The Journal of Systems and Software 219 (2025) 112230

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Information needs in bug reports for web applications☆

Di Wang*, Matthias Galster, Miguel Morales-Trujillo

University of Canterbury, Christchurch, New Zealand

ARTICLE INFO

Keywords:
Bug reproduction
Bug reports
Bug report evolution
Web applications

ABSTRACT

Given the widespread popularity and increasing reliance on long-lived web applications (such as Netflix and Facebook), effective and efficient bug reproduction is essential to maintain functionality and user satisfaction throughout the application's lifetime. Developers use bug reports to localize, reproduce, and eventually fix software bugs. However, the content of bug reports is not always helpful (e.g., due to incomplete or missing information). In this study, we explore what type of information is often missing in bug reports and how that information is presented in them. We manually analyzed the initial and final versions of 1000 bug reports from 10 popular open-source web-based applications. The analysis revealed that, regardless of the type of software (e.g., e-commerce software or personal tools), diagnostic suggestions from developers and end-user usage information are often missing in initial bug reports but only added later throughout the lifetime of a bug report. Also, textual descriptions and screenshots are used most to describe bugs, regardless of the type of bug (e.g., a functional or performance error). The study highlighted the need for improved bug reporting templates and tools to improve bug report quality and efficiency in web application development and maintenance.

1. Introduction

1.1. Background and problem

Web-based applications are systems deployed and used via the internet (e.g., as web or cloud services). Users typically interact with this type of application via a website. Web-based applications are a de-facto standard for "end-user oriented" software (Muzaki et al., 2020). Like other types of software (e.g., embedded systems or mobile application), web-based applications accumulate bugs that need to be fixed during development and maintenance. Here, we use "bug" as an umbrella concept for failure (a software service deviates from its correct behavior), error (a condition in a system that may lead to a failure), and fault (the underlying cause of an error in the source code) (Avizienis et al., 2004).

When fixing bugs, developers typically try to *reproduce* them first (Johnson et al., 2022). Bug reproduction is the process of identifying and verifying the existence of a bug in the software by recreating the conditions under which the bug occurred (Vyas et al., 2014). This requires a developer to replicate the scenario that caused the bug to appear and then determine how the bug affects the software. Recently, most bug reproduction studies have focused on mobile applications (Fazzini et al., 2022; Bhattacharya et al., 2013), while the reproduction of bugs in web applications has not been as thoroughly explored. As web technologies evolve, new challenges for bug reproduction in web applications have emerged that require further investigation. Given the widespread use of web applications, the findings from such studies would be highly relevant and have significant practical implications for modern web application development.

In this context, a bug report is a document (e.g., a record in an issue or bug tracker) about what went wrong and needs to be fixed in the application. Reproducing a bug can be difficult, especially when bug reports lack sufficient details (Zimmermann et al., 2012; Davies and Roper, 2014; Zhang et al., 2017). Developers have to spend a significant amount of time and effort to understand the bugs based on the information in a bug report or follow up with those who reported the bug to provide additional details (Karim et al., 2017; Erfani Joorabchi et al., 2014). For this reason, accurate and complete bug reports are essential to provide developers with the information they need to understand and reproduce the bug (Johnson et al., 2022).

1.2. Research questions

Well-written bug reports increase the likelihood of a bug being reproduced (Zimmermann et al., 2010). Therefore, in this paper, our goal is **to understand better what makes good bug reports for web-based applications and suggest ways to improve the information**

☆ Editor: Dr Alexander Chatzigeorgiou.
* Corresponding author.
E-mail addresses: di.wang@pg.canterbury.ac.nz (D. Wang), mgalster@ieee.org (M. Galster), miguel.morales@canterbury.ac.nz (M. Morales-Trujillo).

https://doi.org/10.1016/j.jss.2024.112230
Received 17 April 2024; Received in revised form 23 August 2024; Accepted 24 September 2024
Available online 5 October 2024

provided in bug reports. In detail, we ask the following research questions exploring *what* information is required but might be missing in bug reports (RQ1 and RQ2) and *how* that information is provided and communicated in bug reports (RQ3, RQ4, and RQ5):

- **RQ1: What types of information are frequently missing in bug reports?** Incomplete or insufficient information in bug reports can hinder reproducibility and make it more difficult for developers to fix the bug. For example, information about how exactly an end user used the software when the bug occurred or error logs for technical details about the state of the software when the bug occurred play an important role in bug reproduction. Understanding missing information in initial bug reports, which is added later during the evolution of a bug report, can lead to recommendations for bug reports that support reproducibility. For this RQ, we compare the information provided in the initial version of bug reports with what is provided in their final versions.
- **RQ2: Does information added to bug reports differ based on the type of bug or type of software?** Bug reports capture different types of bugs (e.g., functional errors or performance issues) (Rejaul, 2019; Johnson et al., 2022; Bhattacharya et al., 2013). Similarly, there are different types of software (e.g., e-commerce software or communication software, see Section 3.1). This RQ explores if the information added to bug reports depends on the type of bug we want to reproduce or the type of software the bug occurred in. By studying the relationship between the type of bug or type of software and the type of information, it may be possible to identify trends or patterns that can inform best practices for creating effective bug reports and improve the overall efficiency of bug fixing.
- **RQ3: What data formats are generally used for bug reports?** Bug reports can contain different types of data formats, such as free text, screenshots, code snippets, or a mix of formats. Understanding data formats will help us identify best practices for providing information about bugs in a clear and concise manner to improve the efficiency of bug fixing.
- **RQ4: Do data formats used in bug reports differ based on the type of bug or type of software?** This research question is similar to RQ2, but specifically focuses on the format of the data (e.g., text, images, videos, code) rather than the content of the information in bug reports (i.e., the *what*, see RQ1). We explore if the data formats used in a bug report differ based on the type of bug being reported or the type of software the bug is found in.
- **RQ5: Is the data format used in bug reports impacted by the type of information?** This research question aims to explore in more detail the relationship between the type of information provided (RQ1) and the data format used to describe it (RQ3). The outcome of this research question can help improve bug report quality by recommending data formats for the information type provided in the bug report.

To answer the above RQs, we study bug reports in popular open-source projects on GitHub.

1.3. Paper contributions

By answering the above research questions, we can improve the quality and effectiveness of bug reports, leading to faster and more efficient bug reproduction and resolution. Compared to previous studies discussed later in Section 2, due to the wide usage of the web applications in different domains,[1] our study focuses on web applications. Also, unlike other studies that focus on bug reports for system crashes, our study differentiates types of bugs, such as functional errors. Finally, we examine bug reports manually for an in-depth analysis (rather than automatically analyzing bug reports and mining a large number of bug reports). In detail, in this paper, we make the following contributions:

- We study 1000 bug reports from 10 popular open-source projects on GitHub to identify what information is often missing from initial reports. We performed a manual analysis that provides a granular view of common deficiencies in bug reports, allowing for more nuanced insights compared to automated analyses.
- We identify which types of information are missing for which types of bugs and software. For example, we found that, regardless of the type of software, *diagnostic suggestions* from developers and end-user usage information are often missing in early bug reports but are added throughout the lifetime of a bug report. Also, textual descriptions and screenshots are used most, regardless of the type of bug (e.g., a *functional error*).
- We discuss implications for research and practice. For example, we argue for more systematic techniques for bug data collection and tailored data collection techniques depending on the types of bugs. Furthermore, the skill set of developers is key when managing bug reports.

1.4. Paper outline

In Section 2, we discuss related work. In Section 3, we provide an overview of the research method. We then present the results of our study in Section 4 and discuss our findings in Section 5. In Section 6, we acknowledge threats to the validity of our study before we conclude in Section 7.

2. Related work

We first discuss work related to the quality of bug reports. Then we explore previous research on improving bug reports. We identified previous works from the main digital libraries for computer science and software engineering, such as IEEE,[2] ACM,[3] and Science Direct.[4]

2.1. Quality of bug reports

Several studies have investigated the problem of missing information in bug reports and the impact on bug reproduction. For example, Rejaul recommends including detailed descriptions of the execution environment of software and concrete steps to reproduce the bug, as well as any relevant logs or error messages (Rejaul, 2019).

Similarly, Johnson et al. manually analyzed a dataset of 645 bug reports for 164 Android apps and found that a significant number of reports were incomplete (Johnson et al., 2022). For example, many reports were missing the version of the app, the device model, and steps to reproduce the bug. The authors also found that more complete bug reports were more likely to be successfully reproduced.

Bhattacharya et al. also focused on bug reports for Android apps and investigated the factors that influence the completeness of the reports and the success rate of bug reproduction (Bhattacharya et al., 2013). The study evaluated 24 Android apps and developed a quality matrix system to evaluate bug report quality automatically. The authors found that the completeness of the bug reports differed based on the type of the bug (e.g., security bugs generally had more complete bug reports but took longer to fix). They also found that the success of bug reproduction was affected by the quality of the bug reports and the skill and experience of the involved developers.

[1] https://amplitude.com/guides/2022-app-vs-website-report

[2] https://ieeexplore.ieee.org/Xplore/home.jsp

[3] https://dl.acm.org/

[4] https://www.sciencedirect.com/

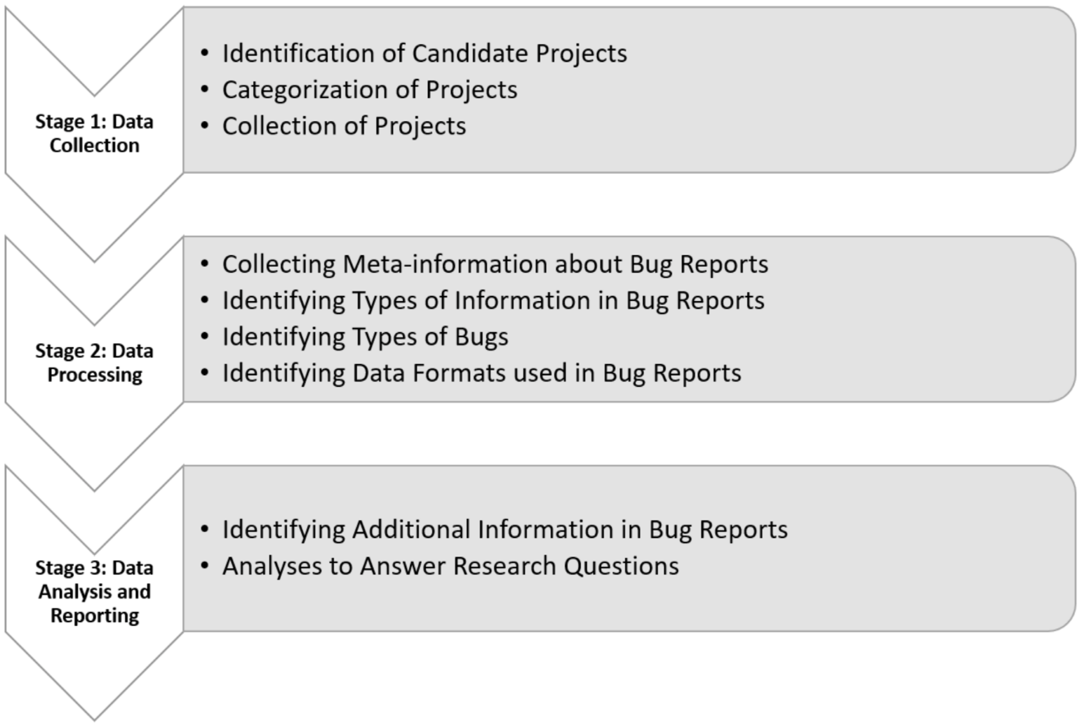

Stage 1: Data Collection
• Identification of Candidate Projects
• Categorization of Projects
• Collection of Projects
Stage 2: Data Processing
• Collecting Meta-information about Bug Reports
• Identifying Types of Information in Bug Reports
• Identifying Types of Bugs
• Identifying Data Formats used in Bug Reports
Stage 3: Data Analysis and Reporting
• Identifying Additional Information in Bug Reports
• Analyses to Answer Research Questions

Fig. 1. Research process.

In summary, recent studies on bug report quality mostly focus on mobile apps. As highlighted by Bhattacharya, earlier works also studied desktop and server-side bugs (Bhattacharya et al., 2013). With the importance of web applications (see Section 1) and the availability of bug reports for those applications, we need to provide a quantitative basis for improving the quality of web applications.

2.2. *Improvement of bug reports*

Various works explored how to improve bug reports. For example, Karim et al. proposed a method for recommending key features to include in bug reports (Karim, 2019). The authors used machine learning to analyze a dataset of bug reports and identified features for improving the completeness and quality of the reports (e.g., including usage flows). They applied their method to bug reports from the Mozilla Firefox project.

Fazzini et al. proposed a method for improving the completeness and accuracy of bug reports for mobile apps by automatically capturing and recording the steps to reproduce a bug (Fazzini et al., 2022). The method uses machine learning techniques to analyze the source code and user interaction data to build a model of system behavior. Then, when a bug is reported, usage data based on the pre-built model will be included in the bug report. The authors demonstrated the effectiveness of their method by applying it to bug reports from the Google Play Store.

In summary, previous works on improving bug reports focus on providing specific kinds of data (e.g., usage data). However, they do not explore what information is missing from bug reports and how this missing information impacts bug reproduction.

3. Research method

To answer the research questions outlined in Section 1, we analyzed information in the initial version of a bug report and compared it with its final version. We studied bug reports in popular open-source projects on GitHub (see below for details on the project selection). We conducted our study following the stages outlined in Fig. 1. The details of each stage are discussed in the following sections.

