W&M ScholarWorks

Dissertations, Theses, and Masters Projects

Theses, Dissertations, & Master Projects

2024

Automated Bug Report Management To Enhance Software Development

Yang Song
College of William and Mary - Arts & Sciences, songyang9446@gmail.com

Follow this and additional works at: https://scholarworks.wm.edu/etd

Part of the Computer Sciences Commons

Recommended Citation

Song, Yang, "Automated Bug Report Management To Enhance Software Development" (2024). *Dissertations, Theses, and Masters Projects.* William & Mary. Paper 1727787973.
https://dx.doi.org/10.21220/s2-wk2w-hn09

This Dissertation is brought to you for free and open access by the Theses, Dissertations, & Master Projects at W&M ScholarWorks. It has been accepted for inclusion in Dissertations, Theses, and Masters Projects by an authorized administrator of W&M ScholarWorks. For more information, please contact scholarworks@wm.edu.

Automated Bug Report Management to Enhance Software Development

Yang Song

Jiamusi, Heilongjiang, China

Bachelor of Mathematics, Sichuan University, China, 2016

A Dissertation presented to the Graduate Faculty of
The College of William and Mary in Virginia in Candidacy for the Degree of
Doctor of Philosophy

Department of Computer Science

The College of William and Mary in Virginia
August 2024

Copyright by Yang Song 2024

APPROVAL PAGE

This Dissertation is submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

Yang Song

Yang Song

Approved by the Committee, May 2024

Committee Chair
Assistant Professor Oscar Chaparro, Computer Science
College of William & Mary

Associate Professor Robert Michael Lewis, Computer Science
College of William & Mary

Professor Denys Poshyvanyk, Computer Science
College of William & Mary

Professor Weizhen Mao, Computer Science
College of William & Mary

Assistant Professor Kevin Moran, Computer Science
University of Central Florida

Research approved by

Protection of Human Subjects Committee

Protocol number(s): PHSC-2019-01-22-13374
PHSC-2020-04-27-14262

Date(s) of approval: 01/22/2019
04/27/2020

ABSTRACT

Bug report management is crucial yet challenging process that affects the efficiency of software development process. It involves reporting, triaging, detecting duplicates, assigning, localizing, fixing bugs, and thorough verification. The high volume and variety of bug reports complicate these tasks, highlighting the need for innovative solutions to improve the process and boost development efficiency. This dissertation explores the potential of automating the bug management process to optimize the effectiveness of software development and maintenance. It focuses on three key stages of bug management: reporting, assignment, and localization, presenting four innovative solutions for these phases.

First, it discusses the challenges faced by developers due to poor-quality bug reports on GitHub, often lacking crucial details. To address this, the dissertation leverages machine learning to automatically analyze user-written bug reports, identifying key elements of the software system. It aims to automate bug report analysis and inform reporters to provide the missing information timely, thereby enhancing the quality of bug reports and aiding developers in bug triage and resolution.

Second, the dissertation proposes an interactive bug reporting system for end-users, implemented as a task-oriented chatbot named Burt. This system guides users through the bug reporting process, offering real-time feedback on each element of a bug description and interactive suggestions to bridge the knowledge gap between end-users and developers. It is designed to make bug reporting more engaging and user-friendly while ensuring the generation of high-quality, informative reports.

Third, the dissertation investigates the efficacy of automated methods for recommending developers for bug reports in open-source software projects. It reveals that these methods do not perform consistently across different reports, leading to a proposal for using the most effective method for each report, assessed through machine learning. The findings suggest a gap in the understanding of real-world bug assignment processes and call for further research.

Lastly, the dissertation explores different deep learning models that can automatically localize buggy UI screens and components from the bug descriptions of mobile apps. This approach is critical for understanding, diagnosing, and resolving underlying bugs in GUI-centric software applications.

Together, these contributions present a comprehensive strategy for enhancing the automated bug report management process, promising significant improvements to the efficiency and effectiveness of the software development process.

TABLE OF CONTENTS

Acknowledgments vi

Dedication vii

List of Tables viii

List of Figures ix

1 Introduction 2

1.1 Contributions 5

1.1.1 Automated Bug Report Analysis and Quality Assessment . . . 5

1.1.2 Interactive Bug Reporting for End-Users 6

1.1.3 Automated Buggy Mobile App UI Localization 7

1.1.4 Bug Assignment Approaches Recommendation 8

1.2 Dissertation Organization 9

2 Background 10

2.1 Bug Report Management 10

2.2 Bug Reporting Tools 11

2.3 Bug Report Quality Analysis 12

2.4 Interactive Bug Reporting Systems 13

2.5 UI Representation Learning 14

3 Automating Bug Report Analysis and Quality Assessment 16

3.1 Introduction 17
3.2 BEE's Approach 18
3.2.1 BEE's Usage Scenario and Features 18
3.2.2 Under the Hood of BEE 20
3.2.2.1 Issue Classification 20
3.2.2.2 Sentence Classification 21
Sentence representation 21
Classification models 21
3.2.2.3 Detecting Missing Elements 22
3.2.3 Implementation 22
3.3 Evaluation 23
3.3.1 Data 23
3.3.2 Methodology 24
3.3.3 Results 25
3.4 Related work 26
3.5 Conclusions and Future work 27

4 Interactive Bug Reporting for (Android App) End-Users 29
4.1 Introduction 30
4.2 BURT: A Chatbot for Bug Reporting 32
4.2.1 Graphical User Interface (GUI) 33
4.2.2 Natural Language Parser (NL) 34
4.2.3 Dialogue Manager (DM) 36
4.2.3.1 Dialogue Flow for Bug Element Quality Checks (OB/E-B/S2R) 37
4.2.3.2 Dialogue Flow for Suggesting S2Rs 38
4.2.3.3 Collecting Input Values 39

4.2.4 Report Processing Engine (RP) 39
4.2.4.1 App Execution Model 39
4.2.4.2 Dialogue Quality Processor 41
4.2.4.3 S2R Response Predictor 44
4.2.5 BURT Implementation 44
4.3 Empirical evaluation design 45
4.3.1 Apps and Bug Dataset 46
4.3.2 RQ_1 & RQ_2: BURT's User Experience 47
4.3.2.1 BURT Bug Reporter Recruitment 47
4.3.2.2 Bug Assignment and Reporting 48
4.3.2.3 BURT's User Experience Assessment 48
4.3.3 RQ_3: BURT's Intrinsic Accuracy 49
4.3.4 RQ_4: BURT's Bug Report Quality 49
4.3.4.1 Itrac: A Web Form for Bug Reporting 50
4.3.4.2 Bug Reporting with Itrac 50
4.3.4.3 Measuring Bug Report Quality 51
4.4 Results and Analysis 51
4.4.1 RQ_1: BURT's Perceived Usefulness 52
4.4.2 RQ_2: BURT's Perceived Ease of Use 54
4.4.3 RQ_3: BURT's Intrinsic Accuracy 55
4.4.4 RQ_4: Bug Report Quality 57
4.5 Limitations and Threats to Validity 59
4.6 Related work 61
4.7 Conclusions 63
4.8 Data-Availability Statement 63

5 Recommending Bug Assignment Approaches for Individual Bug Reports 64

5.1 Introduction 65
5.2 Study 1: Bug Assignment on Individual Bug Reports 67
5.2.1 Dataset 67
5.2.2 Bug Assignment Approaches 68
5.2.3 Metrics and Methodology 69
5.2.4 Results 70
5.3 Study 2: Recommending the Best Performing Approach 71
5.3.1 Model Features 71
5.3.2 Models and Methodology 72
5.3.3 Results and Discussion 73
5.4 Threats to Validity 75
5.5 Related work 75
5.6 Conclusions and Future Work 76

6 Automated Localization of Buggy Mobile App UIs from Bug Descriptions 77
6.1 Introduction 78
6.2 Background, Problem, and Motivating Example 82
6.2.1 Bug Descriptions and App UI Screen/Components 82
6.2.2 Problem and Motivating Example 84
6.3 Study 1: Buggy UI Localization 86
6.3.1 Retrieval Approaches 86
6.3.2 Synthetic Dataset Construction 89
6.3.2.1 Synthetic OB Generation 90
6.3.2.2 Synthetic Retrieval Data 93
6.3.3 Real Dataset Construction 94
6.3.3.1 Bug Report Selection 95
6.3.3.2 Bug Description Annotation 96

6.3.3.3 Retrieval Corpus Collection 96
6.3.3.4 Ground Truth Construction 98
6.3.3.5 Summary of the Collected Retrieval Data 99
6.3.4 Approach Execution 100
6.3.4.1 GPT-4 Customized Prompting and Execution 100
6.3.4.2 Model Fine-tuning and Execution 106
6.3.5 Evaluation Metrics 107
6.3.6 Results 108
6.3.6.1 RQ_1: Screen Retrieval Results 108
6.3.6.2 RQ_2: Component Retrieval Results 111
6.3.6.3 RQ_3: Results vs. query qualities & retrieval difficulties 114
6.3.6.4 Discussion 116
6.4 Study 2: Improving Bug Localization 120
6.4.1 UI-based Bug Localization in Code 121
6.4.2 Using Buggy UIs for Bug Localization 122
6.4.3 Approach Execution, Dataset, and Metrics 124
6.4.4 Results 125
6.5 Threats to Validity 126
6.6 Related Work 127
6.7 Conclusions 128

7 Conclusion and Future Work 129
7.1 Main Contributions 130
7.2 The Vision for Future Work 131

Bibliography 133

ACKNOWLEDGMENTS

Above all else, I would like to express my deepest appreciation to my advisor, Professor. Oscar Chaparro, whose guidance, patience, and encouragement have been invaluable throughout this journey. His vast knowledge, profound insights, and tireless dedication to excellence have been an inspiration to me. I am immensely grateful for his patience, for challenging me to think critically, and for the countless hours he has spent reviewing and providing feedback on my work. In addition to his academic guidance, his kindness and wisdom have made a world of difference in my personal life. He has been a steady source of support, advice, and encouragement through the ups and downs. I am incredibly fortunate to have had him as not only the best advisor I've ever known, but also as an role model for my future career. This dissertation would not have been possible without his invaluable support, and for that, I am profoundly grateful.

