Shen, Runzhe Wang, Xiaohai Shi, and Yu Jiang. 2025. PatchScope: LLM-Enhanced Fine-Grained Stable Patch Classification for Linux Kernel. *Proceedings of the ACM on Software Engineering* 2, ISSTA (2025), 1513–1535.

[65] Rongkai Liu, Heyuan Shi, Yongchao Zhang, Runzhe Wang, Yuheng Shen, Yuao Chen, Jing Luo, Xiaohai Shi, Chao Hu, and Yu Jiang. 2024. PatchBert: Continuous stable patch identification for Linux kernel via pre-trained model fine-tuning. In *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 349–358.

[66] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. Coconut: combining context-aware neural translation models using ensemble for program repair. In *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*. 101–114.

[67] Maryam Marzban, Zahra Khoshmanesh, and Ashkan Sami. 2011. Cohesion between size of commit and type of commit. In *Computer Science and Convergence: CSA 2011 & WCC 2011 Proceedings*. Springer, 231–239.

[68] Moritz Mock, Thomas Forrer, and Barbara Russo. 2024. Where do developers admit their security-related concerns?. In *International Conference on Agile Software Development*. Springer, 189–195.

[69] Patrick Morrison, Tosin Daniel Oyetoyan, and Laurie Williams. 2018. Identifying security issues in software development: are keywords enough?. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. 426–427.

[70] Alessandro Murgia, Giulio Concas, Michele Marchesi, and Roberto Tonelli. 2010. A machine learning approach for text categorization of fixing-issue commits on CVS. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*. 1–10.

[71] K Muthukumaran, Abhinav Choudhary, and NL Bhanu Murthy. 2015. Mining GitHub for novel change metrics to predict buggy files in software systems. In *2015 International Conference on Computational Intelligence and Networks*. IEEE, 15–20.

[72] Truong Giang Nguyen, Thanh Le-Cong, Hong Jin Kang, Xuan-Bach D Le, and David Lo. 2022. Vulcurator: a vulnerability-fixing commit detector. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1726–1730.

[73] Giang Nguyen-Truong, Hong Jin Kang, David Lo, Abhishek Sharma, Andrew E Santosa, Asankhaya Sharma, and Ming Yi Ang. 2022. Hermes: Using commit-issue linking to detect vulnerability-fixing commits. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 51–62.

[74] Chao Ni, Wei Wang, Kaiwen Yang, Xin Xia, Kui Liu, and David Lo. 2022. The best of both worlds: integrating semantic features with expert features for defect prediction and localization. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 672–683.

[75] Henning Perl, Sergej Dechand, Matthew Smith, Daniel Arp, Fabian Yamaguchi, Konrad Rieck, Sascha Fahl, and Yasemin Acar. 2015. Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits. In *Proceedings of the 22nd ACM SIGSAC conference on computer and communications security*. 426–437.