3.1. *Stage 1: Data collection*

3.1.1. *Identification of candidate projects*

To identify open-source projects for our study, we started with a list of web applications from the `awesome list`.[5] This list includes 136 open-source web applications (as of November 12, 2023) that are (a) used in production, (b) ready to use (no additional development required), (c) can be deployed locally, or (preferably) offered as SaaS, and (d) have issue repositories (GitHub Issues). We needed issue repositories to access bug reports (we treated issues with certain labels as bug reports, see below).

3.1.2. *Categorization of projects*

We categorized all projects into five types of software based on their main usage scenarios and types of users. This was necessary to analyze information in bug reports based on the type of software, see RQ2 and RQ4:

- **Communication software:** Facilitate communication and collaboration between people, either in real-time or asynchronously. Examples include email clients, chat and messaging apps, video conferencing tools, and social media platforms.
- **E-commerce software:** Facilitate online buying and selling of goods and services. Examples include shopping cart software, payment software, and inventory management systems.
- **Personal tools:** Help individuals manage personal lives and tasks. Examples include personal finance software, task management and productivity apps, and note-taking software.
- **Work tools:** Support tasks in the workplace, such as project management software and customer relationship management systems.
- **Development tools:** Support tasks that involve programming and software development. Examples include application monitoring systems and debugging tools.

[5] https://github.com/sdil/open-production-web-projects

Table 1
Metadata of project repositories.

Project	Forks	Stargazers	Open issues	Created	Subscribers	Template
matomo	2384	16 944	2143	30/03/2011	424	Yes
magento2	9042	10 305	2119	30/11/2011	1315	Yes
zammad	619	3172	451	2/04/2012	132	Yes
PrestaShop	4491	6791	2725	19/11/2012	468	Yes
freeCodeCamp	30 235	356 132	209	24/12/2014	8466	Yes
sourcegraph	850	7045	4680	24/08/2015	156	No
zulip	5615	16 614	2692	25/09/2015	375	No
parabol	269	1537	378	21/12/2015	45	No
mastodon	4694	30 124	2391	22/02/2016	631	Yes
cal.com	1683	13 934	520	22/03/2021	114	No

3.1.3. Collection of projects

We filtered the projects based on the number of issues labeled with the keyword "bug" (in the following, we refer to the issues labeled as "bug" as bug reports). We took the top two project repositories from each type of software with the most "bug" labels to ensure a diverse and representative sample of projects. The metadata of the selected projects are presented in Table 1. As can be seen in Table 1, projects vary in terms of activity (e.g., the number of forks and open issues), maturity in terms of age (based on the dates created), and community size (e.g., subscribers and stargazers). Regarding the use of templates to document bug reports in the selected projects, the projects included in this study with a "Yes" in the last column of Table 1 explicitly defined the minimum requirements for the information expected (e.g., each bug report for the matamo project requires a description, expected outcome of the activity reported in the bug report, steps to reproduce the bug and setup information to describe the system setting when a bug occurred). However, not all bug reports in those projects follow the template since (a) a project may have stopped using templates in the period we analyzed, or (b) a project may have started using templates in the period we analyzed it (there is no indication about the start or end of using templates in the project documentation). We then used the GitHub API[6] to fetch the top 100 most recently closed issues and their comments from each of the selected project repositories and saved them in a local database. We only retrieved closed issues to analyze the final version of bug reports rather than bug reports in progress.

The outcome of Stage 1 was a list of ten projects (two for each type of software), including 100 issues (and the comments for these issues) labeled with the keyword "bug" for each of the 10 projects (i.e., 1000 issues in total). Compared to previous studies discussed in Section 2, most included fewer than 300 bug reports.

3.2. Stage 2: Data processing

The aim of Stage 2 was to understand the information provided in the bug reports (i.e., issues and comments). For each bug report, we recorded the following:

1. Meta-information about bug reports to characterize bug reports;
2. Types of information in bug reports (used for RQ1, RQ2, and RQ5);
3. Types of bugs (used for RQ2 and RQ4);
4. Data formats used in bug reports (used for RQ3, RQ4, and RQ5).

Some bug reports may be difficult to judge (e.g., for certain types of information in bug reports, see below). In these cases, we discussed amongst researchers to ensure the reliability of the analysis.

[6] https://docs.github.com/en/rest?apiVersion=2022-11-28

3.2.1. Collecting meta-information about bug reports

To understand the characteristics of bugs, we collected some meta-information about bug reports:

- **Lifetime:** The time it took to close (fix) a bug report. This is defined as the time between opening a bug (and submitting the first initial version of a bug report) and the time it was marked as closed.
- **Comments count:** This captures how many comments were made for a bug report. We counted all comments made for (and added to) the original bug report.
- **People involved:** This captures the number of people involved in a bug report. We included those who originally reported the bug and those who commented on the bug report. Each person only counts once if they made multiple comments. This number gives an idea about the human resources involved in fixing a bug.

3.2.2. Identifying types of information in bug reports

We tagged all issues (including their comments) for the type of information they capture. One issue can be tagged with multiple tags as it may include multiple types of information. We started with six information types mentioned in previous studies (Zimmermann et al., 2012; Davies and Roper, 2014; Zhang et al., 2017):

- **Description:** general information to describe the bug. For example, the following is a *description* of a functional error:

  > *We are using Cal.com on a self-hosted instance. The email invites do not have any MS Teams invite or URL. The location shown is "Office365_Video".*
  >
  > *[Source: https://github.com/calcom/cal.com/issues/3854]*

- **Steps to reproduce:** specific actions or sequence of actions a developer must follow to reproduce the bug. This information allows developers to verify that the bug exists and understand the context in which the bug occurs. Following is an example of *steps to reproduce*:

  > *1. Cal.com self-hosted*
  > *2. Add Office365 Teams app*
  > *3. Create an event with location as MS Teams*
  > *4. Book a meeting and receive an invite with no MS Teams link*
  >
  > *[Source: https://github.com/calcom/cal.com/issues/3854]*

- **Outcome Info:** expected behavior of software if the bug did not occur. This information helps developers understand the expected behavior of the software and compare it with what actually happened. Following is an example:

  > *Outcome Info: Show only customer attributes with existing content*
  >
  > *[Source: https://github.com/zammad/zammad/issues/4172]*

- **Setup data:** information about the environment in which the bug occurred, such as the version of the software, the operating system, and the hardware configuration. This information is important because it allows the developer to recreate the conditions under which the bug occurred. Following is an example:

  > *Preconditions*
  > *1. Magentoto 2.1.7*
  > *2. PHP Version 7.0.7*
  > *3. Apache/2.4.7 (Ubuntu)*
  >
  > *[Source: https://github.com/magento/magento2/issues/10685]*

- **Error log:** record of error messages or other diagnostic information generated when the bug occurs. This information can help identify the cause of the bug. Following is an example:

  > *runtime error: index out of range [0] with length 0*
  >
  > *[Source: https://github.com/sourcegraph/sourcegraph/issues/41016]*

- **Running log:** record of the actions that were taken and the results that were produced during the execution of the software until the bug occurred. This information can be useful in understanding the sequence of events leading up to the bug. We provide an example below:

  ```
  Response code: 200 (OK); Time: 149ms
  event: progress data: {
  "done":true,
  "matchCount":0,
  "durationMs":15
  }
  event: done
  data: {}
  ```
  > *[Source: https://github.com/sourcegraph/sourcegraph/issues/39835]*

We then identified three more types of information as existing types were not comprehensive enough to fully describe what we found in the bug reports:

- **End user usage data:** information about how the software was used by end users, such as the frequency and duration of usage, as well as any feedback or observations provided by the users. This information can be helpful in understanding the impact of the bug on end users and prioritizing it. We provide an example below:

  > *Yeah we should move the picture at more narrower windows. And having it at the top and sufficiently narrow would be better, as I have seen profile pictures are usually at the top at other places.*
  >
  > *[Source: https://github.com/matomo-org/matomo/issues/18612]*

  Unlike the *steps to reproduce* discussed above, end-user usage information is unstructured and not directly related to the bug. End-user usage information is about user interaction with the system but not a step-by-step instruction for how the error can be triggered. User interaction data alone cannot make a bug reproducible.
- **Diagnostic suggestions:** any ideas or hypotheses the reporter has about the cause of the bug. This information can be useful in guiding the developer's investigation. An example is as follows:

  > *In case it helps, it seems to mostly happen with high traffic instances. It could be though that it's unrelated to high traffic and maybe they use certain features or so.*
  >
  > *[Source: https://github.com/matomo-org/matomo/issues/18937]*

- **Solution suggestions:** any ideas for fixing the bug that are provided by the reporter. While these suggestions may not always be feasible or practical, they can still provide valuable insights and help the developer come up with a solution. Below we provide an example:

  > *Ok, great. So our plan for this issue is:*
  > *- [ ] Make those hotkeys a noop if the compose box is already open.*
  > *- [x] Call the draft save function whenever we're about to clear the compose box to start a new message. This is primarily a defense in depth thing, but seems well worth doing. Either one will close this bug, but we want to do both.*
  >
  > *[Source: https://github.com/zulip/zulip/issues/21128]*

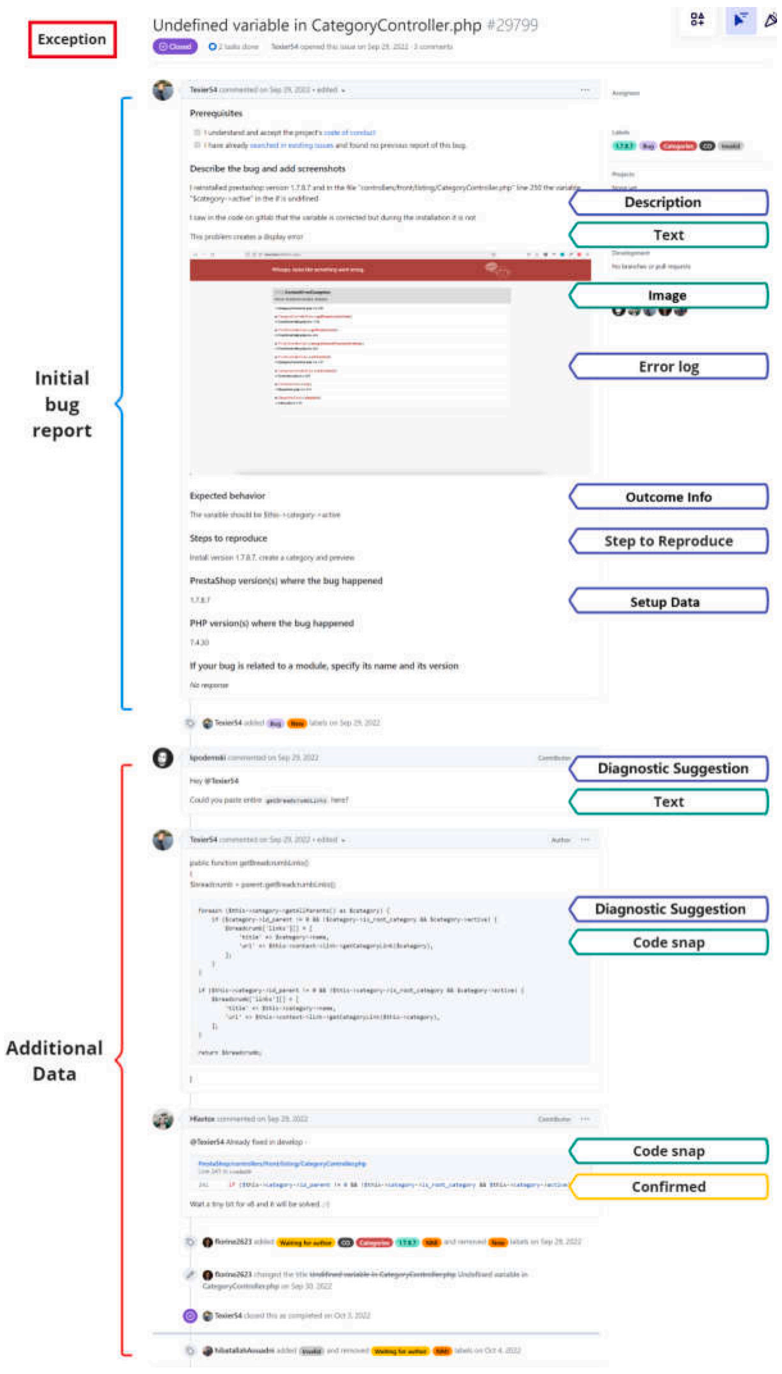
3.2.3. Identifying types of bugs

We identified five types of bugs based on the symptoms of the bug. The symptoms of a bug are identified based on the title and description fields of the bug report (see Fig. 2). For example, the tag at the top left of Fig. 2 ("Exception") is indicating the type of bug (in this case the bug report is for an exception). Classifying the types of bugs is based on the title and description of the bug in the report. These types are based on previous studies (Rejaul, 2019; Johnson et al., 2022; Bhattacharya et al., 2013). Each bug report was tagged with one type of bug:

- **Functional error:** A *functional error* affects the functionality of a program. This type of bug may cause the program to behave incorrectly or fail to perform the tasks it was designed to do. *Functional error*s can be caused by problems in the code, incorrect user input, etc.
- **System error:** A *system error* prevents the whole application or parts of the application from starting up appropriately. It can arise due to problems with the environment in which the program is running or how it is configured. This could be related to the operating system, hardware, etc.
- **Performance error:** A *performance error* is an issue that affects the program's performance, such as slowing down or crashing. This type of bug may be caused by problems with the code, system resources, etc.
- **Exception:** An *exception* is an unexpected and not handled error that occurs during the execution of a program. *Exceptions* are usually caused by problems in the code, such as a null reference or an index out-of-bounds error. When an *exception* occurs, it typically causes the program to stop and display an error message.
- **Miss-labeled report:** Keywords in GitHub are assigned by human users. Some of the keywords could be incorrectly used with issues that do not report an actual bug. For example, within the 1000 issues we analyzed, we found 55 which were not bugs but included other content, such as instructions for writing bug reports.