I would like to thank my Ph.D. committee members, Professor Oscar Chaparro, Professor Robert Michael Lewis, Professor Denys Poshyvanyk, and Professor Weizhen Mao for their extremely valuable and constructive feedback.

I would also like to extend my thanks to my collaborators Junayed Mahmud, Ying Zhou, Antu Saha, Nadeeshan De Silva, Professor Kevin Moran, Professor Denys Poshyvanyk, Professor Andrian Marcus, and many others. I'm grateful to collaborate with these excellent researchers and to have received their valuable advice.

Finally, I would like to thank my family and friends for standing by me with constant support and encouragement all the way. In particular, I want to thank my cousin Ye Song for her steadfast support. And I must give a special shout-out to my constant companion, Corky, my beloved cat, who has been my loyal companion throughout this long, lonely and hard journey. Thank you, Corky!

In loving memory of my mom, your love has always been my spiritual support throughout my life. I will always love and miss you with all my heart.

This work is dedicated to you.

LIST OF TABLES

3.1 Detection performance of OB, EB, and S2R sentences and missing elements in bug reports 26

4.1 Apps and bug dataset 47

4.2 Questionnaire for evaluating Burt's user experience 49

4.3 Quality assessment results for bug reports (BRs) collected by Burt and Itrac 54

4.4 S2R quality by bug reporting experience 58

5.1 Bug assignment performance for each system 70

5.2 % of reports for which the approaches perform best 70

5.3 Bug assignment performance on the test set 74

6.1 Statistics of the synthetic dataset of OB/bug descriptions 92

6.2 Examples of templates used to generate synthetic OB/bug descriptions 94

6.3 Dataset statistics for SL and CL 99

6.4 Screen/component retrieval results on synthetic data 108

6.5 Screen Localization (SL) Results 110

6.6 Component Localization (CL) Results 114

6.7 Bug Loc. Performance via Buggy UI Localization 126

LIST OF FIGURES

3.1 Bee's feedback generated for bug report #95598 from Eclipse [2], which is submitted on GitHub [1]. 28

4.1 Burt's graphical user interface 34

4.2 Burt's dialogue flow for quality checking 36

4.3 Dialogue Flow for S2R Predictions 38

4.4 User experience results for Burt (Q1-Q5) 51

6.1 Bug report #191 from the WiFi Analyzer app [40] 83

6.2 Example of the UI screen/component localization process for an OB/bug description of the WifiAnalyzer app [40]. 83

6.3 The prompt template for screen localization 104

6.4 The prompt template for component localization 105

6.5 Buggy screen of a bug from Aegis App [5] 112

6.6 SL results for different query quality levels 115

6.7 CL results for different query quality levels 116

6.8 SL results for easy- and hard-to-retrieve tasks 117

6.9 CL results for easy- and hard-to-retrieve tasks 118

6.10 Buggy screen of a bug from GnuCash App [4] 119

Automated Bug Report Management to Enhance Software Development

Chapter 1

Introduction

The development and maintenance of software is a complicated process. Regardless of the advancements made in technology, software defects, also known as bugs, pose significant challenges to the efficiency of software development processes. The bug report or bug management process plays a crucial role in addressing these problems by providing a systematic process from identifying bugs to fixing bugs. The lifecycle of software bug management is a comprehensive process, integral to the development and maintenance of high-quality software systems [152]. This process consists of a series of important stages. It starts with the initial documentation or reporting of bugs, followed by a bug triage phase to assess their priority and severity, the detection of duplicate bug reports, and assigning the bugs to appropriate developers. After successfully reproducing the bug, it goes into the resolution phase. The resolution process starts with pinpointing the location of the bug within the software program, known as bug localization. This resolution then undergoes change design and impact analysis, leading to its implementation, which may include code refactoring. The cycle concludes with thorough verification and validation through code review and testing, and ends with the delivery of the solution via version control systems, building, deployment, and integration into the existing software environment [205, 124, 172, 239].

The bug management process is overwhelmed by the high volume and diversity of

bug reports, complicating tasks like reporting, assignment, localization, and resolving. This dissertation focuses on the potential of automating the bug management process to improve the efficiency of software development and maintenance, with a particular focus on the phases of the bug management lifecycle: bug reporting, bug assignment, and bug localization.

Bug Reporting Traditional bug reporting processes need users or testers to manually report a software defect [15, 20, 24, 29, 33, 34, 68]. This manual process, however, can be inefficient and error-prone, which may result in low-quality bug reports that hamper the bug resolution process. Over the past few years, research on automated bug reporting systems has made substantial progress [104, 161, 111, 123, 174, 163]. The goal is to minimize human errors and increase efficiency by enabling automatic identification, documentation, and reporting of software defects. Despite these advancements, there are still several challenges to overcome, and a considerable research gap exists in the understanding, design, and application of these systems. They were usually complex, creating a high barrier of entry for individuals with little or no experience. Moreover, they merely provided structured tools for the reporting process, without any interactive support to assist the users, making the process less engaging and user-friendly for users.

The dissertation is motivated by this context to address the challenges by investigating various methods for automating the bug reporting process. It seeks to provide a comprehensive understanding of the design and implementation of automated bug reporting systems and assesses their impact on software development efficiency.

Bug Assignment Complex software systems, comprising multiple modules or components, encounter numerous bugs and require maintenance by a large number of developers each with their own unique skills and knowledge. Manual bug assignment for a large-scale software system is not straightforward, is prone to human errors and inconsistencies, and often delays bug resolution, thus prompting the need for automated assignment techniques [239, 57, 197]. A variety of automated methods employing machine learning, deep learning, and information retrieval techniques have been developed to facilitate bug as-

signment, aiming to match bug reports with suitable developers by analyzing the content and context of each report [58, 199, 204, 214]. However, these methods often adopt a generalized approach that may not cater effectively to the unique characteristics of individual reports, leading to variability in their effectiveness.

The dissertation is motivated by this context to address the challenges by conducting an empirical study on different automated bug assignment techniques. It aims to assess the feasibility and efficiency of autonomously identifying and employing the most effective assignment technique for each specific report.

Bug Localization Researchers have been working to automate bug localization by developing approaches that automatically retrieve and rank potentially buggy files or code snippets, often treating the task as a text retrieval problem [53, 110, 148]. The fundamental limitation of these methods lies in the assumption that bug reports and source code exhibit term overlap, an assumption undermined by the considerable semantic gap between the bug report descriptions and code terminology. This challenge becomes particularly evident in the context of mobile applications [156], where bug resolution crucially involves the identification of the UI screens and components involved in the bug (Buggy UI Localization). Given the UI-centric nature of mobile apps, accurately locating the buggy UI elements is essential for effective bug resolution, highlighting the unique requirements and challenges of bug localization within this context.

This dissertation explores whether it is possible to improve Text Retrieval-based bug localization by information from the graphical user interface (GUI). It investigates different methods for automatically locating buggy UI elements based on bug descriptions, and also the effectiveness of those buggy UI localization methods to improve the traditional bug localization approaches.

In this dissertation, we introduce four works that target to address the above challenges. Firstly, we leverage machine learning to automatically analyze user-written bug reports and provide feedback to reporters and developers. Secondly, we propose an interactive bug reporting system for end-users, which provides guided reporting of essential bug

report elements, instant quality verification, and graphical suggestions. Thirdly, we investigate how effective automated bug assignment approaches are in suggesting developers for bug reports, emphasizing the need for further research to deepen our understanding of developers' bug report assignment methods and to enhance automated bug triage techniques. Finally, we empirically explore how deep learning models can automatically localize buggy UI screens and components from the bug descriptions of mobile apps, thus improving the traditional bug localization approaches.

1.1 Contributions

1.1.1 Automated Bug Report Analysis and Quality Assessment

Bug reports are crucial for developers in software bug identification and resolution [3, 66, 97, 67, 238, 237, 187, 182]. Key elements in these reports include the system's observed behavior (OB), steps to reproduce the bug (S2R), and the software's expected behavior (EB) [237, 67, 97, 85, 77]. However, the quality of these reports is far from satisfactory, with many being incomplete, unclear, or lacking in essential details. This was highlighted in 2016 when developers from over 1.3k open-source projects penned a letter to GitHub, expressing their frustration at often receiving bug reports without the S2R and system version [3]. This not only inflates the time and effort developers expend on triage and fixing tasks but can render bug reproduction and rectification impossible. The core issue is the absence of a robust feedback and quality verification mechanism within issue trackers.

To address this challenge, we present Bee, a plugin that enhances GitHub's issue tracker capabilities. It provides vital feedback to reporters and developers about OB, EB, and S2R in bug reports. Utilizing machine learning models, Bee analyzes submitted issues, distinguishing bug reports from enhancement suggestions or questions. It can identify sentences describing OB, EB, and S2R and detect the omission of these elements. It informs reporters about missing elements so that they can provide the information timely. Designed to assist developers by structuring bug descriptions and researchers via a public

web API for various tasks, Bee is universally applicable. It can analyze any bug report for any software system and is easy to install in any GitHub repository. Bee is designed to notify reporters of incomplete information in their bug reports, support developers in bug triage and resolution, and stimulate advancements in research on automated bug management.

1.1.2 Interactive Bug Reporting for End-Users

Bug report management, a key and expensive task in software engineering, mainly involves handling bugs that cannot be reported automatically. A significant proportion of these bugs, typically seen in open-source software, are associated with functional issues that need manual reporting. To facilitate effective bug triage and resolution, high-quality bug reports that describe at least the incorrect behavior, steps to reproduce the bug, and the expected correct behavior are required. However, generating such reports often presents challenges, primarily due to the knowledge gap between the end-users and the developers [163, 125]. This gap usually arises from end-users' lack of understanding of the software internals and what information developers find essential.