[76] Oton Pinheiro, Lincoln Rocha, and Windson Viana. 2023. How they relate and leave: Understanding atoms of confusion in open-source java projects. In *2023 IEEE 23rd International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 119–130.
[77] Md Saidur Rahman and Chanchal K Roy. 2017. On the relationships between stability and bug-proneness of code clones: An empirical study. In *2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 131–140.
[78] Jacek Ratzinger, Harald Gall, and Martin Pinzger. 2007. Quality assessment based on attribute series of software evolution. In *14th Working Conference on Reverse Engineering (WCRE 2007)*. IEEE, 80–89.
[79] Sofia Reis, Rui Abreu, Hakan Erdogmus, and Corina Păsăreanu. 2022. SECOM: Towards a convention for security commit messages. In *Proceedings of the 19th International Conference on Mining Software Repositories*. 764–765.
[80] Sofia Reis, Rui Abreu, and Corina Pasareanu. 2023. Are security commit messages informative? Not enough!. In *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering*. 196–199.
[81] Antonino Sabetta and Michele Bezzi. 2018. A practical approach to the automatic classification of security-relevant commits. In *2018 IEEE International conference on software maintenance and evolution (ICSME)*. IEEE, 579–582.
[82] Arthur D Sawadogo, Tegawendé F Bissyandé, Naouel Moha, Kevin Allix, Jacques Klein, Li Li, and Yves Le Traon. 2022. SSPCatcher: Learning to catch security patches. *Empirical Software Engineering* 27, 6 (2022), 151.
[83] Kedi Shen, Yun Zhang, Lingfeng Bao, Zhiyuan Wan, Zhuorong Li, and Minghui Wu. 2023. Patchmatch: A tool for locating patches of open source project vulnerabilities. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 175–179.
[84] Shivkumar Shivaji, E James Whitehead, Ram Akella, and Sunghun Kim. 2012. Reducing features to improve code change-based bug prediction. *IEEE Transactions on Software Engineering* 39, 4 (2012), 552–569.
[85] NC Shrikanth and Tim Menzies. 2020. Assessing practitioner beliefs about software defect prediction. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*. 182–190.
[86] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. 2005. When do changes induce fixes? *ACM sigsoft software engineering notes* 30, 4 (2005), 1–5.
[87] Jiamou Sun, Zhenchang Xing, Qinghua Lu, Xiwei Xu, Liming Zhu, Thong Hoang, and Dehai Zhao. 2023. Silent vulnerable dependency alert prediction with vulnerability key aspect explanation. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 970–982.
[88] Shiyu Sun, Shu Wang, Xinda Wang, Yunlong Xing, Elisa Zhang, and Kun Sun. 2023. Exploring security commits in python. In *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 171–181.
[89] Sho Suzuki, Hirohisa Aman, Sousuke Amasaki, Tomoyuki Yokogawa, and Minoru Kawahara. 2017. An application of the pagerank algorithm to commit evaluation on git repository. In *2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, 380–383.
[90] Ming Tan, Lin Tan, Sashank Dara, and Caleb Mayeux. 2015. Online defect prediction for imbalanced data. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 2. IEEE, 99–108.
[91] Xin Tan, Yuan Zhang, Chenyuan Mi, Jiajun Cao, Kun Sun, Yifan Lin, and Min Yang. 2021. Locating the security patches for disclosed oss vulnerabilities with vulnerability-commit correlation ranking. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 3282–3299.
[92] Lingxiao Tang, Jiakun Liu, Zhongxin Liu, Xiaohu Yang, and Lingfeng Bao. 2025. LLM4SZZ: Enhancing SZZ Algorithm with Context-Enhanced Assessment on Large Language Models. *Proceedings of the ACM on Software Engineering* 2, ISSTA (2025), 343–365.
[93] Afsana Tasnim and Md Rayhanur Rahman. 2018. Inferring Bug Patterns for Detecting Bugs in JavaScript By Analyzing Abstract Syntax Tree. In *2018 Joint 7th International Conference on Informatics, Electronics & Vision (ICIEV) and 2018 2nd International Conference on Imaging, Vision & Pattern Recognition (icIVPR)*. IEEE, 503–507.
[94] Haoye Tian, Xunzhu Tang, Andrew Habib, Shangwen Wang, Kui Liu, Xin Xia, Jacques Klein, and Tegawendé F Bissyandé. 2022. Is this change the answer to that problem? correlating descriptions of bug and code changes for evaluating patch correctness. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–13.
[95] Yuan Tian, Julia Lawall, and David Lo. 2012. Identifying linux bug fixing patches. In *2012 34th international conference on software engineering (ICSE)*. IEEE, 386–396.
[96] Yuchi Tian and Baishakhi Ray. 2017. Automatically diagnosing and repairing error handling bugs in C. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*. 752–762.