3.2.4. Identifying data formats used in bug reports

We initially identified two main categories of data formats used to represent information in bug reports: (1) text-based data and (2) visual data. After further analysis we noticed that there are different types of textual data and visual data. We, therefore, refined the initial two categories into four categories based on recurring data formats (one bug report can include multiple data formats).



Undefined variable in CategoryController.php #29799

Closed 2 tasks done Texier54 opened this issue on Sep 29, 2022 · 3 comments

Texier54 commented on Sep 29, 2022 • edited

Prerequisites

- I understand and accept the project's code of conduct
- I have already searched in existing issues and found no previous report of this bug.

Describe the bug and add screenshots

I reinstalled prestashop version 1.7.8.7 and in the file "controllers/front/listing/CategoryController.php" line 250 the variable "$category->active" in the if is undefined

I saw in the code on gitlab that the variable is corrected but during the installation it is not

This problem creates a display error

Whoops, looks like something went wrong.

Expected behavior

The variable should be $this->category->active

Steps to reproduce

Install version 1.7.8.7, create a category and preview

PrestaShop version(s) where the bug happened

1.7.8.7

PHP version(s) where the bug happened

7.4.30

If your bug is related to a module, specify its name and its version

No response

Texier54 added Bug New labels on Sep 29, 2022

lpodemski commented on Sep 29, 2022 Contributor

Hey @Texier54

Could you paste entire `getBreadcrumbLinks` here?

Texier54 commented on Sep 29, 2022 • edited Author

public function getBreadcrumbLinks()
{
$breadcrumb = parent::getBreadcrumbLinks();

```
foreach ($this->category->getAllParents() as $category) {
    if ($category->id_parent != 0 && !$category->is_root_category && $category->active) {
        $breadcrumb['links'][] = [
            'title' => $category->name,
            'url' => $this->context->link->getCategoryLink($category),
        ];
    }
}

if ($this->category->id_parent != 0 && !$this->category->is_root_category && $category->active) {
    $breadcrumb['links'][] = [
        'title' => $this->category->name,
        'url' => $this->context->link->getCategoryLink($this->category),
    ];
}

return $breadcrumb;
```

}

Hlavtox commented on Sep 29, 2022 Contributor

@Texier54 Already fixed in develop -

PrestaShop/controllers/front/listing/CategoryController.php
Line 247 in 1ceda6b

```
241        if ($this->category->id_parent != 0 && !$this->category->is_root_category && $this->category->active
```

Wait a tiny bit for v8 and it will be solved :-)

florine2623 added Waiting for author CO Categories 1.7.8.7 NMI and removed New labels on Sep 29, 2022

florine2623 changed the title ~~Undefined variable in CategoryController.php~~ Undefined variable in CategoryController.php on Sep 30, 2022

Texier54 closed this as completed on Oct 3, 2022

hibatallahAouadni added Invalid and removed Waiting for author NMI labels on Oct 4, 2022

Fig. 2. Example bug report (final version).

1. **Text-based data:**
    - **Free text:** Content written in natural language (e.g., English) regardless of the type of information, type of bug, etc. (all examples in the previous Section 3.2.2).
    - **Code snap:** Snapshot of code to provide context or illustrate the reported problem at the implementation level. The snapshot may include a small section of code that is relevant to the bug, or it may include the entire codebase if the bug is widespread or difficult to isolate. Code can be especially useful if the bug involves a specific piece of code or if the issue is related to the implementation of a feature.
2. **Visual data:**
    - **Image:** Images provide visual context or illustrate the reported issue. For example, if the bug involves a glitch in the graphical user interface, an *image* can help the developers understand the problem.
    - **Video:** Videos provide more extensive visual context of a bug (Feng and Chen, 2022). A *video* can be particularly useful if the bug involves a series of events or if it is difficult to illustrate with a static *image*.

3.3. Stage 3: Data analysis and reporting

3.3.1. Identifying additional information in bug reports

We used the *initial* version of bug reports and the *final* (or last) version of a bug report to identify *additional data/information* added to bug reports. An initial version of a bug report captures the information before any modifications to it. For example, when we analyze types of information, we first look at the initial bug report and the types of information included. The final version of the bug report is the last version before a bug is confirmed (or closed). It includes additional information in comments added to a bug report during its lifetime.

We show an example bug report and the *additional data/information* (i.e., the final version of the bug report) as well as the tags to record types of information and types of data formats in Fig. 2.

3.3.2. Analyses to answer research questions

To answer RQ1 (types of information frequently missing in bug reports), we analyzed the "tags" assigned to the original version of a bug report and the last (final) version to see what information types were added. We assigned a tag to the initial bug report and any additional information (i.e., comments added to the initial bug report) only once (e.g., if *diagnostic suggestions* appeared in two additional comments, that type of information was assigned twice to the bug report, regardless of whether one comment included more than one diagnostic suggestion). We then analyzed the "delta" between the initial bug report and additional information in comments added to the initial bug report to find out the most commonly missed types of information in the initial bug report (i.e., the additional types of information). We also analyzed the frequency of instances of additional types of information. When counting instances, we did not include the instances in the initial version of the bug report but only in the additional information.

Similarly, to answer RQ3 (data formats used in bug reports), we looked at the data format used in the initial and final versions of the bug report to determine whether a specific data format can better provide some information.

To answer RQ2, RQ4, and RQ5, we cross-referenced the data we collected for the types of information and data format with software types, bug types, etc. We then analyzed the data using a bubble chart to determine any significant relations between different data types.

When presenting the differences between bug reports, we indicate the *spread* of added information (i.e., how many bug reports added information, regardless of how many times a tag related to data items was assigned) and also *intensity* (i.e., considering the number of times a tag was added to a bug report). For example, one tag may only appear in two bug reports of the 1000 bug reports (low spread), but the tag may appear many times in each of these two bug reports (high intensity). Since each information type tag can only be applied once to an initial version of a report (since there are no comments), the data intensity and the data spread for initial bug reports are the same.

Table 2
Characteristics of the 1000 bug reports in sample.

	Min	Max	Median	Mean	Standard deviation
Lifetime (days)	<1	1,191	5	28.90	99.30
Comments count	0	30	2	3.37	4.42
People involved	1	14	2	2.63	1.76

4. Results

In this section, we present the primary findings derived from the collected data. The raw data can be accessed from a public GitHub repository.[7]

4.1. Overview

Based on meta-information about bug reports (see Section 3.2.1) we characterize our sample of bug reports as shown in Table 2.

As shown in Fig. 3, of the 1000 bug reports, 732 (73.2%) are classified as *functional errors*, 118 (11.8%) as *system errors*, 94 (9.4%) as *exceptions*, and one as *performance error*. Fifty-five are *miss-labeled reports*. As the miss-labeled bug reports do not report a bug, and we only have one bug report that reports a performance error in our sample, we do not consider those in the following analysis.

In addition, we cross-referenced the types of applications (based on our sampling, we had 200 bug reports for each type of application, see Section 3) and bug types, see Fig. 4. This figure shows that while our sample was balanced in terms of the types of applications, it was not balanced regarding the types of bugs. *Functional errors* dominate the bug reports in our sample across all types of applications.

4.2. RQ1: Missing information in bug reports

To answer RQ1, we collected the type(s) of information provided in the initial and final versions of the same bug report.

Fig. 5 shows the overall distribution of types of information at the level of bug reports (i.e., regardless of how many times an information type was included in a bug report, but only counting whether or not it was added to a bug report). This shows the *spread* of types of information across bug reports. The blue columns in Fig. 5 show the types of information added to the initial version of bug reports, while the orange bars show the types of information already included in the initial version of the reports.

In contrast, Fig. 6 shows the distribution of types of information in bug reports considering how many times an information type was included in a bug report. This shows the *intensity* of information types across bug reports. This allows us to analyze if some types of information were added more frequently to individual bug reports (e.g., some types of information may have a low spread across bug reports but high density in a few bug reports). For example, the 89 occurrences of *steps to reproduce* appeared in 64 individual bug reports (i.e., some bug reports had multiple instances of *steps to reproduce*).

For the initial versions of bug reports (orange bars in Figs. 5 and 6) we observed the following:

[7] https://doi.org/10.5281/zenodo.11878280

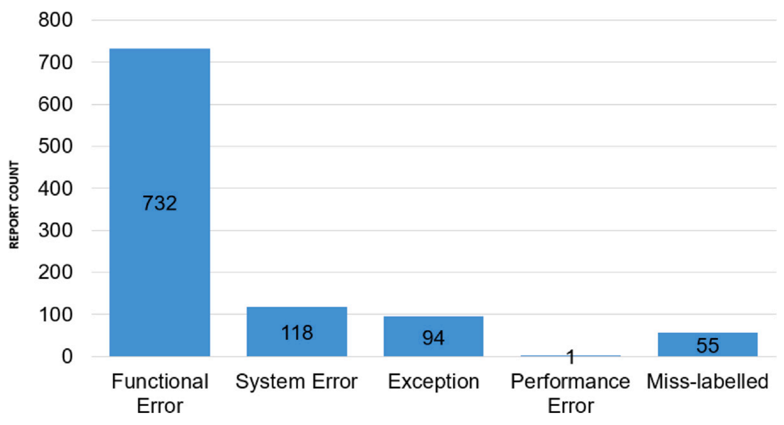


Fig. 3. Types of bugs.

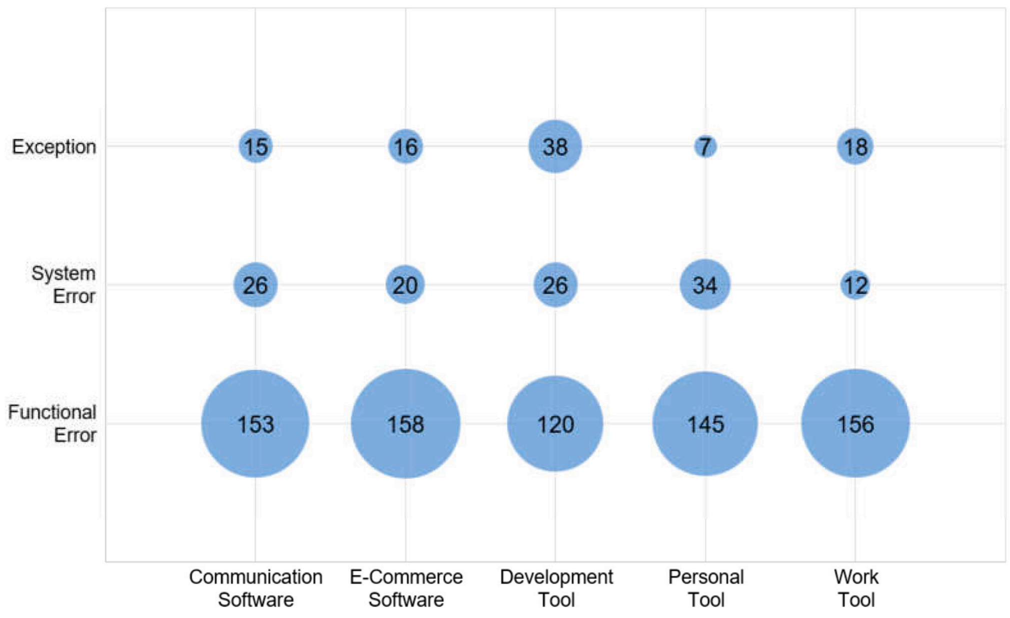


Fig. 4. Types of software vs. types of bugs (excluding the one performance bug).

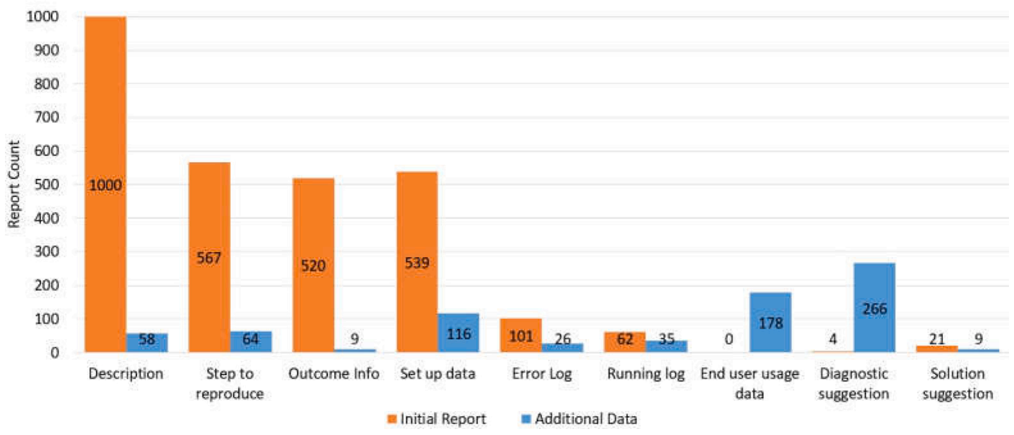


Fig. 5. Types of information in bug reports (spread).