Current bug reporting systems do not adequately address this knowledge gap, as they often do not provide sufficient guidance on what and how information should be reported, and lack feedback mechanisms for the information provided by reporters. Consequently, the onus of providing high-quality information falls on the reporters. We propose that an interactive reporting solution, such as a chatbot, can help to bridge this knowledge gap by guiding end-users through the reporting process, providing interactive suggestions and immediate quality verification. In this work, we introduce a task-oriented dialogue system for Bug Reporting Burt, designed to provide real-time feedback for each element of a bug description, guide corrections where needed, and bridge the knowledge gap between end-users and developers. The proposed system has been evaluated empirically and has been found to improve the quality of bug reports. This work paves the way for a new approach to end-user bug reporting, transitioning from static to interactive bug reporting

systems, and is expected to serve as a foundation for a new generation of interactive bug reporting systems.

1.1.3 Automated Buggy Mobile App UI Localization

The range of tasks associated with bug report management exhibits considerable variation across different software domains. For mobile apps, one task that is central to fixing reported bugs is identifying the UI screens and components that cause and/or show a reported issue (*Buggy UI Localization*). Given the UI-driven nature of mobile apps, bugs typically manifest through the UI, hence the identification of buggy screens and components is important to localizing the buggy behavior and eventually fixing it. However, this task is challenging as developers must reason about bug descriptions, which are often low-quality, and the visual or code representations of UI screens.

This research explores the feasibility of automating the task of Buggy UI Localization through a comprehensive study that evaluates the capabilities of two textual and three multi-modal deep learning (DL) techniques. Buggy UI Localization is critical for assisting developers during bug triage and resolution and also plays a significant role in enhancing the effectiveness of existing automated bug report management techniques [231, 230, 105, 195, 78, 194]. For example, an automated Buggy UI Localization approach can facilitate the automated replication of reported bugs [231, 230], guide the generation of test cases to verify if the reported bugs can be replicated [105], and improve the quality of bug descriptions by aligning them more closely with UI screens/components. Notably, an automated buggy UI localization approach presents an opportunity for Burt (Section 1.1.2) to greatly enhance its ability to direct users in reporting bugs, by offering more precise feedback on UI elements related to the described bugs. Furthermore, this approach can augment textual bug reports with UI screenshots, assisting developers in more effectively understanding and resolving issues, and has potential applications in bug localization and duplicate bug report detection using visual information. Significantly, integrating automated Buggy UI Localization with the Burt dialogue system has

a potenitial to significantly improve the system's ability to guide users in reporting bugs by providing precise feedback on bug descriptions related to specific UI components.

The study explores the application of textual and multi-modal (visual-textual) deep learning or large language model (LLM) techniques to Buggy UI Localization, examining three specific models: two text-based model (SBert [179] and OpenAI-txt-embed [50]), and three multi-modal learning models (Clip [175], Blip [146] and GPT-4 [48]). To evaluate model effectiveness, we introduced two datasets: a synthetic one for model fine-tuning and a real dataset for testing, including manual annotations from actual bug reports. SBert, Clip, and Blip were fine-tuned on the synthetic dataset. Meanwhile, we directly utilize GPT-4 and OpenAI-txt-embed's APIs due to their exceptional zero-shot capabilities as OpenAI has not provided the option to fine-tune these models. Our results demonstrate the effectiveness of DL models, particularly LLMs, for automatically localizing buggy UI screens and components from the bug descriptions of mobile apps, while also highlighting Buggy UI Localization can be useful to automate and improve buggy code localization approaches.

1.1.4 Bug Assignment Approaches Recommendation

Software projects often face the challenge of efficiently triaging and solving numerous daily bug reports, necessitating the assignment of these reports to developers with the right expertise. While various automated approaches, utilizing machine learning or information retrieval techniques, have been proposed to recommend suitable developers based on bug report data (*e.g.*, [54, 58, 122, 199, 204, 214]), these methods tend to adopt a one-size-fits-all strategy that may not perform equally well across different reports due to their unique characteristics. An empirical study examining three distinct assignment techniques across thousands of bug reports from open-source systems revealed that no single method consistently outperforms others, highlighting the variability in their effectiveness. This finding led to the exploration of a novel approach named Mix, which aims to automatically identify and apply the most effective assignment technique for each individual

report. Although MIX shows promise in improving recommendation accuracy, it still falls short of maximizing potential benefits, suggesting the need for further research into better understanding developers' practical bug assignment behaviors and enhancing automated assignment methodologies.

1.2 Dissertation Organization

The remainder of this dissertation is structured as follows. Chapter 2 outlines the background and conducts a thorough literature review of our research topics. Chapter 3 introduces BEE, a tool that automatically analyzes user-written bug reports and provides feedback to reporters and developers. Chapter 4 presents BURT, an interactive bug reporting system for end-users, implemented as a task-oriented chatbot. Chapter 5 presents an empirical investigation into the effectiveness of automated bug assignment approaches. Chapter 6 shows an empirical study that explored the effectiveness of deep learning models for automatically localizing buggy UI screens and components from the bug descriptions of mobile apps, and evaluates their effectiveness in improving bug localization. Finally, Chapter 7 summarizes this dissertation and discusses the ongoing and future research works.

Chapter 2

Background

2.1 Bug Report Management

Bug report management plays a crucial role in software development and maintenance [152]. This process consists of the entire lifecycle of a bug, starting from its initial reporting to its eventual resolution and verification. The first step in this lifecycle is bug reporting, where users and developers encounter unforeseen issues such as crashes, errors, and malfunctions in functionality, and report these problems through issue or bug tracking systems. This initial step is crucial as it provides developers with the necessary information to diagnose, locate, and finally solve these issues within the code.

A comprehensive bug report usually consists of several key components: metadata about the bug (like severity, affected system version, and priority), a textual description of the issue in natural language, and any supplementary materials (screenshots, logs, etc.) that provide further context [237, 97, 187]. The textual bug description is crucial, as it outlines the bug's observed behavior, the steps to reproduce, and the expected behavior, aiding developers in the debugging process. Increasingly, the inclusion of visual aids like screenshots and recordings/videos, has become valuable in bug documentation. As a result, there's a growing trend towards the use of visual aids to more effectively capture and document bugs, particularly in mobile applications [91, 106, 142].

Following the submission of bug reports, the management process involves triage, assignment, resolution, and verification of these reports. During triage, reports are evaluated and prioritized based on their urgency and impact, ensuring that the most critical bugs are addressed promptly. The assignment phase assigns bugs to the appropriate developers for resolution, after which the resolved issues are verified to ensure they are adequately addressed. This organized management of bug reports enhances software quality by enabling timely identification and resolution of bugs and enhanced communication among developers and stakeholders. In essence, bug report management is essential at every stage of development, significantly contributing to the project's success by saving time, reducing development costs, and improving overall productivity.

2.2 Bug Reporting Tools

A range of issue/bug tracking tools are widely utilized for bug reporting in software by users. Some examples of these well-known web-based, general-purpose tools include Bugzilla [17], Jira [24], and GitHub's Issues [20], BugHerd [41], ClickUp [44], among others. Significantly, these tools provide a comprehensive solution for managing and resolving bugs and issues in software development, which goes beyond just reporting bugs. They typically provide templates with various fields for users to detail software bugs, including a textual bug description field and other custom fields, allowing for the organization of issues by different metrics (such as severity, priority and assignee and due date *etc.*) and customization of the bug reporting workflow to meet the needs of specific projects. Certain tools like BugHerd [41] focuses on actionable bug reporting by automating screenshot capture and gathering crucial details like browser, operating system, among others, to build actionable and detailed reports directly from the website. This capability significantly aids non-technical stakeholders in reporting issues accurately.

Besides these platforms, specialized applications exist that enhance functionality for various software types, particularly for mobile and web apps. Popular bug reporting

tools like Embrace [45], Instabug [49], Bugsee [33], and Shake [51] are tailored for mobile applications offering easy setup and integration via SDK installation. Meanwhile, tools like Bugsnag [42] and Gleap [47] are versatile, supporting both mobile and web applications. These tools feature user-friendly interfaces for both technical and non-technical users and automatically capture essential data during bugs or crashes, including screenshots, user interactions, network data, and logs. Additionally, they may provide advanced diagnostics, such as stack traces and device information, alongside direct user communication options like live chat. Some tools like Bugseee [33] emphasize visual aids, using screenshots, annotated media, and high-definition video recordings to enhance issue comprehension and resolution. Certain tools distinguish themselves with innovative features: Instabug [49] allows users to report a bug and send feedback by shaking their device, while Gleap [47] has an AI bot, Kai, that can automatically answer support questions.

2.3 Bug Report Quality Analysis

Despite the improvements made by introducing tools that allow users to attach screenshots or videos to bug reports, and even automatic data capture and recording features for mobile apps, textual descriptions remain important for developers to comprehend and solve bugs. This significance is underscored by the fact that not all users provide visual data, and advanced bug reporting tools with these capabilities are often not open-source, require in-app installation, and incur additional subscription costs. Consequently, textual bug description continues to be crucial, especially for most open-source tools like GitHub Issues. These platforms may offer templates to encourage the submission of essential information such as observed and expected behavior, along with steps to reproduce the bug. However, the submission of low-quality reports, including incomplete, unclear, or ambiguous ones, remains a common issue indicated by numerous studies and developers [101, 236, 74, 113, 237, 105, 136, 3].