[97] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2019. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 28, 4 (2019), 1–29.
[98] Neil Walkinshaw and Leandro Minku. 2018. Are 20% of files responsible for 80% of defects?. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. 1–10.
[99] Shichao Wang, Yun Zhang, Liagfeng Bao, Xin Xia, and Minghui Wu. 2022. Vc-match: a ranking-based approach for automatic security patches localization for OSS vulnerabilities. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 589–600.
[100] Xinda Wang, Shu Wang, Pengbin Feng, Kun Sun, Sushil Jajodia, Sanae Benchaaboun, and Frank Geck. 2021. Patchrnn: A deep learning-based system for security patch identification. In *MILCOM 2021-2021 IEEE Military Communications Conference (MILCOM)*. IEEE, 595–600.
[101] Supatsara Wattanakriengkrai, Patanamon Thongtanunam, Chakkrit Tantithamthavorn, Hideaki Hata, and Kenichi Matsumoto. 2020. Predicting defective lines using a model-agnostic technique. *IEEE Transactions on Software Engineering* 48, 5 (2020), 1480–1496.
[102] Bozhi Wu, Shangqing Liu, Ruitao Feng, Xiaofei Xie, Jingkai Siow, and Shang-Wei Lin. 2022. Enhancing security patch identification by capturing structures in commits. *IEEE Transactions on Dependable and Secure Computing* (2022).
[103] Meng Yan, Xin Xia, David Lo, Ahmed E Hassan, and Shanping Li. 2019. Characterizing and identifying reverted commits. *Empirical Software Engineering* 24, 4 (2019), 2171–2208.
[104] Klaus Changsun Youm, June Ahn, Jeongho Kim, and Eunseok Lee. 2015. Bug localization based on code change histories and bug reports. In *2015 Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 190–197.
[105] Sarim Zafar, Muhammad Zubair Malik, and Gursimran Singh Walia. 2019. Towards standardizing and improving classification of bug-fix commits. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 1–6.
[106] Zhengran Zeng, Yuqun Zhang, Haotian Zhang, and Lingming Zhang. 2021. Deep just-in-time defect prediction: how far are we?. In *Proceedings of the 30th ACM SIGSOFT international symposium on software testing and analysis*. 427–438.
[107] Junwei Zhang, Xing Hu, Lingfeng Bao, Xin Xia, and Shanping Li. 2024. Dual Prompt-Based Few-Shot Learning for Automated Vulnerability Patch Localization. In *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 940–951.
[108] Tao Zhang, Wenjun Hu, Xiapu Luo, and Xiaobo Ma. 2019. A commit messages-based bug localization for android applications. *International Journal of Software Engineering and Knowledge Engineering* 29, 04 (2019), 457–487.
[109] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. 2018. An empirical study on tensorflow program bugs. In *Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis*. 129–140.
[110] Hao Zhong. 2025. Understanding compiler bugs in real development. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 605–605.
[111] Xin Zhou, DongGyun Han, and David Lo. 2022. Simple or complex? together for a more accurate just-in-time defect predictor. In *Proceedings of the 30th IEEE/ACM international conference on program comprehension*. 229–240.
[112] Xin Zhou, DongGyun Han, and David Lo. 2025. Bridging expert knowledge with deep learning techniques for just-in-time defect prediction. *Empirical Software Engineering* 30, 1 (2025), 37.
[113] Xin Zhou, Jianmin Pang, Zheng Shan, Feng Yue, Fudong Liu, Jinlong Xu, Junchao Wang, Wenfu Liu, and Guangming Liu. 2023. Tmvdpatch: a trusted multi-view decision system for security patch identification. *Applied Sciences* 13, 6 (2023), 3938.
[114] Yaqin Zhou and Asankhaya Sharma. 2017. Automated identification of security issues from commit messages and bug reports. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*. 914–919.
[115] Yaqin Zhou, Jing Kai Siow, Chenyu Wang, Shangqing Liu, and Yang Liu. 2021. Spi: Automated identification of security patches via commits. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 1 (2021), 1–27.
[116] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. 2007. Predicting defects for eclipse. In *Third international workshop on predictor models in software engineering (PROMISE'07: ICSE workshops 2007)*. IEEE, 9–9.
[117] Fei Zuo, Xin Zhang, Yuqi Song, Junghwan Rhee, and Jicheng Fu. 2023. Commit message can help: security patch detection in open source software via transformer. In *2023 IEEE/ACIS 21st International Conference on Software Engineering Research, Management and Applications (SERA)*. IEEE, 345–351.