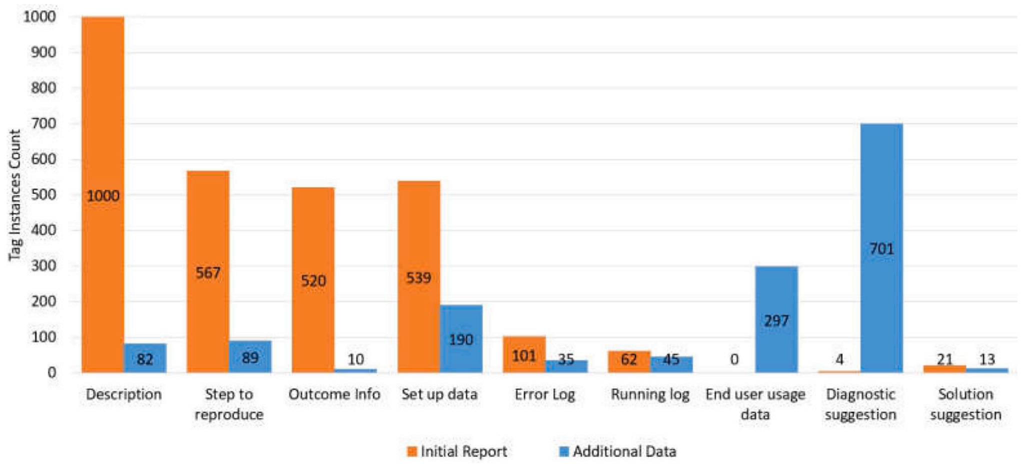


Fig. 6. Types of information in bug reports (intensity).

- All bug reports include a problem description (note that there were no mandatory fields in the form to report bugs in the studied systems). The average length of problem descriptions is 2521 characters (ranging from 87 characters to 97,867 characters).
- More than 50% of bug reports include *steps to reproduce*, *outcome info*, and *setup data*. This could imply that those types of information are considered highly relevant when dealing with a bug.
- *Error logs* and *running logs* are not common (in 5% and 10% of bug reports, respectively). This could be due to the fact that those types of information are difficult to obtain and require more technical knowledge and insights into the software.
- Not many bug reports contain *diagnostic suggestions* or *solution suggestions*. This again could be due to the fact that presenting suggestions would require further analysis and investigations before they could be put into a report.
- No bug report included *end-user usage information*. This information requires insights into how users use the software. When reporting a bug, those who report it may not have detailed insights into the actions and usage scenarios when the bug occurred.

We then looked at the types of information added in the final versions of bug reports (blue bars in Figs. 5 and 6):

- *Diagnostic suggestions* are the most added type of information (provided 701 times spreading across 266 (27%)) of bug reports, followed by *end-user usage information* (provided 297 times spreading across 178 (18%) bug reports) and *set-up data* (provided 190 times spreading across 116 (12%) bug reports). This could indicate that *diagnostic suggestions* and *end-user usage information* were required to deal with a bug and collected while fixing a bug. While *problem descriptions* were most common in original bug reports, see above, additional *problem descriptions* were added 82 times, spreading across 41 (4%) bug reports.
- *Outcome Info* (provided 10 times spreading across 9 (1%) bug reports) and *solution suggestions* (provided 13 times spreading across 9 (1%) bug reports) are the least added types of information. This could be because *solution suggestions* were taken from the initial bug reports or implemented rather than documented in a big report.

Table 3 provides additional information about the types of information added to bug reports to their final version. *Diagnostic suggestions* emerge as the most emphasized category, with the highest intensity (701), spread (266), maximum (26 occurrences in one bug report), and average values (2.64 occurrences of *diagnostic suggestions* on average per bug report), indicating their crucial role in bug resolution. *Setup data* and *End User Usage Data* also stand out due to their high frequency in reports, as shown by a spread of 116 and 178, respectively. This underscores their importance in understanding and addressing bugs.

Key insights for RQ1:

- All bug reports initially include a problem description but often lack technical details like *error logs* and *diagnostic suggestions*.
- Over half of the reports initially contain *steps to reproduce* and *outcome info*. However, as the bug resolution process advances, the final versions significantly incorporate additional details, particularly *diagnostic suggestions* and *end-user usage information*, underscoring their importance in bug resolution.
- The evolution to comprehensive reports with technical and user-centric information highlights the dynamic and progressive nature of bug reporting and resolution.

4.3. RQ2: Missing types of information based on type of bug and type of software

4.3.1. Missing information based on types of bugs

We cross-referenced bug types with the types of information provided in the initial version of bug reports (Fig. 7) and the types of information added (Fig. 8). In Figs. 7 and 8, we present two charts: the upper chart displays the “absolute” count of instances, while the lower chart shows the “normalized data” (based on the number of reports of a bug type category). Below, we summarize the spread of types of information in the initial and final versions of the bug reports:

- **Functional errors:** The most commonly provided type of information in the initial bug reports is *steps to reproduce*, followed by *outcome info* and *setup data*. All other types of information appear in less than 5% of reports.

  Over the lifetime of a *functional error*, the most commonly added type of information is *diagnostic suggestions*, followed by *setup data.*

  Types of information that were not provided in initial versions of bug reports were not always added (e.g., *error logs*, *running logs*, *solution suggestions*, *end user usage data*), while other types of

Table 3
Overview of the types of information added to bug reports.

Type of Information	Spread	Intensity	Average	Min	Max	Median	StdDev
Description	82	96	1.17	0	3	0	0.35
Steps to Reproduce	64	89	1.39	0	3	0	0.36
Outcome Info	9	10	1.11	0	2	0	0.11
Setup Data	116	190	1.64	0	7	0	0.62
Error Logs	26	35	1.35	0	3	0	0.21
Running Logs	35	45	1.29	0	7	0	0.32
End User Usage Data	178	297	1.67	0	9	0	0.76
Diagnostic Suggestion	266	701	2.64	0	26	0	1.70
Solution Suggestion	9	13	1.44	0	3	0	0.11

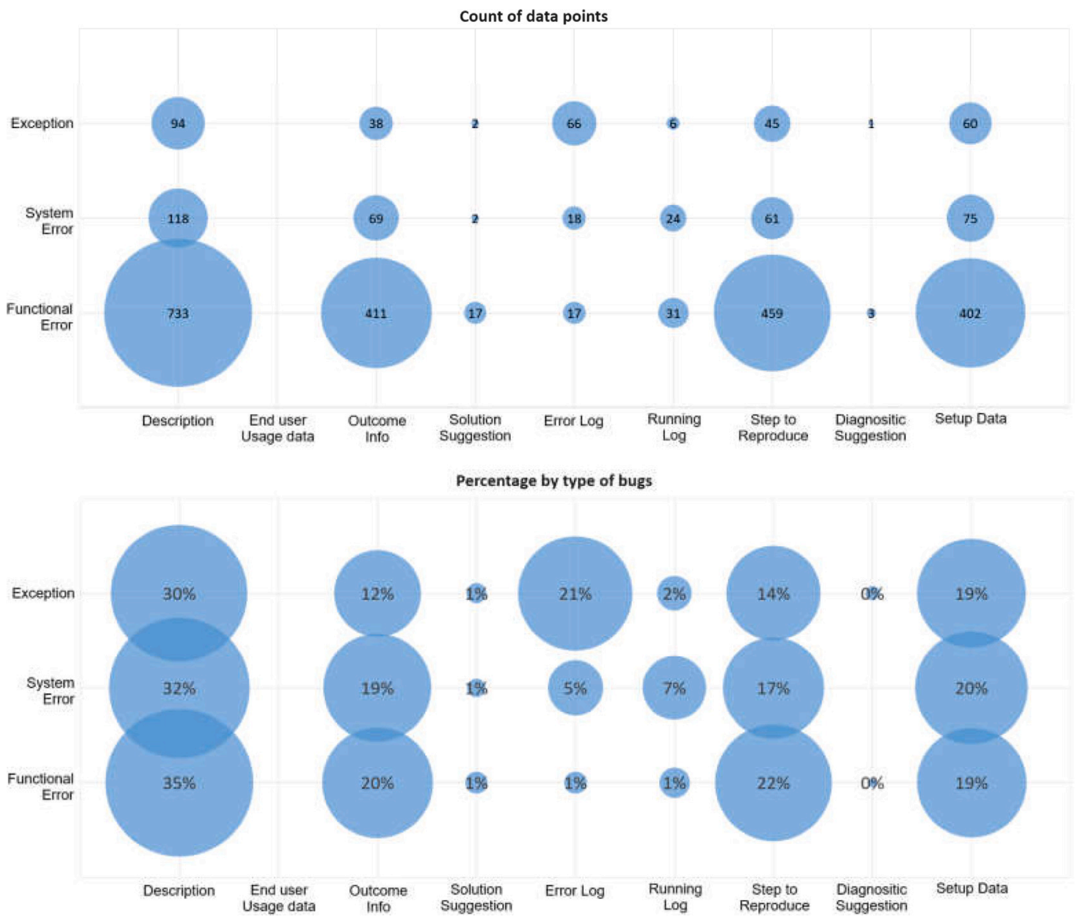


Fig. 7. Types of information vs. types of bugs (initial bug reports).

information that were provided in initial reports were extended (e.g., *outcome info*).

- **System errors:** The most commonly provided type of information in the initial reports is *setup data*, followed by *outcome info* and *steps to reproduce*. Compared to *functional errors*, *running logs*, and *error logs* are provided more often. *Solution suggestions* are provided in 2% of the bug report, while *diagnostic suggestions* is not provided in any of the bug reports.
  The most provided types of information added in the final version of bug reports are *diagnostic suggestions* and *setup data*. *Solution suggestions* and outcome information are not provided in any of the bug reports for *system errors*.
  In contrast to *functional errors*, bug reports for *system errors* more frequently include *running logs* (7%).
- **Exceptions:** The most frequently provided type of information in the initial versions of bug reports is the *error logs*, followed by *setup data*, *steps to reproduce*, and *outcome infos*. All other data types are provided in less than 10% of bug reports.
  Interestingly, the most frequently added type of information is also *error logs*.

Our analysis reveals distinct patterns in the type and intensity of information provided in bug reports, as shown in Figs. 7 and 8, offering insights into the nature of different bug types.

- **Functional error:** The analysis shows a strong tendency toward *diagnostic suggestions*, which comprise 45% of all additional data in *functional error* reports. This high percentage, derived from 494 instances of *diagnostic suggestions* out of a total of 1101 instances

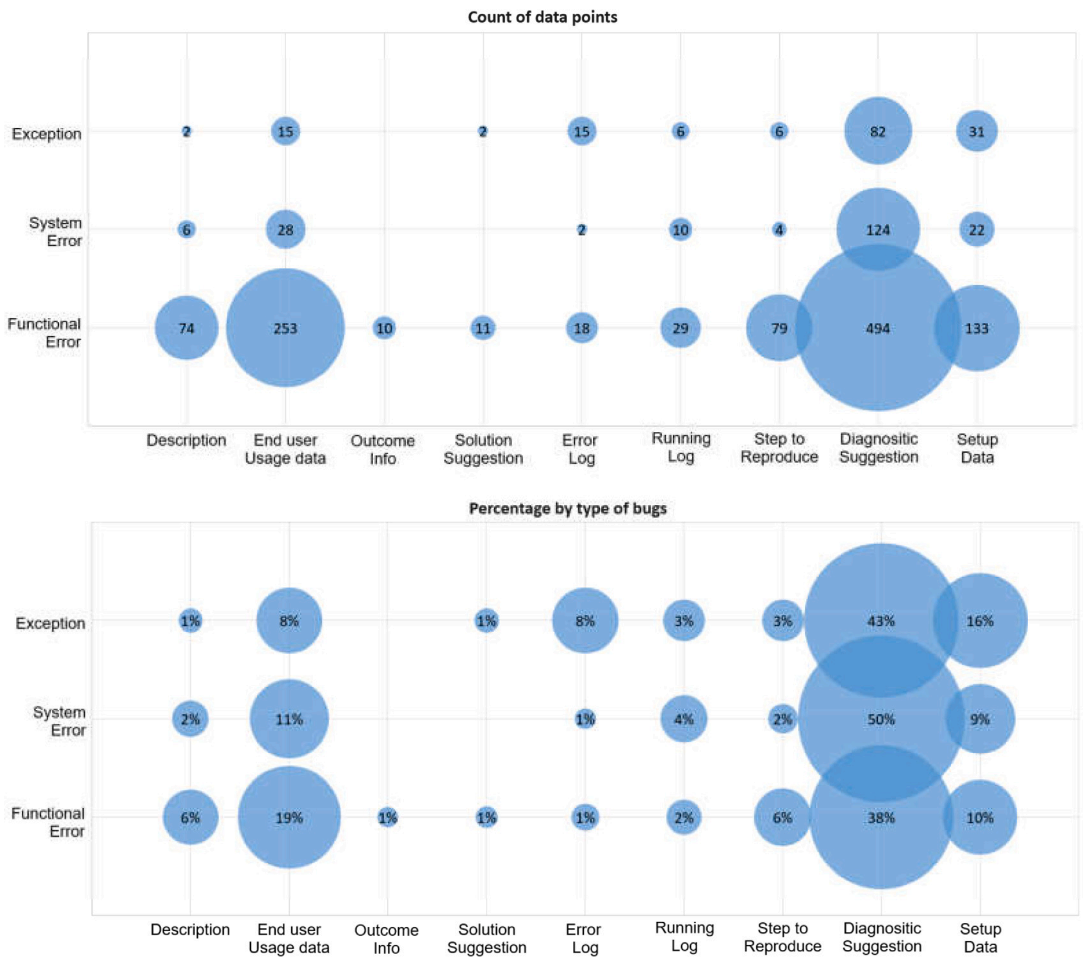


Fig. 8. Types of information vs. types of bugs (additional data).

(i.e., 74+253+10+11+18+29+79+494+133 [see last row of bubbles in Fig. 8]) of additional data, underscores a focus on identifying root causes of errors. Furthermore, *end-user usage information*, accounting for 19% of the additional data, is significant. It highlights the importance of understanding user interactions in the context of *functional error*s. *Steps to reproduce*, though representing a smaller portion (6%), are critical in enabling developers to replicate and resolve the issues effectively.