Researchers have developed techniques to enhance the capture and management of

high-quality information in bug reports. These methods aim to identify essential elements within reports, analyze their quality, and provide feedback to contributors on potential issues. Specific approaches include predicting a bug report's quality based on readability and keyword presence proposed by Zimmermann *et al.* [237]. Hooimeijer *et al.* [121] measures report properties to forecast triage outcomes. *et al.* [221] identify invalid, duplicate, or incomplete reports through collaborative information. Additionally, Imran *et al.* [126] proposed to suggest follow-up questions for incomplete reports. *et al.* [78] evaluated the quality of the S2Rs in bug reports through the Euler tool.

Different from previous studies, our research focuses on enhancing the quality of text-based bug reports by pinpointing missing information such as observed/expected behavior and reproduction steps at the sentence level in reports submitted by end-users for any type of software (see Chapter 3). The primary goal is to furnish reporters with practical guidance on improving their descriptions, thereby enhancing the overall quality of bug reporting.

2.4 Interactive Bug Reporting Systems

As introduced in Section 2.2, most bug reporting tools are available for both developers and end-users, such as GitHub Issue, which allows developers and users to communicate through comments on submitted bug reports. Tools like Instabug [49] allows developers to send updates to users who reported an issue via in-app chats. Moreover, Gleap [47] supports live chat with users so that their questions can be answered in real-time. Nevertheless, these tools offer interactive features only after the submission of bug reports, which means they do not offer guidance or feedback during the bug reporting phase. Moreover, verifying the quality of bug reports through comments and in-app chats requires a non-negligible amount of manual effort. Thus, enabling users to write high-quality bug reports from the beginning could significantly reduce time and communication costs, thereby speeding up the bug resolution process.

Our research focuses on bugs related to Graphical User Interface (GUI) for mobile apps. In this field, there have been efforts to enhance the bug reporting systems for mobile apps. Moran *et al.* [163] proposed FUSION, a web-based system that allows the user to report the S2Rs graphically by selecting (via dropdown lists) images of the GUI components from the static/dynamic analysis of app screens and actions (taps, swipes, *etc.*) that can be applied on them. More recently, Fazzini *et al.* proposed EBUG [104], a mobile app bug reporting system similar to FUSION that suggests potential future S2Rs to the reporter while they are writing them.

This thesis introduces multiple improvements over existing methods by presenting a task-oriented chatbot designed for interactive bug reporting in Android applications. Distinct from current bug reporting systems, our chatbot assists end-users in identifying and reporting key elements of bug reports (such as Observed Behavior, Expected Behavior, and Steps to Reproduce). It also offers immediate feedback on any issues with the reported information and generates visual recommendations for the elements likely to be mentioned in the reports (see Chapter 4).

2.5 UI Representation Learning

Our new bug reporting system is tailored for GUI-related bugs in mobile applications. A key feature within this system is to locate/retrieve buggy UI elements through textual bug reports, which we refer to it as *Buggy UI Localization*. To accomplish this, understanding how to accurately represent UI elements is essential. Various GUI representation learning approaches have been developed for mobile or web applications. The goal of UI representation learning is to encode UI elements or related text into embeddings [147, 119, 61, 151], supporting downstream tasks like image captioning or text-image retrieval. A key application of UI representation learning is mapping (*a.k.a.* grounding) textual instructions to UI action/elements [173, 149, 217]. Pasupat *et al.* [173] evaluated three baseline models to ground natural language commands to web elements. Li *et al.* [149] addresses the

grounding problem by utilizing transformers models, based on three synthetic datasets for training. Other applications are UI image captioning [206, 165, 90] and UI component labeling [89, 150, 90], which typically leverage multi-modal models. Although this grounding task may appear similar to Buggy UI Retrieval, there are significant differences that make it difficult to adapt those models to our problem. For example, Li *et al.*'s approach [149] requires a sequence of screens where the instructions are performed, and then locating the corresponding UI component for each instruction. In contrast, our work focuses on identifying the buggy UI elements without any prior information about which screens are relevant. Furthermore, our study deals with bug descriptions, which are considerably more complicated than UI instructions [86].

Another category of models that can be utilized for our research objectives includes general-purpose deep learning models, such as multi-modal models like Vision-Language models. These supervised models are capable of processing and interpreting both images and text prompts, such as Clip [175], Blip [146], Flamingo [55], *etc.* Vision-Language models usually consists of various combinations of transformer-based encoders or decoders. Those models are trained by different pre-training objectives through different pre-training objectives, such as contrastive learning, masked language modeling, and image-text matching, among others. In addition to multi-modal models, our research has also made use of text-only models, such as SBert [179], a neural text-based language model derived from a pre-trained model Bert [99], as well as embedding models from OpenAI [50].

Recently, the advent of Large Language Models (LLMs) or Large Multimodal Models (LMMs), such as GPT-4 [48], Claude-3 [43], and Gemini [46], which have been trained on extremely large datasets, demonstrate remarkable capabilities in both natural language understanding and visual understanding. These models possess a significant zero-shot capability, allowing us to perform tasks without specific training or hard coding, even when pre-trained on general-purpose datasets [179, 146, 175]. Additionally, they can benefit from test-time techniques developed for LLMs, such as few-shot learning and chain-of-thought prompting [209, 227].

Chapter 3

Automating Bug Report Analysis and Quality Assessment

This work introduces Bee, a tool that automatically analyzes user-written bug reports and provides feedback to reporters and developers about the system's observed behavior (OB), expected behavior (EB), and the steps to reproduce the bug (S2R). Bee employs machine learning to (i) detect if an issue describes a bug, an enhancement, or a question; (ii) identify the structure of bug descriptions by automatically labeling the sentences that correspond to the OB, EB, or S2R; and (iii) detect when bug reports fail to provide these elements. Bee is integrated with GitHub and offers a public web API that researchers can use to investigate bug management tasks based on bug reports. We evaluated Bee's underlying models on more than 5k existing bug reports and found they can correctly detect OB, EB, and S2R sentences as well as missing information in bug reports. Bee is an open-source project that can be found at `https://git.io/JfFnN`. A screencast showing the full capabilities of Bee can be found at `https://youtu.be/8pC48f_hClw`. This work has been published at FSE 2020 [192]

3.1 Introduction

Bug reports are essential in helping developers triage, replicate, locate, and fix the bugs in the software [3, 66, 97, 67, 238, 237, 187, 182]. From the information that reporters provide in bug reports, the system's *observed (unexpected) behavior* (OB), the *steps to reproduce* the bug (S2R), and the software *expected behavior* (EB) are among the most important elements for developers [237, 67, 97, 85, 77]. These elements are typically expressed by end-users or developers in free-form natural language through issue trackers.

While these elements are essential, they are often incomplete, unclear, or not provided at all by the reporters [3, 74, 237]. Indeed, in 2016, developers from more than 1.3k open-source projects wrote a letter to GitHub expressing their frustration that bug reports are often submitted without the S2R and the system version [3]. The consequence of this is that developers often spend too much effort triaging and fixing the problems [237, 74, 113], and often, they cannot even reproduce and fix the bugs in the code [101, 236]. One of the main reasons for having low-quality bug reports is the lack of feedback and quality verification of issue trackers. In the GitHub letter [3], developers demand improvements to GitHub's issue tracker to ensure higher-quality bug reports. However, as of today, no major improvements have been made by GitHub. The alternative for some projects is to provide templates in the issues, explicitly asking for the OB, EB, S2R, and other information. Unfortunately, this approach does not guarantee that reporters will submit high-quality bug reports and developers still need to reach out to reporters asking for clarifications or more information.

In this work, we introduce BEE (**B**UG R**E**PORT ANALYZ**E**R), a tool that provides feedback to reporters and developers about the OB, EB, and S2R in bug reports. BEE is an app that extends the capabilities of GitHub's issue tracker, by analyzing incoming issues submitted by end-users on GitHub repositories. Through its machine learning models, BEE can detect if an issue reports a bug, an enhancement (*e.g.*, a feature), or a question. For bug reports, BEE can automatically identify the sentences that describe the

OB, EB, and/or S2R, and detect if the reporter does not provide any of these elements.

Bee adds comments and labels to the bug report to alert reporters (and developers) about missing elements so that they can provide the information timely. Bee is meant to assist developers, by structuring the bug descriptions via automated identification and labeling of OB, EB, and S2R sentences, allowing them to quickly spot these elements. Bee is also meant to assist researchers through a public web API for OB, EB, and S2R identification, which they can use for investigating and automating tasks that are based on these elements, such as bug reproduction [232, 137], test case generation [105], bug localization [80, 82], duplicate bug report detection [84], and bug report quality assessment [85, 77].

Bee can analyze any bug report, written in any textual form and format, for any software system. Bee can be installed in seconds, in any GitHub repository. Inspired by prior work (including ours) [237, 163, 224, 121, 221, 75, 77], Bee's main vision is to perform fine-grained quality assessment of bug reports and support reporters and developers in bug reporting and management.

3.2 BEE's Approach

Bee (**B**ug r**E**port analyz**E**r) is a GitHub app that analyzes GitHub issues submitted by end-users, and provides feedback to reporters and developers about the system's OB, EB, and S2R.

3.2.1 BEE's Usage Scenario and Features

Bee can be installed easily in any repository through Bee's installation website [11]. The users just have to follow a few steps for installing the app in their repositories. The current version of Bee does not require any configuration from the user.

Once installed, Bee analyzes any issue reported by the project users or developers, as shown in Figure 3.1. Since Bee focuses on bug reports, the first step of the tool, right

after an issue is submitted ①, is to automatically check if the issue describes a bug, as opposed to an enhancement (*e.g.*, a feature) or a question. If so, Bee tags the issue with the label Bug ② and proceeds with further analysis of the bug report. Figure 3.1 illustrates a report submitted on GitHub that describes a bug for the Eclipse project [1]. Such a bug was originally submitted by one developer on Eclipse's issue tracker [2]. If the issue is not a bug report, Bee tags the issue with a label corresponding to the type (enhancement or question), without further analyzing its content. This initial categorization of the issue is intended to help developers prioritize and manage the reported problems.

Bee analyzes the title and description of a bug report, focusing on the OB, EB, S2R. Bee can detect when any of these elements is not provided by the reporter. In that case, Bee makes a comment in the issue ③, alerting the reporter about the missing information and asking her to provide the information. Besides, Bee assigns the issue to the reporter ④ and tags the issue with the label info-needed ⑤. This feedback encourages reporters to provide the information needed by the developers. If all the three elements are provided by the user, Bee makes a comment indicating the bug report appears to be complete.

Bee provides additional feedback by structuring the bug description. This feature is meant to support developers (and reporters) in understanding and assessing the quality of the OB, EB, and S2R, by helping them easily identify these elements in the bug report. The bug report is structured automatically by Bee in an additional comment ⑥, which contains the bug title and description as provided in the original issue (with the same format), but with the sentences labeled as OB O, EB E, or S2R S (see Figure 3.1). Bee labels the sentences with the respective icon(s), at the end of the sentences, only if they convey the OB, EB, or S2R. Notice that a single sentence can convey one or more of the three types of information. The decision of labeling the sentences rather than re-organizing them into sections is made so that the (structured) bug description is easier to understand.

Bee supports any issue format, including GitHub's Markdown format, and does not impose any particular discourse on the users. This means that reporters can write their issues as they normally do. Bee treats each code snippet in the issue as a single piece of

text, and identifies if they provide information about the OB, EB, and S2R. When this is the case, BEE tags the snippets at the end of the code block. Reporters get alerted about BEE's feedback via email if they have email notifications enabled on GitHub.

Finally, BEE offers a public web API for automated OB/EB/S2R identification in textual documents. Users can send API requests containing any piece of text, BEE parses the text into sentences and returns them to the user, each one marked as OB, EB, and/or S2R. These elements can be incorporated in existing or new tools, and can be leveraged to perform automated bug localization [80, 82], duplicate bug report detection [84], bug report quality assessment [85, 77], and other tasks that rely on bug reports [232, 137, 105].

3.2.2 Under the Hood of BEE

BEE performs automated textual classification to determine the type of issue (bug, enhancement, or question) and the type of sentence (OB, EB, and/or S2R) for bug reports. Based on the sentence-level classification, BEE determines if the bug report does not contain any of the three elements.

3.2.2.1 Issue Classification

For classifying issues, BEE relies on the classification model of Ticket Tagger [135], which is based on *fastText* [133]. The model is a multi-class linear neural model that receives the set of n-grams (*i.e.*, sequences of n consecutive words) extracted from the issue title and description, and outputs the probability distribution of the issue over the predefined categories [135]. The model is pre-trained using 30k issues from 12k GitHub projects and classifies an issue into one of three categories: bug report, enhancement, or question. These categories are among the default labels of GitHub Issues [6]. The model can detect bug reports, enhancements, and questions with more than 82%, 76%, and 78% precision/recall, respectively, as indicated by its evaluation [135].

3.2.2.2 Sentence Classification

If an issue is detected as a bug report, BEE proceeds to classify each one of its sentences.

Sentence representation The sentences of a bug report are represented as binary vectors based on n-grams and part-of-speech (POS) tags. This representation captures sentence vocabulary, word types, relations between consecutive words, and syntactic patterns that can help with the classification. To represent the sentences, BEE first parses the text of the bug report using the Stanford CoreNLP library [157]. The bug description is split up into sentences, considering the title as a single sentence. Then, n-grams and POS tags are extracted from each sentence using tokenization, lemmatization, and POS tagging. BEE extracts {1,2,3}-grams and {1,2,3}-POS tags, which correspond to sequences of one, two, and three consecutive words and POS tags, respectively. Each element of the vector represents an n-gram or a POS tag and takes the value one (1) if the sentence contains the element, and the value zero (0) otherwise. The size of the vector for a sentence is 902,565, which is the number of n-grams + POS tags found in the data we used to train the classification models. BEE keeps an index of n-grams and POS tags for building the vector representation of the sentences. Stop word removal is not performed as (some of) these words can help determine the meaning of the sentences (OB, EB, and S2R) [85].

Classification models Inspired by our prior work [85], we use linear Support Vector Machines (SVMs) for classifying the sentences. SVMs are robust learning algorithms for high-dimensional and sparse data, used in text classification [129, 168]. Since the sentences in bug reports are relatively short, which means their vectors are highly sparse, SVMs are a good option for their classification.

Rather than relying on one multi-class SVM for classifying the sentences, BEE implements three binary SVMs, one for each of the information types (OB, EB, S2R). For example, the SVM for OB classifies a sentence as OB (the sentence conveys the OB) or non-OB (the sentence conveys other information than the OB). The SVMs for EB and

S2R work the same way for their respective elements. By using three classifiers, Bee can detect if a sentence conveys any combination of information elements (*e.g.*, OB and EB, OB and EB and S2R, *etc.*). Also, this approach allows us to evaluate Bee's classification performance easily.

Each SVM is trained using 116,084 sentences from 5,067 bug reports, where each sentence is represented as vectors, as described above. Section 3.3.1 provides more details about this dataset. Since the data is imbalanced, we train the SVMs by tuning their parameter $j = C_{+}/C_{-}$, which balances the cost factors for incorrect predictions of positive ($C+$) and negative sentences ($C-$) [168]. Larger j means higher penalty on false positives (*e.g.*, non-OB sentences predicted as OB), while lower j means higher penalty on false negatives (*e.g.*, OB sentences predicted as non-OB). We select the best parameters j during the training of all three models (see Section 3.3.2).

Once the sentences of a bug report are represented as vectors, Bee executes each SVM model on each sentence to determine its respective information type (OB, EB, S2R, a combination of these, or other information). Based on these results, Bee can tag each sentence on GitHub.

3.2.2.3 Detecting Missing Elements

Since each sentence of the bug report is identified as OB, EB, S2R, or other information, Bee can use these categories to determine if the entire report fails to provide any of the three elements. If no sentence is detected as OB/EB/S2R, then it means the bug report does not provide the OB/EB/S2R. If this is the case, Bee makes a comment about this situation, alerting the user and encouraging her to provide the missing information.

3.2.3 Implementation

Bee is mainly implemented using Node.js runtime environment, ensuring fast, real-time processing. Bee is built as a GitHub App, which uses GitHub's Webhooks and REST API that allows integration with GitHub's issues tracker [8]. These technologies are used

to obtain newly-submitted issues (their text, reporter, and other data), make comments on the issues, and assign users and labels to them. BEE's underlying classification models are implemented using the *fastText* [133] and the SVM^{light} frameworks [128], which are also known for being fast during training and execution. BEE currently analyzes a bug report in around 3-5 seconds.

3.3 Evaluation

We evaluated BEE's models to measure their expected performance in identifying the OB, EB, and S2R in bug reports. BEE's website contains the replication package of the evaluation [10].

3.3.1 Data

We compiled the bug reports used in our prior research [85, 80, 82, 84], which amount to 5,067 reports from 35 different software systems (*e.g.*, Eclipse, Firefox, Docker, WordPress Android, OpenJPA), spanning different domains (*e.g.*, data storage, software development, machine learning, virtualization, web browsing) and types (*e.g.*, desktop, web, mobile, libraries). The bug reports contain 116,084 sentences total (including the title), where the ones describing the OB, EB, and S2R are manually annotated to make up the ground truth. Nearly 12% of the sentences describe the OB, 2% describe the EB, and 6% describe the S2R; 82% of the sentences describe other types of information. The proportion of positive and negative instances for OB, EB, and S2R is close to 1:8, 1:54, and 1:16, respectively. This indicates the data is extremely imbalanced, which may lead to biased and less effective models. We address this issue in two ways: (1) we tune the parameter j of the SVMs; and (2) we use oversampling of the positive sentences (OB, EB, and S2R) using SMOTE [88], which generates synthetic instances nearby the positive sentences in the vector space. Although undersampling may also be helpful, it has the risk of discarding useful sentences for training the models, hence we prefer using oversampling. The average

(median) # of sentences in a bug report is 23 (9), and out of these, 3 (2) are marked as OB, 1 (1) is marked as EB, and 2 (2) are marked as S2R. On average (median), 21 (6) sentences are not marked as OB, EB, or S2R.

For evaluating the detection of missing elements, we use the same data and consider a bug report missing OB, EB, and S2R as one without sentences marked as OB, EB, and S2R, respectively. Only 2% of the bug reports do not provide any OB, and nearly 69% and 45% of them do not provide any EB and any S2R, respectively.

3.3.2 Methodology

We performed 10-fold cross validation (10-CV) [177, 76] to measure the expected detection accuracy of each of the three SVMs models (for OB, EB, and S2R). We randomly partitioned the data into 10 equal folds, using 8 folds for training, one fold for validation, and the remaining fold for testing. At each execution of the 10-CV approach, different folds compose the data sets, thus guaranteeing that all the sentences are used for model training, validation, and testing. The validation sets as disjoint and are used for SVM tuning. The testing sets are also disjoint and used for accuracy measurement. Note that we perform 10-CV on all the sentences in our data, as opposed to the sentences of each software system. We followed this approach since our prior work revealed similar performance between project-based and cross-project evaluation settings [85]. We applied SMOTE only to the training sets, which allowed measuring the models' accuracy on actual data, without synthetic sentences.

We measured the models' detection performance using precision, recall, accuracy, and F_1 score. We tuned the parameter j of each SVM model with the values 0.1, 0.2, ..., and 1, on each validation set. For each element type (OB, EB, S2R), we train 10 SVM models (*i.e.*, for 10 j values), select the best model using the validation set, and estimate its accuracy on the testing set using the selected metrics. The best model is the one achieving the highest F_1 score. This process is repeated 10 times, following the 10-CV approach. The best SVM parameter j is 0.2 for OB and S2R, and 0.1 for EB. The overall

results are computed by aggregating the true/false positives and negatives across the 10 testing sets, and then computing the metrics.