- **System errors:** These reports predominantly contain *diagnostic suggestions* (50%), underlining their crucial role in troubleshooting system-related issues. The lesser prominence of *end-user usage data* (11%) in these reports suggests a reduced emphasis on user interactions. Notably, the minimal presence of *error log* information (1%) in *system error* reports points to a potential area for improvement in reporting practices for these types of errors.
- **Exceptions:** Reports on *exceptions* show a balanced distribution of data types. *Diagnostic suggestions*, constituting 43%, highlight their importance in addressing *exceptions*. The equal representation of *end-user usage data* and *error logs* (8% each) emphasizes the need to understand both the user context and technical specifics of errors. *Steps to reproduce* are less common (3%) in these reports but still provide valuable insights for troubleshooting.

These findings suggest that the nature of the bug significantly influences the type of information reported. The prevalence of *diagnostic suggestions* across all bug types highlights their overall importance in the bug resolution process. However, the varying degrees of emphasis on user data and *steps to reproduce* across different bug types indicate specific areas where reporting practices could be optimized for more effective troubleshooting.

Key insights for RQ2 (types of bugs):

- All types of information appear in reports for all types of bugs, either already in the initial bug reports or after adding data in the final reports (see combination of bubbles in Figs. 7 and 8), but the distribution of types of information differs for bug types.
- Reports of *functional error*s include end-user usage data more often than *exceptions*.
- Reports of *exceptions* include *error logs* and *setup data* more frequently compared to reports of *functional error*s.

4.3.2. *Missing information based on types of software*

We analyzed the types of software in relation to the types of information present in both initial bug reports and additional information, as illustrated in Figs. 9 and 10. The findings are summarized as follows:

- **Communication software:** Initially, *steps to reproduce*, *outcome infos*, and *setup data* were most common in over half of the reports. Over time, *diagnostic suggestions* became the most added information, with 175 additional occurrences in more than half (51%) of the bug reports for *communication software*. End user usage data add to 17% of bug reports and descriptions in 11% of bug reports.

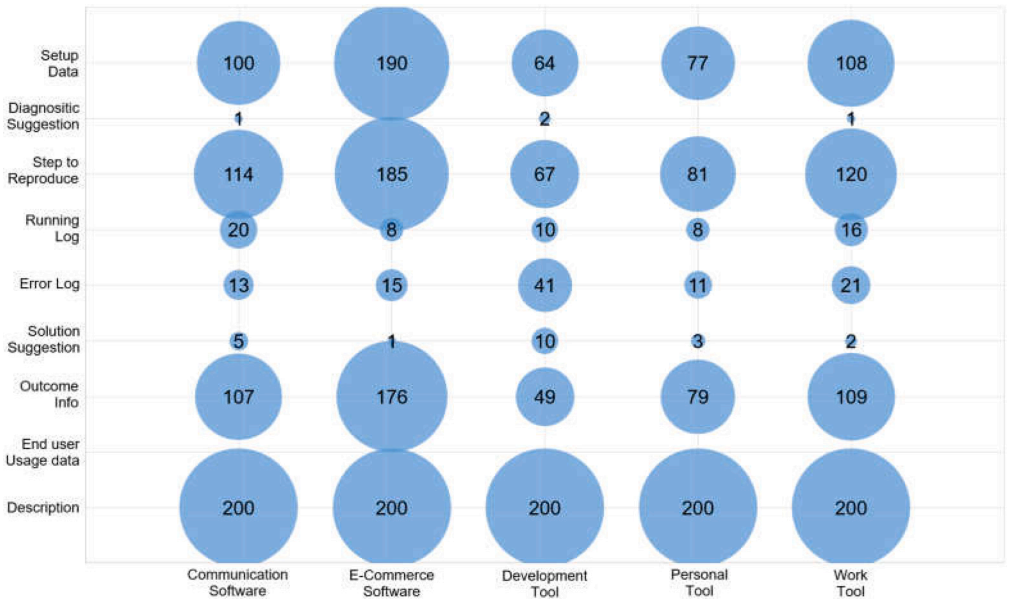


Fig. 9. Types of information vs. types of software (initial bug reports).

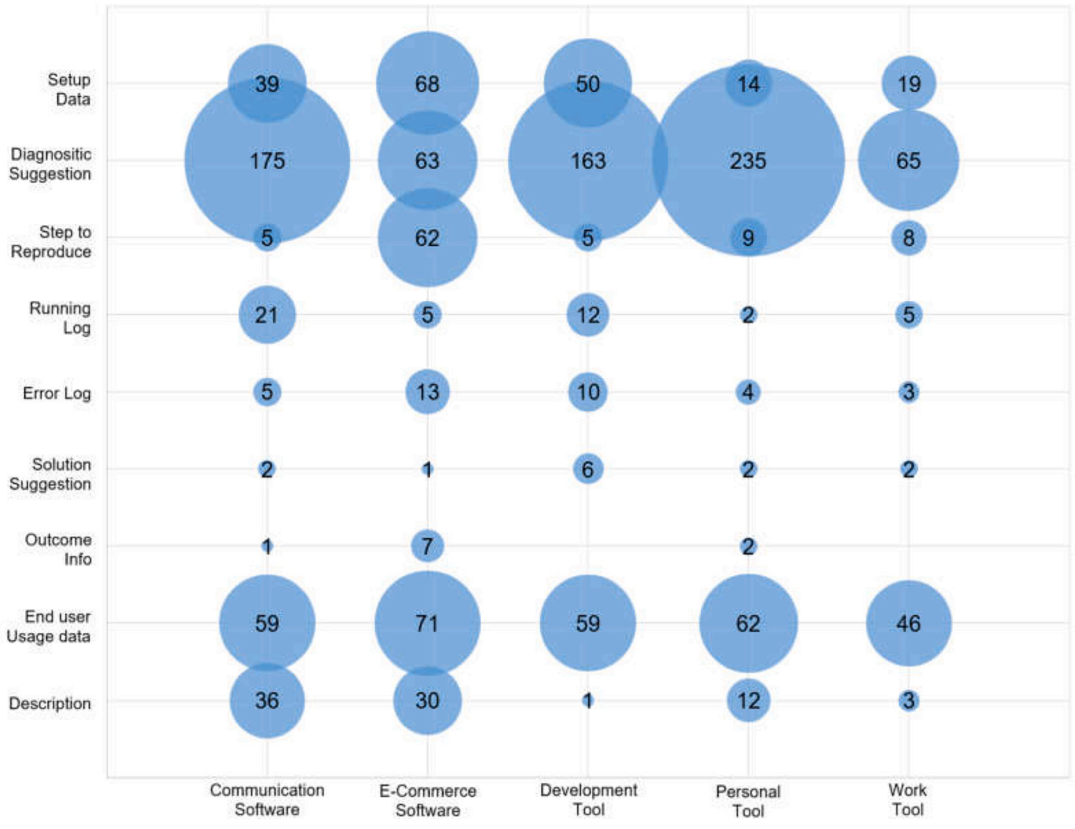


Fig. 10. Type of information vs. type of software (additional data).

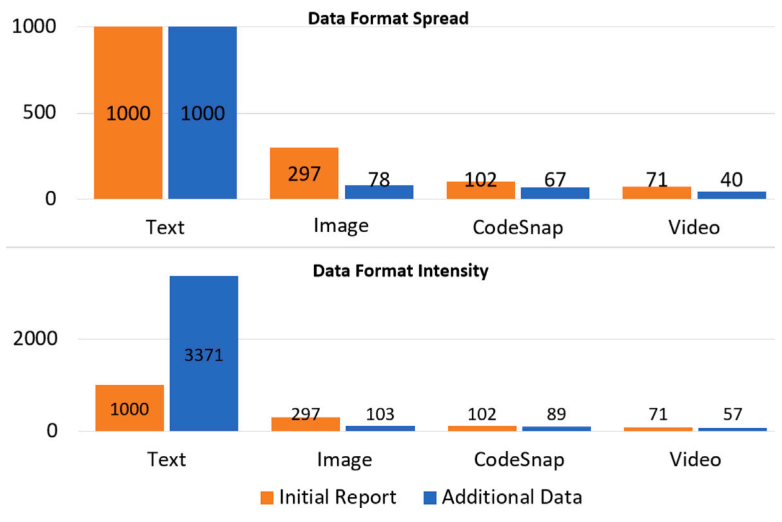


Fig. 11. Data format used in bug reports.

- **E-Commerce software:** Initial reports predominantly included *steps to reproduce* and *setup data*. The final versions saw a rise in *setup data*, *diagnostic suggestions* (17%), and *end-user usage information* (22%).
- **Personal tools:** In the initial reports, *steps to reproduce* and *setup data* appeared in nearly half of the reports. The final versions showed a notable addition of *diagnostic suggestions* (40%), indicating a shift in focus during bug resolution.
- **Work tools:** The initial reports frequently contained *steps to reproduce*, *outcome infos*, and *setup data*. However, in the final versions, *diagnostic suggestions* (17%) and *end-user usage information* (14%) were the most commonly added types of information.
- **Development tools:** *Setup data* and *error logs* were dominant in less than half of the initial reports. In the final versions, *diagnostic suggestions* significantly increased to 41%, with *end-user usage data* (15%) and *setup data* (13%) increasing as well.

Key insights for RQ2 (types of software):

- Across all software types, *diagnostic suggestions*, and *end-user usage information* increased from the initial to the final versions of bug reports.
- Additional data regarding *steps to reproduce* for *e-commerce software* is provided significantly more compared to other software types.
- The increase in *end-user usage information* highlights a trend toward a more diagnostic and user-centric approach in resolving bugs as more information becomes available over time.

4.4. RQ3: Data formats used in bug reports

Fig. 11 provides an overview of the data formats used in the analyzed bug reports.

- For **initial reports**, the most used data format is text (used in all bug reports), followed by *images* (in 297 reports), *code snaps* (used in 102 reports), and *videos* (in 71 bug reports).
- A similar trend for data formats appears in the **additional data** in the final versions of bug reports. Text data was added to all bug reports. 78 bug reports added screenshot *images*, 67 *code snaps*, and 40 bug reports, including *videos*. Furthermore, the intensity of data shows the same trend as the spreading.

Key insights for RQ3:

- *Text* is the most frequently used format to document bugs.
- *Images* and *code snaps* are added during the lifetime of a bug report.

4.5. RQ4: Data formats based on type of bug and type of software

To answer RQ4, we cross-referenced the data formats identified for RQ3 with types of bugs and types of software. Since text is used in all bug reports, we will not include it in this analysis.

4.5.1. Data formats based on types of bugs

Figs. 12 and 13 show how data formats are used in bug reports depending on the type of bug. In each of these figures, we present two charts: the upper chart displays the “absolute” count of instances, while the lower chart shows the normalized percentage based on the number of bug reports for a type of bug. Below, we summarize our observations:

- **Functional errors:**
  - *Image* data is predominant in initial bug reports, featuring in 264 (69%) reports, highlighting its utility in capturing bugs that are observable on the interface. Other visual data (*video*) are used less.
  - For additional information, *code snap* usage increases to 57 occurrences across 43 reports, *video* appears 50 times in 35 reports, and *Image* data is used 87 times in 67 reports.
- **System errors:**
  - *Code snap* is the dominating data format, with 80% in relevant bug reports.
  - *Video* is minimally used, appearing in only one report.
  - In the additional data, the distribution of data formats is similar to the original bug reports. *Code snap* is used 21 times in 15 reports, *video* 3 times in 3 reports, and *images* in 7 reports.
- **Exceptions:**
  - *Exception* reports show similar usage of *code snap* (in 10 cases) and *image* in 13. However, compared to the *code snap*