3.3.3 Results

The overall results of sentence classification are shown in Table 3.1. Bee's SVM models achieve 87%+ recall, which indicates their ability in correctly detecting OB, EB, and S2R sentences. Note that recall is most almost perfect for EB (about 98%). The results mean that Bee correctly detects the 3 OB, 1 EB, and 2 S2R sentences (out of 23 on average) expected in a typical bug report (according to our data). However, recall comes at the cost of precision, which is nearly 70% for all three models. The positive prediction rate of the models is 14.2%, 2.5%, and 7.4% for OB, EB, and S2R, respectively. This indicates that, on average, nearly 3, 1, and 2 sentences of a typical bug report are predicted as OB, EB, and S2R, however, the prediction is correct for ≈2.1, 0.7, and 1.4 sentences (on avg.), respectively.

Table 3.1 also shows the overall performance of Bee at detecting missing OB, EB, or S2R in an entire bug report. The results show low performance when detecting missing OB. However, only 2% of the bug reports are expected to lack this information, therefore, we anticipate a negligible effect of the misclassifications produced by the tool in practice. This observation is supported by Bee's high accuracy (≈97%). When detecting missing EB and S2R, Bee achieves substantially higher precision (93%+) and recall (70%+). Precision is almost perfect when detecting missing EB. Given the results, we can expect a few cases in which reporters are bothered with false alerts. The recall results mean that in about 1 (and 3) out of 10 bug reports, the tool fails to detect missing EB (and S2R). Since Bee is accurate in most bug reports, we expect Bee to have a positive effect on bug report quality and the bug resolution process.

In summary, Bee's models are conservative as they try not to miss any of the OB, EB, and S2R sentences in a bug report. This produces fewer false negatives, at the expense of more false positives (at sentence level). This phenomenon translates into high precision

(*i.e.*, few false alarms) and lower recall (*i.e.*, more misdetections) when detecting missing elements in entire bug reports.

Table 3.1: Detection performance of OB, EB, and S2R sentences and missing elements in bug reports

	Sentences			Missing elements		
	OB	**EB**	**S2R**	**OB**	**EB**	**S2R**
Precision	72.6%	70.0%	72.0%	33.3%	99.7%	93.4%
Recall	87.9%	98.4%	90.8%	33.7%	88.7%	70.4%
Accuracy	94.7%	99.2%	97.4%	97.4%	92.0%	84.4%

3.4 Related work

A few efforts have been made to automatically identify and extract the OB, EB, and S2R from bug reports, by using heuristics and machine learning [237, 67, 97, 85, 77, 228]. Examples of heuristics include matching keywords such as "observed results" to identify the OB, or using regular expressions to detect bullets as proxies to the S2R [237, 67, 97]. Since these approaches often fail to capture the diverse discourse [85] found in bug reports, machine-learning-based approaches, like the ones Bee implements, have been proposed [85, 77, 228]. Our prior work [85] used SVMs based on textual features to detect when bug reports lack the EB and S2R. More recently, SVM- and sequence-labeling-based techniques have been proposed to identify S2R sentences in bug reports from mobile apps [228, 77].

Many approaches have been proposed to classify issues into bug reports, feature requests, enhancements, questions, and other categories [135, 9, 113, 235, 56, 202, 216, 112]. These approaches typically implement machine learning models that use textual features for classification. Bee uses Ticket Tacker's pre-trained model [135] to identify if a newly-submitted issue reports a bug.

Different from prior work, Bee identifies the OB, EB, and S2R, at sentence level, in bug reports written by end-users in any form and for any software system. Bee's features enable many applications in bug management, as indicated by prior work [232, 137, 105,

80, 82, 84, 85, 77].

3.5 Conclusions and Future work

Bee is an open-source tool, integrated with GitHub, that uses machine learning to automatically (1) detect the type of user-written GitHub issues (bug report, enhancement, or question), (2) identify and label sentences describing the system's observed behavior (OB), expected behavior (OB), the steps to reproduce (S2R) the bug in bug reports, and (3) detect if these elements are not provided by the reporters. Bee is meant to alert reporters about missing information in their bug reports, assist developers on bug triage and resolution, and foster new research developments on automated bug management. The evaluation of Bee's underlying models, using 5k+ bug reports, provides evidence of its high accuracy in identifying the OB, EB, and S2R in bug reports. Given the results, we anticipate Bee can have a positive effect on bug report quality and bug management, yet this is to be confirmed by our planned user studies. Improvements to Bee include the implementation of a mechanism to automatically retrain Bee's models based on user feedback and autocompleting missing bug report elements.

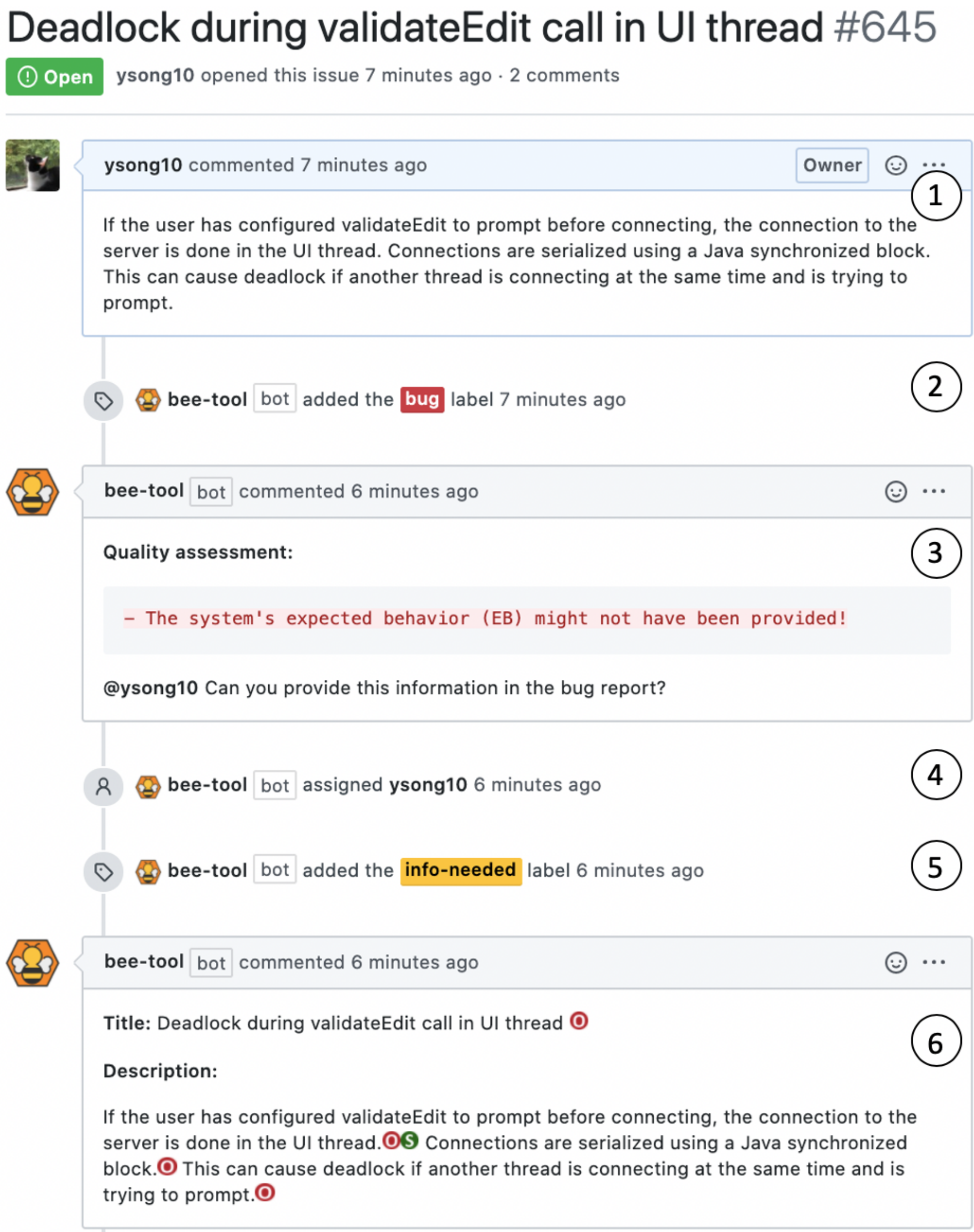

Figure 3.1: BEE's feedback generated for bug report #95598 from Eclipse [2], which is submitted on GitHub [1].

Chapter 4

Interactive Bug Reporting for (Android App) End-Users

Many software bugs are reported manually, particularly bugs that manifest themselves visually in the user interface. End-users typically report these bugs via app reviewing websites, issue trackers, or in-app built-in bug reporting tools, if available. While these systems have various features that facilitate bug reporting (*e.g.*, textual templates or forms), they often provide *limited* guidance, concrete feedback, or quality verification to end-users, who are often inexperienced at reporting bugs and submit low-quality bug reports that lead to excessive developer effort in bug report management.

We propose an interactive bug reporting system for end-users (Burt), implemented as a task-oriented chatbot. Unlike existing bug reporting systems, Burt provides guided reporting of essential bug report elements (*i.e.*, the observed behavior, expected behavior, and steps to reproduce the bug), instant quality verification, and graphical suggestions for these elements. We implemented a version of Burt for Android and conducted an empirical evaluation study with end-users, who reported 12 bugs from six Android apps studied in prior work. The reporters found that Burt's guidance and automated suggestions/clarifications are useful and Burt is easy to use. We found that Burt reports contain higher-quality information than reports collected via a template-based bug re-

porting system. Improvements to BURT, informed by the reporters, include support for various wordings to describe bug report elements and improved quality verification. Our work marks an important paradigm shift from static to interactive bug reporting for end-users. This work has been published at FSE 2022 [195].

4.1 Introduction

Bug report management is an important and costly software engineering activity. While certain types of bugs can be reported automatically via a known oracle (*e.g.*, crashes), recent studies have illustrated that more than half of the bugs reported in open source software relate to functional problems with no automatically identifiable oracle [201] and, hence, must be reported manually. High-quality bug reports are essential for bug triage and resolution and they are expected to describe *at minimum* the observed (incorrect) behavior (**OB**), the steps to reproduce the bug (**S2Rs**), and the expected (correct) software behavior (**EB**) [67, 143, 237].

One of the main difficulties that contributes to quality issues in end-user bug reporting is the *knowledge gap* between end-users and developers [163, 125]. That is, there is often a gap between what end-users *know* and what developers *need* [237], generally due to the fact that users are both unfamiliar with the internals of the software and with the explicit types of information that are important for developers (*e.g.*, the OB, EB, and S2Rs).

Most current reporting systems are not designed to address the above-mentioned knowledge gap between end-users and developers. In particular, current systems are typically lacking along two important dimensions: (1) they offer *limited guidance* related to *what* needs to be reported and *how* it needs to be reported; and (2) no *feedback* is offered to reporters on whether the information they provided is correct or complete. In consequence, given the *static nature* of these bug reporting interfaces, the burden of providing high-quality information rests on the reporters.

We posit that an *interactive* reporting solution can help to bridge the developer–end-

user knowledge gap. Inspired by prior work on question/answering systems for debugging [139], we argue that a conversational agent (*i.e.*, a chatbot) can successfully guide end-users through the reporting process, while offering interactive suggestions and instant quality verification.

In this work, we introduce and evaluate a task-oriented dialogue system for **BU**g Repo**RT**ing (or Burt) that is capable of providing instant feedback for each element of a bug description (*i.e.*, OB, EB, and S2Rs), while actively guiding corrections, where needed. Burt combines novel and state-of-the-art techniques for dynamic software analysis, natural language processing, and automated report quality assessment. We designed and developed the current version of Burt to work for Android apps, but its architecture is platform-agnostic and it can be instantiated, with some engineering effort, for other types of GUI-based applications (*e.g.*, web-based, desktop, or iOS-based). In particular, Burt constructs a graph of program states using both crowdsourced app usage data and automated GUI-based exploration techniques. The chatbot then parses and interprets end-user descriptions of various bug report elements by matching them to states and transitions in the constructed graph, and produces graphical suggestions regarding information that is likely to be reported (*e.g.*, the next S2Rs). Additionally, Burt recognizes when end-users provide incomplete or ambiguous information and suggests improvements or clarifications to the users. Traditional task-oriented chatbots typically have direct access to a structured and easily parseable knowledge-base [26]. In contrast, Burt is more complex, as it reconciles high-level descriptions provided by end users and matches these to technical program information, bridging the end-user to developer knowledge gap.

We evaluated Burt empirically, asking 18 end-users, with various levels of prior bug reporting experience, to report 12 bugs from six Android apps using a prototype implementation of Burt. We found that the guidance and automated suggestions/clarifications made by the chatbot were accurate, useful, and easy to use, and the collected bug reports are high-quality. We asked 18 additional end-users to report the same bugs with a template-based bug reporting system (Itrac) and compared the quality of these reports

to those reported with Burt. Burt reports have fewer incorrect and missing S2Rs than the Itrac reports. We also found that Burt helps novice bug reporters provide more correct steps, and experienced reporters avoid missing steps.

In summary, the contributions of this work are as follows:

- Burt, the first task-oriented, conversational agent that supports end-users in reporting bugs (currently for Android apps), with features such as automated suggestions, real-time feedback, prompts for information clarification, and graphical cues.
- The results of an empirical evaluation involving 36 end-users that investigates user experiences, preferences, and attributes of interactive bug reporting with Burt, as well as the quality of the resulting bug reports.

Our work opens the door to a new way of thinking about end-user bug reporting, using conversational agents, shifting the state of the art from *static* to *interactive* bug reporting. While Burt is a prototype, we expect that it will serve as the foundation for a new class of interactive bug reporting systems, combining elements of existing static systems with features of conversational agents [109].

4.2 Burt: A Chatbot for Bug Reporting

We propose a task-oriented chatbot for **BU**g Repo**RT**ing (Burt). Burt offers a variety of features for interactive bug reporting such as the ability to (i) guide the user in reporting essential bug report elements, (ii) check the quality of these elements, (iii) offer instant feedback about issues, and (iv) provide graphical suggestions.

Burt is designed to collect three key elements for developers during bug triage and resolution [237, 143, 187]: the *observed behavior* (**OB**), the *expected behavior* (**EB**), and the *steps to reproduce* the bug (**S2Rs**). Burt collects these from the user through a dialogue and generates a web-based bug report containing textual descriptions for these elements with attached screen captures of the system.

BURT's design consists of three main components, inspired by the typical architecture of task-oriented dialogue systems [109], which adapt techniques from automated program analysis and natural language processing to facilitate bug reporting. BURT's **Natural Language Parser (NL)** parses the relevant information from end-user responses to the chatbot. The **Dialogue Manager (DM)** dictates the structured conversation flow for BURT's reporting process and handles the presentation of multi-modal (*e.g.*, screenshots and text) information to the user. Finally, the **Report Processing Engine (RP)** maps information parsed from user responses to various states in a program execution model for a given app in order to assess bug element quality. The current version of BURT is designed for Android apps and builds its execution model using a combination of automated app exploration and crowdsourced user traces. In this section, we present BURT's components in detail.

4.2.1 Graphical User Interface (GUI)

We designed BURT as a web-based application that includes both a standard chatbot interface along with additional visual components as illustrated in Fig. 4.1. The *Chat Box* 1 allows the end-user to provide textual descriptions of the OB, EB, and S2Rs as well as interact with the graphical information that BURT displays (*e.g.*, recommendations of the next S2Rs via screenshots). The *Reported Steps Panel* 2 enumerates and displays the S2Rs that the user has reported. The textual description of the reported steps can be edited and the last reported step can be deleted, if the user makes a mistake and wishes to correct it. The *Screen Capture Panel* 3 displays screen captures of the last three S2Rs. The *Quick Action Panel* 4 provides buttons to finish reporting the bug, restart the bug reporting session, and (pre)view the bug report being created – these can be activated anytime. The *Tips Panel* 5 displays recommendations to end-users on how to use BURT and how to better express the OB, EB, and S2Rs. The tips change depending on the current stage of the conversation.

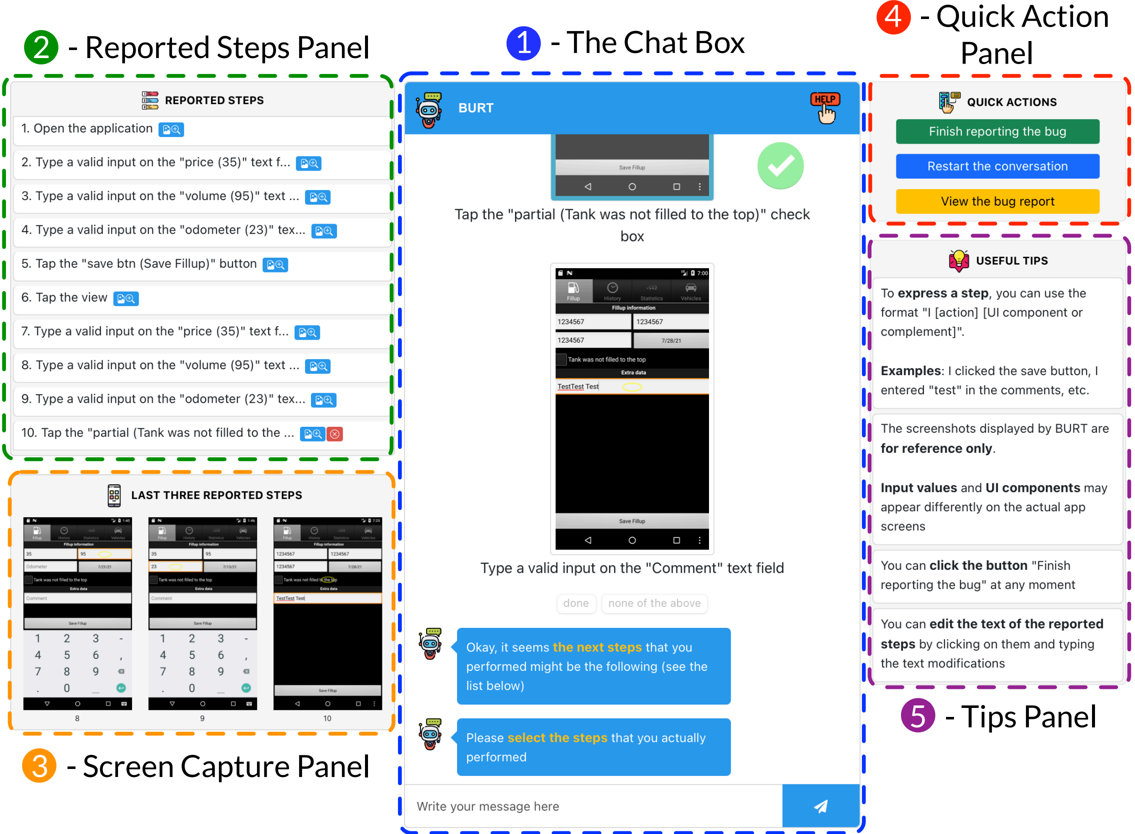

Figure 4.1: BURT's graphical user interface

4.2.2 Natural Language Parser (NL)

BURT parses the OB, EB, and S2R descriptions provided by end-users using dependency parsing via the Stanford CoreNLP toolkit [157]. This process obtains the tree of grammatical dependencies [31] between words in a sentence and extracts the relevant words from the tree. This parsing technique is needed by the Report Processing Engine to assess the quality of parsed bug report elements and to help direct the flow of conversation (see Sec. 4.2.4.2).