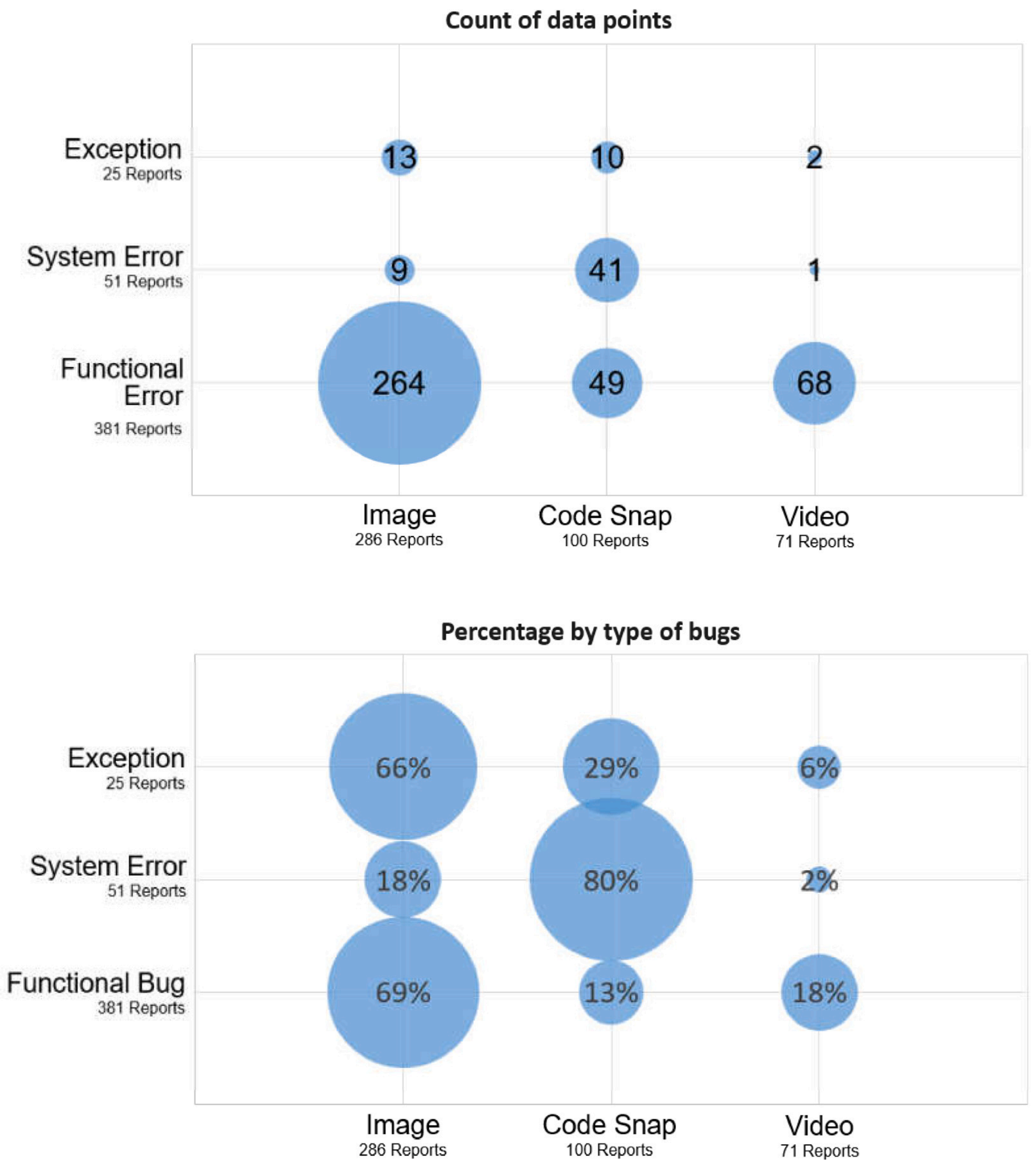


Fig. 12. Types of bugs vs. data formats (initial bug reports).

and *image*, *video* format shows a very low usage in only two bug reports.
– Additional data shows *code snap* used 9 times in 9 reports, *video* 4 times in 3 reports, and *images* 9 times in 6 reports.

Key insights for RQ4 (types of bugs):

- *Screenshot* data is most widely used across bug types.
- The *Video* format, while less common, plays a unique role in reports of *functional error*s.
- The variance in data formats across bug types underscores the need for tailored documentation approaches for different types of bug reports.

4.5.2. Data formats based on types of software

Figs. 14 and 15 show the data formats used in bug reports for different types of software:

- **Communication software:** In initial bug reports, *screenshots* dominated with 49 reports, significantly overshadowing *code snap* (17 reports) and *video* (5 reports). However, the additional data reveals a marked shift, with *screenshots* declining to 13 reports. This highlights a trend from initial visual documentation to a more nuanced approach involving detailed report formats.
- **E-commerce software:** *Screenshots* initially dominated with 76 reports, equally followed by *code snap* and *video* with 12 reports each. Intriguingly, in the additional data, *video* reports increased to 25, indicating their growing importance in dynamic bug reporting, while *screenshots* and *code snap*, though still relevant, became less dominant. This reflects the adaptability of bug reporting formats in response to the dynamic nature of e-commerce apps.
- **Personal tools:** There was a high initial occurrence across all formats, notably *screenshots* (53 reports) and *code snap* (47 reports). The additional data saw *code snap* maintaining a significant presence while screenshots and *video* scaled back. This underscores

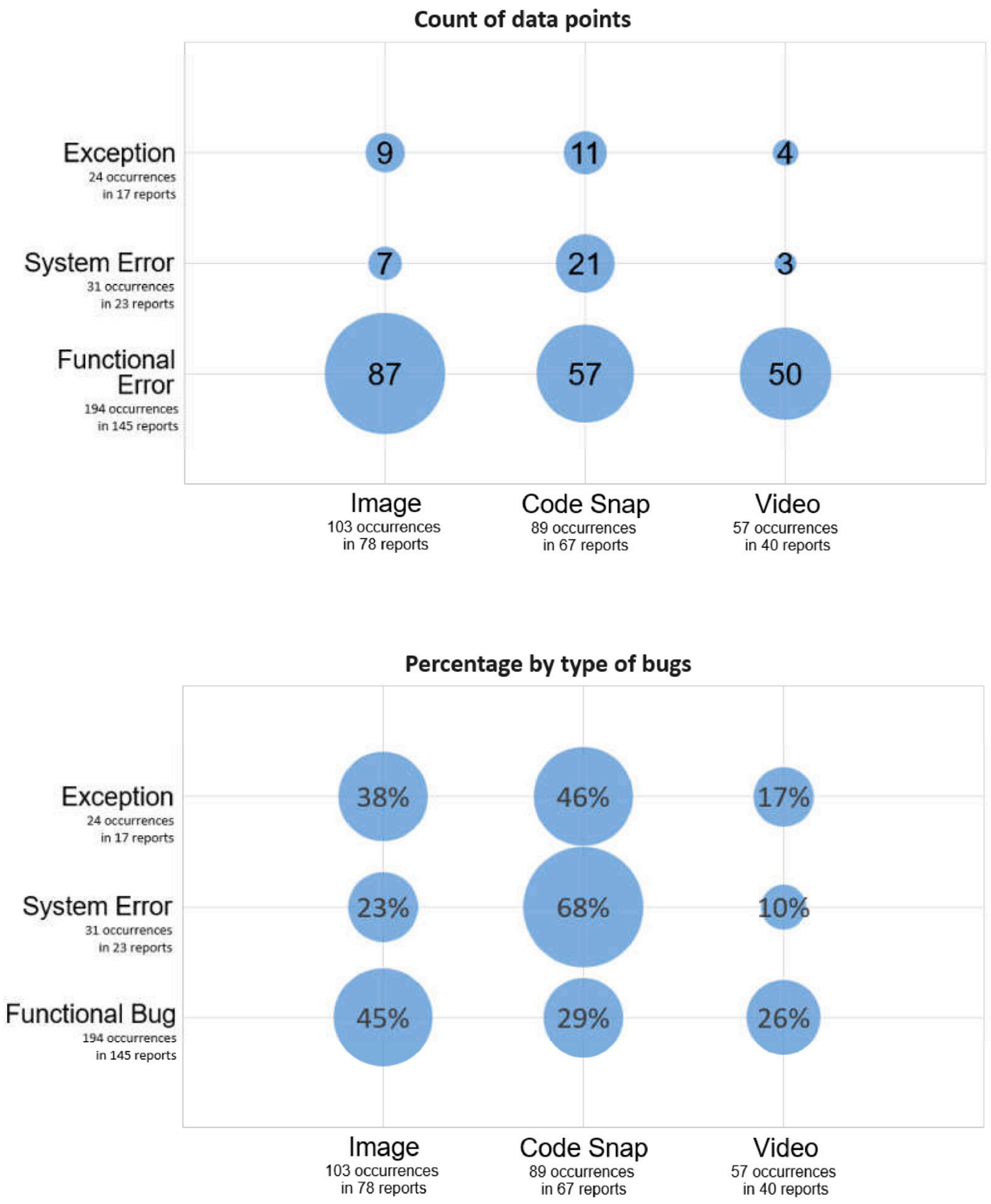


Fig. 13. Types of bugs vs. data formats (additional data).

the importance of *code snap* in providing detailed accounts of personal tool anomalies.

- **Work tools:** In the initial bug reports, *screenshots* were most utilized (60 reports), followed by *video* (18 reports) and *code snap* (7 reports). The additional data included a balanced increase in *code snap* and *screenshots*, with *video* reports remaining static. This trend indicates a preference for detailed reporting in work tool software, accommodating diverse documentation needs.
- **Development tools:** The initial bug reports had a strong presence of *screenshots* (49 reports) and *code snap* (19 reports), with *video* contributing a moderate number (15 reports). The additional data, however, showed *code snap* maintaining its stability, while *video* reports significantly decreased. This pattern emphasizes the continued relevance of textual (*code snap*) reports in documenting complex software issues.

Key insights for RQ4 (types of software):

- The initial versions of reports revealed a strong preference for *screenshots* in bug reporting across various software types.
- Later data indicated a trend shift toward more detailed reporting formats such as *code snap* and *video*.
- Significant increase in *video* data can be observed in bug reports of *E-Commerce software*.
- A balanced use of *code snap* and *screenshots* can be noted in *Work tools*.
- Findings highlight the evolving nature of bug reporting practices in response to the specific needs of different software types.

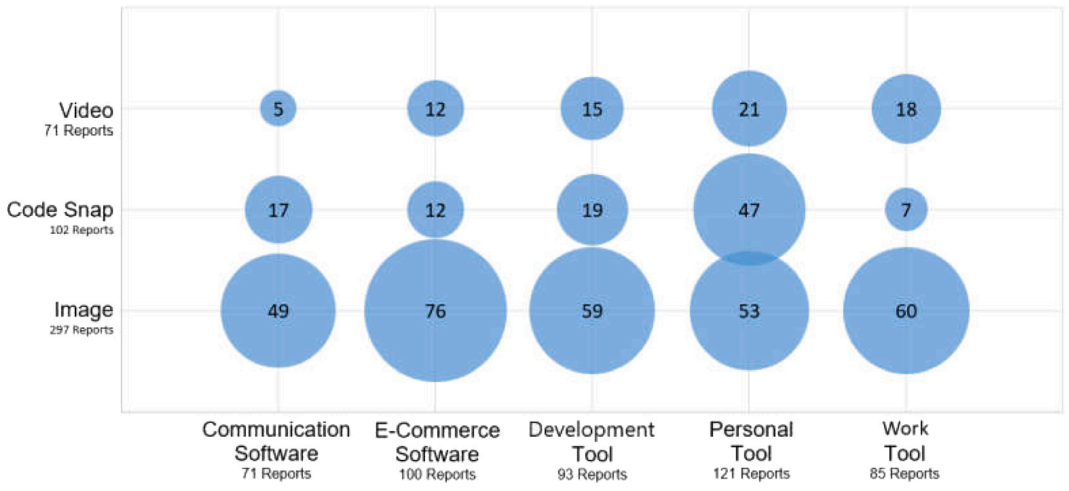


Fig. 14. Types of software vs. data formats (initial bug reports).

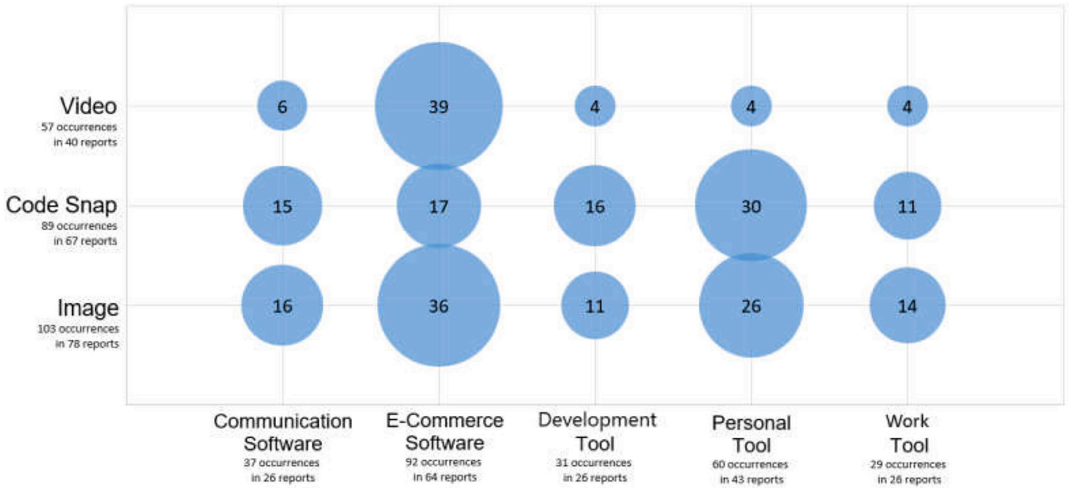


Fig. 15. Types of software vs. data formats (additional data).

4.6. RQ5: Data format based on type of information

To answer RQ5, we cross-referenced the data format with the types of information provided. Figs. 16 and 17 show a summary of data type against different types of data formats for the initial bug report and additional data:

- **Video data:** *Videos* initially take center stage in *steps to reproduce* with 66 reports, showing their effectiveness in visually narrating the problem reproduction. However, their presence as outcome information is not reported. They are not present in the initial report. However, as the bug report progresses, *videos* maintain their relevance in *steps to reproduce* with 31 data points. Also, their presence in end-user usage data grows significantly with 135 points. In terms of data density, *videos* are sparingly used in descriptions (5 points initially, 2 points in additional data) but show a strong presence in *end-user usage data* (135 points) and *steps to reproduce* (31 points), underscoring their capacity for visual storytelling.
- **Code snap data:** *Code snap* prominently features in the *Steps to reproduce* section of initial reports, with 47 instances, but it does not appear in outcome information initially. As the report evolves, *code snap* appears minimally in *steps to reproduce* with 22 points in additional data, and it takes a significant role in *diagnostic suggestions* with a notable increase to 298 reports. This transition highlights its adaptability, particularly in diagnostics. In terms of data density, *Code snap* contributes to descriptions with 15 data points initially and 15 in additional data, adds 23 points to *end-user usage data* in additional data, and becomes especially vital in *diagnostic suggestions* with a remarkable 298 data points.
- **Image data:** In initial reports, *images* emerge as a highly versatile resource, dominating in *steps to reproduce* with 121 reports and outcome information with 86 reports. This extensive usage illustrates their wide-ranging applicability in initial bug documentation. As the report extends, *images* show a balanced distribution across different information types in additional data, with 56 data points in *steps to reproduce*, 7 in outcome information, and a significant surge in *diagnostic suggestions* with 662 data points,

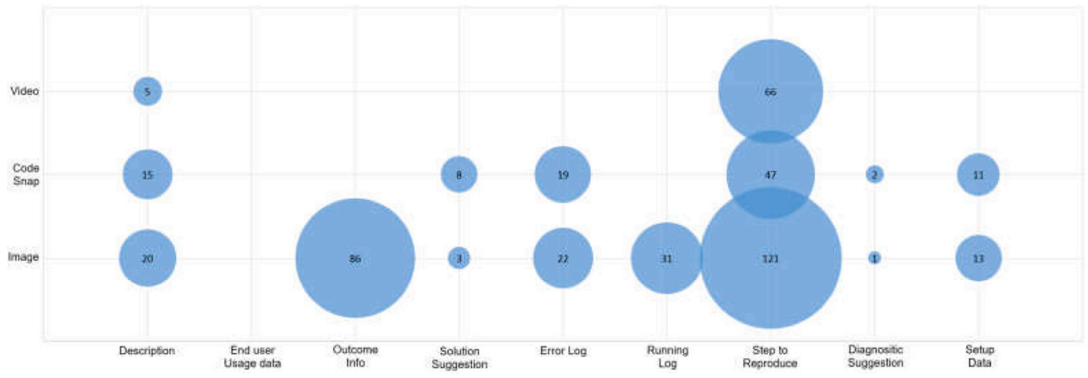


Fig. 16. Information type vs. data format (initial bug reports).

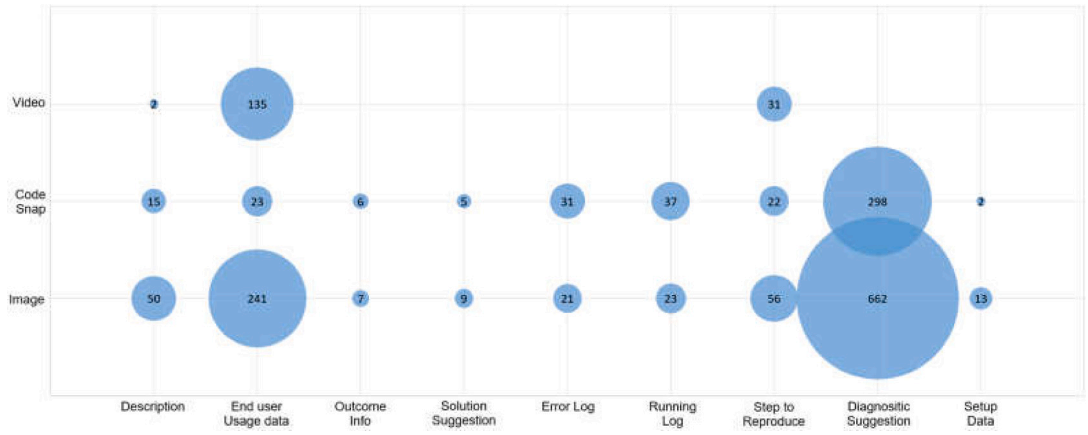


Fig. 17. Information type vs. data format (additional data).

reinforcing their utility in various contexts. Their widespread use across different data types for visual representation cements their role as an integral tool in visual documentation.

Key insight for RQ5:

- The systematic employment of *Code snap*, *Video*, and Screenshot formats in bug reports is evident.
- Each data format aligns with specific types of information, indicating unique application trends that contribute to the effective documentation of bugs.

5. Discussion

5.1. Need for systematic data collection

Our findings for RQ1 (Section 4.2) are in line with previous studies such as Rejaul (2019) and Johnson et al. (2022), confirming a prevalent issue: crucial information is often absent from initial bug reports. Additionally, our observations indicate that developers play a critical role in gathering the necessary details throughout the bug report's life cycle. Often, bug reports are submitted by end users who may lack the technical expertise required to provide comprehensive information. Additionally, the data formats provided by the customer are often constrained by the technical skills of the end user. As indicated by the results of RQ3 in Section 4.4, text and images are the most commonly used formats, while code snippets and videos are less frequently utilized. This is likely because providing data in the form of code snippets and videos requires a higher level of technical expertise. As a result, developers must invest additional time and effort to extract the necessary details from users, typically through back-and-forth communication. We suggest that strategies like the systematic data approach could mitigate this issue by enabling more effective data collection related to bug reports, thereby reducing the reliance on user input.

Some approaches for systematic data collection have been studied before, such as using a bug report template (Nikeghbal et al., 2023) or interactive chatbots (Song et al., 2023). In our study, we noticed that the completeness of bug reports is related to whether projects use a structured bug report template with minimum requirements for information other than attributes of the project, such as age or subscriber count. Only six out of the 10 projects have a structured template (see Table 1), while the remaining four projects provide guidelines for bug reporting without enforcing basic information requirements.

We, therefore, had a closer look at the type of information provided in the initial versions of bug reports (see Table 4), finding that projects with structured bug report templates (see Table 4), on average, have three times more tags to describe types of information compared to those without templates (note that, as discussed in Section 3.1.3, not all bug reports in those projects follow the template, as indicated by the percentages, less than 100%, in Table 4).

Specifically, projects with structured templates have an average of 2.58 tags to describe types of information in initial bug reports, whereas those without structured templates have an average of only 0.68 tags for types of information per report. However, even for the bug reports that use a template, many of them still miss key information from the initial report, as the information needed for different bug reports

Table 4
Use of bug report templates in the studied projects.

Name	Template fields	Template usage	Average information tag per report	Steps to reproduce	Outcome info	Setup data	Diagnostic suggestion	Error log	Running log	End user usage data	Solution suggestion
matomo	Description; Outcome info; Steps to reproduce; Setup data;	27%	1.56	31	25	49	1	25	5	0	9
magento2	Outcome info; Steps to reproduce; Setup data;	96%	3	94	94	94	0	7	6	0	1
zammad	Description; Outcome info; Steps to reproduce; Setup data;	91%	3.12	91	99	98	0	11	11	0	1
PrestaShop	Problem description; Steps to reproduce; Setup data;	100%	2.77	91	82	96	0	5	2	0	0
freeCode-Camp	Description; Outcome info; Setup data;	88%	1.95	37	75	70	0	3	7	0	3
mastodon	Outcome info; Steps to reproduce; Setup data;	98%	3.05	94	95	90	0	5	16	0	2
cal.com	None	32%	0.64	44	4	7	0	7	1	0	0
sourcegraph	None	16%	0.89	36	24	15	1	6	5	0	1
zulip	None	0%	0.57	20	12	10	1	5	4	1	3
parabol	None	36%	0.65	29	10	10	1	8	5	0	1

differs, and the template can only cover the minimum information needed in general.

Therefore, we believe a more systematic data collection approach is required to improve bug report quality. One of the promising directions to fill this gap is application monitoring.[8] An application monitoring system that can collect relevant data at runtime and provide relevant data when a bug report is filed will significantly reduce the time for bug reproduction (Wang et al., 2024).

5.2. Need for tailored data collection

The outcomes of RQ2 (Section 4.3), RQ4 (Section 4.5) and RQ5 (Section 4.6) suggests that different types of information and data formats are required in different contexts. For example, for types of information, *error logs* are notably prevalent in initial reports of *exceptions*, whereas bug reports for *system errors* often include *running logs*. Reports for *functional errors* typically feature detailed 'steps to reproduce'. Over time, reports for *functional errors* tend to accumulate more end-user usage data and *steps to reproduce*, while *exceptions* gather more *error logs* and *setup data*. This suggests that, in order to improve the quality and usefulness of bug reports, we not only need more data but also more relevant data in an appropriate format.

With this in mind, bug templates (discussed before) may need to be flexible to be altered based on the context of the bug report. However, this comes with certain challenges, e.g., it relies on the bug reporter to make the judgment about the context of a bug report and select the correct template. This is not always feasible due to many reasons, such as the limitation of the bug reporter's computer skills. Again, the automated application monitoring system will be able to serve relevant data collected based on the need of the bug report context. The bug reporter can create the initial input of the bug report following a generic template. Then, when developers start working on the bug, the data collected by the application monitoring system can provide additional information based on the judgment of the developer, who will have a much better technical understanding. The data collected can also be provided in a most suitable format (e.g., *error logs* will be provided in a searchable format instead of a screenshot *image*).

Furthermore, a more automated approach will be more maintainable compared to the information collection done by humans. For example, if a new feature is added to an app, we can update the application monitoring system to monitor the new module much easier than educating users about the module to provide correct data. We also can implement new approaches, e.g., AI-based bug report classification method, to help us understand the context of a bug better as data collected by application monitoring will be more consistent and easier to analyze (e.g., searchable *error logs*).

5.3. Developer understanding and skills

Based on the findings for RQ1 (Section 4.2), we conclude that the *diagnostic suggestions* coming from developers play a crucial role in the

[8] Note that due to the diversity of projects regarding age, size, etc. (see Table 1), we were not able to identify "correlations" between information missing in bug reports and other variables such as project maturity and community size.

additional data provided in bug reports throughout their lifetime. The key value of the information comes from the developers' understanding of how the program works and their skill set to analyze problems systematically.

Input from developers helps fill gaps in bug reports submitted by reporters with less advanced computer skills but who have more suggestions about the domain or specific information of the software itself. The information collected based on the *diagnostic suggestions* in bug reports can be very specific for the bug report and combine many context elements (e.g., user setup and program use when the bug occurred).

Even though in the previous sections we discussed how automated application monitoring will improve bug reports and the overall process of bug reproduction, application monitoring may not fully replace the human elements from bug reproduction anytime soon. We believe the role of an application monitoring system will help developers diagnose and reproduce bugs more effectively and efficiently. Still, the role of the developer is to improve the design of the system by making diagnostic judgments based on information from application monitoring. Notably, there are studies about automatic patch generation and program repair (Long and Rinard, 2016; Monperrus, 2014). However, automatic patch generation is not generalizable for all software types, especially web applications.

Furthermore, to manage bug reports effectively, a combination of technical and non-technical skills is required. Previous studies have also highlighted the crucial role of developer skills for bug reproduction (Breu et al., 2010). We believe that the following skills help developers:

- Technical Skills:
  - **Debugging Proficiency:** Proficient debugging skills are essential. Developers must be able to use debugging tools and understand stack traces to identify the root causes of issues (Zeller, 2005).
  - **Code Comprehension:** A strong understanding of the codebase is crucial. Developers need to quickly navigate large codebases, understand code dependencies, and recognize the impact of changes (Sedano, 2016).
- Non-Technical Skills:
  - **Communication:** Clear and concise communication is vital. Developers must articulate their findings and proposed solutions to both technical and non-technical stakeholders effectively (Breu et al., 2010).
  - **Collaboration:** Bugs are rarely fixed by individuals without consulting other stakeholders. Working well within a team and collaborating with other developers, testers, and product managers is essential. Collaborative skills ensure that knowledge is shared and collective problem-solving is leveraged (Vyas et al., 2014).
  - **Analytical Thinking:** Analytical skills enable developers to approach bug reports methodically, breaking down complex problems into manageable components and systematically addressing each aspect (Yılmaz and Kayalı, 2017).

By integrating these specific skills into the developer's role and providing avenues for continuous improvement and professional development, organizations can enhance their capability to manage and resolve bug reports efficiently, thereby improving the overall quality and reliability of their software products.

5.4. Importance of adding information for fixing bugs

In addition to the analysis presented in Section 4, we looked closely at bug reports that clearly indicated that the reported bug had been reproduced (i.e., confirmed). We identified bug reports that contained a developer's comment confirming the bug was reproduced (e.g., *"Thank you for working on this issue. Looks like this issue is already verified and confirmed"*.[9]) or were linked to an issue ticket confirming it (e.g., *"Thanks for reporting. I've created an internal ticket"*.[10]). We observed that bug reports acknowledged by developers tend to include more additional information compared to those that have not been confirmed. This finding indicates a notable information gap between the initial and final versions of bug reports. It appears that filling this information gap increases the likelihood of the bug being successfully reproduced.

5.5. Comparison to previous studies

Our findings confirm what other studies found (Rejaul, 2019; Johnson et al., 2022; Bhattacharya et al., 2013) and that missing information is a critical issue in bug reports. However, our study focused specifically on web applications and provided a more granular view by *manually* analyzing bug reports, allowing us to identify specific missing elements like diagnostic suggestions and end-user usage information that previous work did not identify.

While previous works (Karim, 2019; Fazzini et al., 2022) have used automated methods to improve bug report quality, our study emphasized the importance of manual analysis to identify nuanced deficiencies in bug reports. We highlight the critical role of developers' diagnostic skills in enhancing bug report quality over time. We also suggested systematic data collection techniques and the potential role of application monitoring systems, providing a practical perspective for enhancing bug report quality.

In summary, our study complements and extends previous research by providing a detailed, manual analysis of bug reports in web applications. It underscores the importance of missing information, systematic data collection, and developer skills for improving bug report quality.

6. Threats to validity

In this section, we present potential threats to the validity of this study.

External validity: We chose 10 popular open-source web projects from GitHub for our study. These projects come from different categories to reduce selection bias. However, many popular web projects on GitHub do not have publicly available bug reports. Hence, we cannot claim that our findings can be generalized to all web projects. Furthermore, we did not consider other bug reports from Github or other bug report systems, which might affect our results if other bug reports from Github or other report systems are looked at. Additionally, in proprietary software, bug reports are often distinguished based on internal reporters (e.g., developers) and external reporters (e.g., end users). This distinction is not present or not easily accessible in open-source software projects, affecting the generalizability of our results to proprietary web applications. In Section 7 we outline future work to complement findings based on open-source software with insights from commercial and proprietary software.

Furthermore, we categorized projects into five types of software defined by ourselves. This was done because there is no broadly accepted definition of types of software or industry standards. Analyzing more projects may lead to additional types of software not considered in our study.

Additionally, regarding our insights based on the types of software, even though our study included 200 bug reports per type, these bug

[9] https://github.com/magento/magento2/issues/15060#issuecomment-433591235

[10] https://github.com/magento/magento2/issues/10767#issuecomment-327443964

reports come from two projects for each type of software. This may limit the applicability of conclusions for RQ2 and RQ4.

Finally, we applied a manual analysis of bug reports. This limits the scalability of the analysis and, eventually, the generalizability of the findings. The manual examination of 1000 bug reports allowed for a thorough and nuanced understanding of the types of information commonly missing in bug reports. However, the manual analysis limits the size of the dataset that can be feasibly analyzed. Automated methods could have analyzed a larger dataset, capturing more diverse scenarios and edge cases across a broader range of projects and contexts. Future work may use our dataset and types of information to, for example, train techniques to automatically identify missing information in bug reports.

Internal validity: In our study we collected bug reports from 10 selected web projects. To determine a bug's lifetime, we used the bugs marked as "closed". Our results might be affected if any of those bugs are reopened in the future. Furthermore, confounding factors are related to the data analysis. We used a data tagging system to analyze the bug reports. However, it is not always clear what tag should be applied to a specific data point. For example, *solution suggestions* can sometimes be similar to *diagnostic suggestions*. To mitigate this, all tagging that was not clear were reviewed by multiple researchers.

Reliability: Sometimes, projects are using multiple bug tracking systems, and not all the bug reports are tracked in Github; we are running the risk that some bugs are missing from our data set. Furthermore, the maturity of a developer community or its size may also impact the quality of bug reports. We could not control these variables given the diverse types of projects we analyzed (see Table 1). However, we compared the "oldest" (matomo) and "youngest" (cal.com) communities (based on date of creation, see Table 1), as well as the largest (freeCodeCamp) and smallest (parabol) community (based on the number of stargazers). The missing information in their bug reports and missing types of information these four projects exhibited were consistent with the findings presented in Section 4. This suggests that the information provided in bug reports in our sample is not influenced by the maturity of the project, as no significant correlation was observed between the project's age or community size and the completeness of the bug reports. Furthermore, the use of templates in projects might impact findings. We have already discussed this in more detail in Section 5.1.

7. Conclusions and future work

In this study, we investigated the information missing in bug reports for web applications by manually analyzing 1000 bug reports from 10 popular open-source web-based applications. The study identified the types of information that are frequently missing in initial reports but added over time. Key findings include:

- Diagnostic Suggestions are often missing in initial reports, they are added over time, indicating their importance for bug resolution.
- End-User Information is frequently added to initial bug reports, suggesting that initial bug reports lack user context, which is crucial for reproducing and fixing bugs.
- Regardless of the types of bugs and software, all bug reports miss relevant information. This implies a general problem with initial bug reports. However, the concrete information type needed differs by the type of bugs and software. To summarize, based on the occurrence of types of information (and regardless of whether the types of information were already provided in the initial bug reports or added during the lifetime of a bug report), we observe the following:
  - Bug reports for functional errors require end-user usage data more often than reports for other types of bugs.
  - Bug reports for exceptions need error logs and setup data more frequently than reports for other types of bugs.
  - Bug reports for e-commerce software need steps to reproduce more frequently than bug reports for other types of software.
- Regarding data formats, text is the most common format, but screenshots and code snippets are also used. The format used to describe information in bug reports can impact the effectiveness of the bug report for bug reproduction and bug fixing.

Based on this study's findings, there are several directions for future work:

- Devising systematic data collection to improve the data provided in bug reports: We believe that an automated application monitoring that can collect runtime data and serve the developer based on the context of bug reports will significantly improve the effectiveness and efficiency of the bug reproduction process. Therefore, a follow-up Action Research study is planned to design and implement an application monitoring technique for a commercial web application. We will look into the data needed for different types of bugs and how they can be collected using the application monitoring technique (and supporting tools) designed based on the findings of this study. We will follow the Action Research guideline from ACM SIGSOFT Empirical Standards for Software Engineering[11] and in each iteration of the Action Research cycle, we will analyze the data on hand about the bug reproduction process and adjust the technique accordingly. The evaluation data will be collected during each iteration of the Action Research cycle, and the output will be used as input for the next study iteration. The Action Research study may also results in a tool and guidelines for systematic approaches to collect bug data and tailoring data collection strategies.
- Identifying and evaluating developers' skills that will impact the bug reproduction process: Our study found that diagnostic suggestions are one of the most frequent types of information added to bug reports. Those suggestions are provided based on the developers' skill and experience and their analysis of a bug. Future studies can take a deeper look at the technical and non-technical developer skills (tools, technologies, communication skills, analytical skills) required for bug reproduction.
- Extending the coverage of the study on bug reproducibility: In this study, we provided a deeper analysis of information missing from a bug report. However, as mentioned as a threat to validity, the coverage of the study can be improved. This leads to three potential study opportunities: (1) the use of automated quantitative analysis to cover more bug reports from more projects (including more projects per type of software) that have bug reports available publicly; (2) a study exploring more software types such as desktop application and embedded system; and (3) a study that looks into bug reproduction in commercial software. As most publicly accessible bug reports are from open-source projects, a study into proprietary software, such as commercial software projects, will help verify if findings from open-source projects still apply.

CRediT authorship contribution statement

Di Wang: Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Resources, Project administration, Methodology, Investigation, Data curation, Conceptualization. **Matthias Galster:** Writing – review & editing, Writing –

[11] https://github.com/acmsigsoft/EmpiricalStandards/blob/master/docs/ActionResearch.md

original draft, Validation, Supervision, Resources, Methodology, Investigation, Conceptualization. **Miguel Morales-Trujillo:** Writing – review & editing, Writing – original draft, Validation, Supervision, Methodology, Investigation, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgments

We thank the anonymous reviewers for their feedback that helped us improve the manuscript.

References

Avizienis, A., Laprie, J.-C., Randell, B., Landwehr, C., 2004. Basic concepts and taxonomy of dependable and secure computing. IEEE Trans. Dependable Secure Comput. 1 (1), 11–33. http://dx.doi.org/10.1109/TDSC.2004.2.

Bhattacharya, P., Ulanova, L., Neamtiu, I., Koduru, S.C., 2013. An empirical analysis of bug reports and bug fixing in open source android apps. In: 2013 17th European Conference on Software Maintenance and Reengineering. pp. 133–143. http://dx.doi.org/10.1109/CSMR.2013.23.

Breu, S., Premraj, R., Sillito, J., Zimmermann, T., 2010. Information needs in bug reports: improving cooperation between developers and users. In: Proceedings of the 2010 ACM Conference on Computer Supported Cooperative Work. CSCW '10, Association for Computing Machinery, New York, NY, USA, pp. 301–310. http://dx.doi.org/10.1145/1718918.1718973.

Davies, S., Roper, M., 2014. What's in a bug report? In: Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement. ESEM '14, Association for Computing Machinery, New York, NY, USA, http://dx.doi.org/10.1145/2652524.2652541.

Erfani Joorabchi, M., Mirzaaghaei, M., Mesbah, A., 2014. Works for me! characterizing non-reproducible bug reports. In: Proceedings of the 11th Working Conference on Mining Software Repositories. In: MSR 2014, Association for Computing Machinery, New York, NY, USA, pp. 62–71. http://dx.doi.org/10.1145/2597073.2597098.

Fazzini, M., Moran, K.P., Bernal-Cardenas, C., Wendland, T., Orso, A., Poshyvanyk, D., 2022. Enhancing mobile app bug reporting via real-time understanding of reproduction steps. IEEE Trans. Softw. Eng. http://dx.doi.org/10.1109/TSE.2022.3174028.

Feng, S., Chen, C., 2022. Gifdroid: Automated replay of visual bug reports for android apps. In: Proceedings of the 44th International Conference on Software Engineering. ICSE '22, Association for Computing Machinery, New York, NY, USA, pp. 1045–1057. http://dx.doi.org/10.1145/3510003.3510048.

Johnson, J., Mahmud, J., Wendland, T., Moran, K., Rubin, J., Fazzini, M., 2022. An empirical investigation into the reproduction of bug reports for android apps. In: 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering. SANER, pp. 321–322. http://dx.doi.org/10.1109/SANER53432.2022.00048.

Karim, M.R., 2019. Key features recommendation to improve bug reporting. In: 2019 IEEE/ACM International Conference on Software and System Processes. ICSSP, pp. 1–4. http://dx.doi.org/10.1109/ICSSP.2019.00010.

Karim, M.R., Ihara, A., Yang, X., Iida, H., Matsumoto, K., 2017. Understanding key features of high-impact bug reports. In: 2017 8th International Workshop on Empirical Software Engineering in Practice. IWESEP, pp. 53–58. http://dx.doi.org/10.1109/IWESEP.2017.17.

Long, F., Rinard, M., 2016. Automatic patch generation by learning correct code. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '16, Association for Computing Machinery, New York, NY, USA, pp. 298–312. http://dx.doi.org/10.1145/2837614.2837617.

Monperrus, M., 2014. A critical review of "automatic patch generation learned from human-written patches": essay on the problem statement and the evaluation of automatic software repair. In: Proceedings of the 36th International Conference on Software Engineering. In: ICSE 2014, Association for Computing Machinery, New York, NY, USA, pp. 234–242. http://dx.doi.org/10.1145/2568225.2568324.

Muzaki, R.A., Briliyant, O.C., Hasditama, M.A., Ritchi, H., 2020. Improving security of web-based application using ModSecurity and reverse proxy in web application firewall. In: 2020 International Workshop on Big Data and Information Security. IWBIS, pp. 85–90. http://dx.doi.org/10.1109/IWBIS50925.2020.9255601.

Nikeghbal, N., Hossein Kargaran, A., Heydarnoori, A., Schütze, H., 2023. GIRT-data: Sampling GitHub issue report templates. In: 2023 IEEE/ACM 20th International Conference on Mining Software Repositories. MSR, pp. 104–108. http://dx.doi.org/10.1109/MSR59073.2023.00026.

Rejaul, K.M., 2019. Key features recommendation to improve bug reporting. In: Proceedings of the International Conference on Software and System Processes. ICSSP '19, IEEE Press, pp. 1–4. http://dx.doi.org/10.1109/ICSSP.2019.00010.

Sedano, T., 2016. Code readability testing, an empirical study. In: 2016 IEEE 29th International Conference on Software Engineering Education and Training. CSEET, pp. 111–117. http://dx.doi.org/10.1109/CSEET.2016.36.

Song, Y., Mahmud, J., De Silva, N., Zhou, Y., Chaparro, O., Moran, K., Marcus, A., Poshyvanyk, D., 2023. Burt: A chatbot for interactive bug reporting. In: Proceedings of the 45th International Conference on Software Engineering: Companion Proceedings. ICSE '23, IEEE Press, pp. 170–174. http://dx.doi.org/10.1109/ICSE-Companion58688.2023.00048.

Vyas, D., Fritz, T., Shepherd, D., 2014. Bug reproduction: A collaborative practice within software maintenance activities. In: Rossitto, C., Ciolfi, L., Martin, D., Conein, B. (Eds.), COOP 2014 - Proceedings of the 11th International Conference on the Design of Cooperative Systems, 27-30 May 2014, Nice (France). Springer International Publishing, Cham, pp. 189–207.

Wang, D., Galster, M., Morales-Trujillo, M., 2024. Application monitoring for bug reproduction in web-based applications. J. Syst. Softw. 207, 111834. http://dx.doi.org/10.1016/j.jss.2023.111834, URL https://www.sciencedirect.com/science/article/pii/S0164121223002297.

Yılmaz, M., Kayalı, c., 2017. An exploratory study to assess analytical and logical thinking skills of the software practitioners using a gamification perspective. Süleyman Demirel Üniversitesi Fen Bilimleri Enstitüsü Dergisi 21 (1), 178–189. http://dx.doi.org/10.19113/sdufbed.39411.

Zeller, A., 2005. Why Programs Fail: A Guide to Systematic Debugging. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Zhang, T., Chen, J., Jiang, H., Luo, X., Xia, X., 2017. Bug report enrichment with application of automated fixer recommendation. In: Proceedings of the 25th International Conference on Program Comprehension. ICPC '17, IEEE Press, pp. 230–240. http://dx.doi.org/10.1109/ICPC.2017.28.

Zimmermann, T., Nagappan, N., Guo, P.J., Murphy, B., 2012. Characterizing and predicting which bugs get reopened. In: 2012 34th International Conference on Software Engineering. ICSE, pp. 1074–1083. http://dx.doi.org/10.1109/ICSE.2012.6227112.

Zimmermann, T., Premraj, R., Bettenburg, N., Just, S., Schroter, A., Weiss, C., 2010. What makes a good bug report? IEEE Trans. Softw. Eng. 36 (5), 618–643. http://dx.doi.org/10.1109/TSE.2010.63.

Di Wang is a Ph.D. student in the Department of Computer Science and Software Engineering at the University of Canterbury, New Zealand.

Matthias Galster is a Professor in the Department of Computer Science and Software Engineering at the University of Canterbury, New Zealand.

Miguel Morales-Trujillo is a Senior Lecturer in the Department of Computer Science and Software Engineering at the University of Canterbury, New Zealand. His main research area is Software Quality, Software Processes and Gamification in Software Engineering education.