BURT first utilizes the heuristic-based approach introduced by Chaparro *et al.* [87] to identify the type of a sentence (*e.g.*, conditional, imperative, or passive voice) for each message received from the user. This approach implements heuristics (based on depen-

dency parsing and part-of-speech tagging [157]) to identify discourse patterns in OB, EB, and S2R descriptions [87]. Once the sentence type is identified, Burt executes a series of algorithms to extract the relevant words from the sentence, based on prior work on quality assessment of S2Rs [78]. In essence, we implemented 16 parsing algorithms that traverse the grammatical trees [31] of end-user sentences which have a different structure depending on the sentence type (*e.g.*, conditional or imperative). Each algorithm parses sentences of one type. All the 16 algorithms implemented for the different types of OB/EB/S2R sentences can be found in our online replication package [35].

Burt parses a single sentence using the following format:

`[subject] [action] [object] [preposition] [object2]` where the `subject` is the actor (*e.g.*, the user or an app component) performing the `action`, which is an operation or task (*e.g.*, tap, create, crash); the `object` is an "entity" directly affected by the `action`, and `object2` is another "entity" related to the object by the `preposition`. An "entity" is a noun phrase that may represent numeric/textual app input, domain concepts, GUI components, *etc.* Depending on the sentence, its type, and whether it describes an OB, EB, or S2R, the words (*e.g.*, the `subject`, `preposition` or `object2`) extracted from the entity are required or optional.

For example, for the Mileage Android app [27], the OB sentence *"The average fuel economy shows a NaN value"*, written in present tense, is parsed as `[average fuel economy] [shows] [NaN value]`. The EB sentence *"fuel economy statistics should be calculated correctly"*, which uses the modal "should", is parsed as `[average fuel economy] [is] [calculated]`. The S2R sentence *"Save the car fillup"*, written imperatively, is parsed as `[user] [saves] [car fillup]`.

Some sentences describe a combination of OB, EB and S2Rs in a single phrase. For example, the sentence *"The app stopped when I added a new time range"* describes both an OB and a S2R. This sentence is parsed by Burt as `[app] [stopped]` as the OB, and `[add] [new time range]` as the S2R. In this example, Burt extracts the S2R from the sentence as follows. First, it locates the adverb *"when"* in the parsed grammatical tree, then it

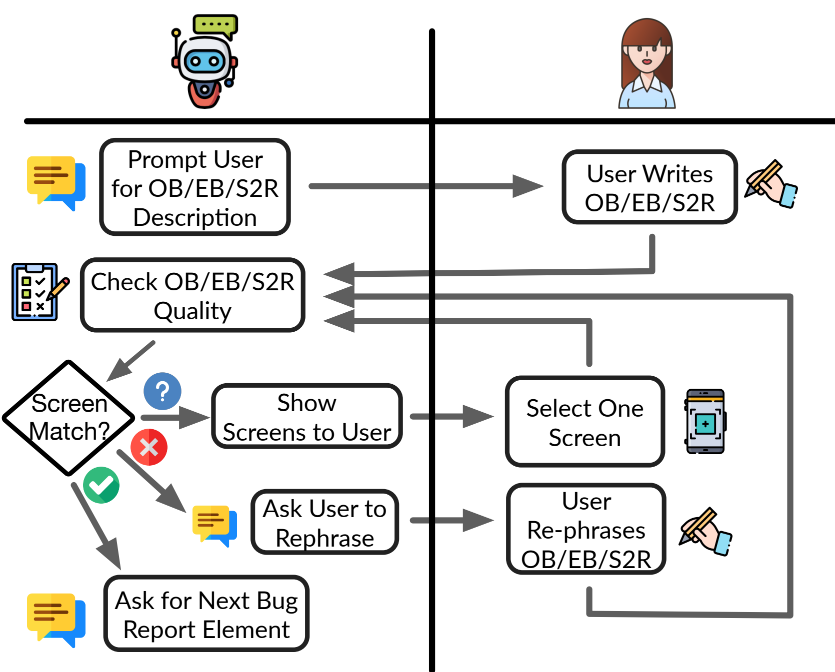

Figure 4.2: BURT's dialogue flow for quality checking

follows the relationship that leads to the verb *"add"* for which *"when"* is the adverbial modifier. Next, BURT locates the verb's nominal subject "I" and its direct object *"time range"*. If these relationships do not exist in the tree, the sentence is not conditional, as expected. Otherwise, BURT extracts the verb *"add"* as the `action` and the noun phrase *"time range"* as the `object`. In the end, this sentence is parsed as the S2R: `[add] [new time range]`.

When multiple sentences compose a single user message, BURT only parses the initial sentence. When BURT is unable to parse a user message (*e.g.*, because it cannot identify the `subject`), it asks the user to rephrase the sentence. BURT's *Tips Panel* 5 and user guide suggests patterns to the user to phrase the OB, EB, and S2Rs.

4.2.3 Dialogue Manager (DM)

BURT's dialogue flow consists of three main phases: OB, EB, and S2R collection. BURT's dialogue is multi-modal in nature, and is capable of suggesting both natural language

and graphical elements, such as screenshots, to help guide the user through the reporting process. The DM relies upon the RP engine to assess the quality of bug elements reported by end users (see Sec. 4.2.4.2). While Burt's dialogue flow proceeds linearly to capture each bug element (the OB, EB, and S2Rs, in that order), the dialogue flow is similar for all elements. There are two main dialogue flows that Burt navigates: (i) performing quality checks on written bug report elements (applies to all bug elements), and (ii) automated suggestion of S2Rs (for S2Rs only). Next we describe these two main dialogue flows.

4.2.3.1 Dialogue Flow for Bug Element Quality Checks (OB/EB/S2R)

Before the dialogue begins, a user must select the target app by clicking on its icon. Then, Burt's dialogue flow for quality checking, illustrated in a modified swimlane diagram in Figure 4.2, is initiated, starting with the OB. To begin the quality checking process, Burt prompts the user to provide the bug element (OB/EB/S2R). Burt automatically parses the description of the element and the RP engine verifies its quality (see Sec. 4.2.4.2).

If the OB/EB/S2R is matched to an app screen from Burt's execution model (see Sec. 4.2.4.1), Burt asks the user for confirmation of the matched screen. If the user confirms, Burt proceeds to the next phase of the conversation (*e.g.*, asking for the EB or next S2Rs), otherwise, Burt asks the user to rephrase the bug element.

If there are no app screen matches, Burt informs the user about the issue and asks her to rephrase the OB/EB/S2R. Once the user provides a new description, the quality verification procedure is re-executed. If there are *multiple* matches, Burt provides a list of up to five app screenshots (derived from the app execution model) that match the description. The user can then inspect the app screens and select the one that she believes best matches her description of the bug element. If none are selected, Burt suggests additional app screens if any. If the user selects one app screen, Burt saves the bug element description and screen, and proceeds to collecting the next bug element. After three unsuccessful attempts to provide a high quality OB description, Burt records the (last) provided OB description for bug report generation. This process proceeds for

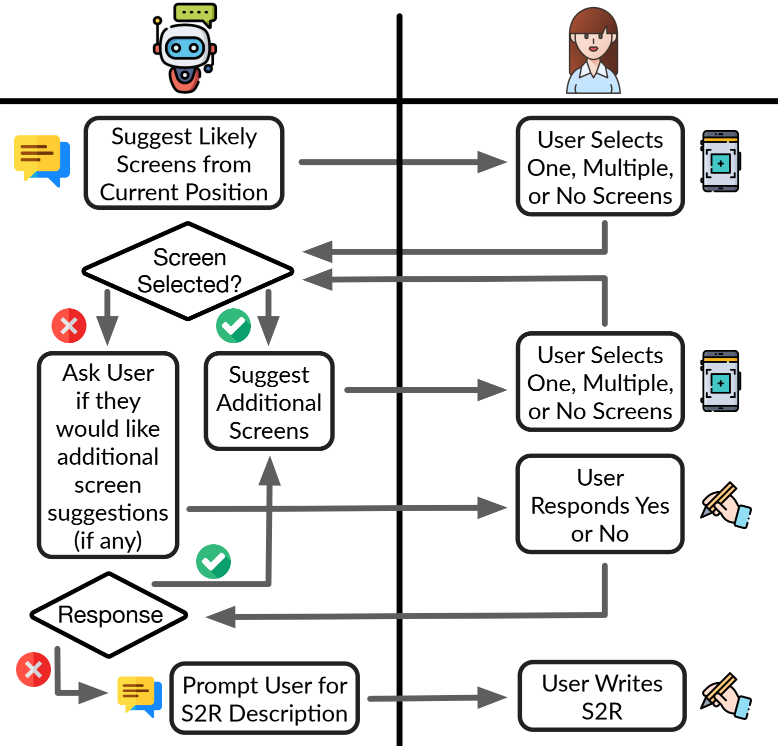

Figure 4.3: Dialogue Flow for S2R Predictions

each bug element starting with the OB. S2Rs are treated slightly differently since BURT can also *predict* S2Rs as we describe next.

4.2.3.2 Dialogue Flow for Suggesting S2Rs

BURT suggests potential next S2Rs that the user may have performed during actual app usage, depending on the last reported step and the user-selected screen that is having the problem, *i.e.*, the OB screen. Figure 4.3 illustrates this process. This dialogue flow uses a predictive algorithm that uses BURT's execution model (see Sec. 4.2.4.3). The suggestions are displayed as a list of app screens, each screen representing a S2R. Each S2R in the list displays the screen capture with a textual description placed below the image. The screen capture is visually annotated with a yellow oval highlighting the GUI component (*e.g.*, a button) executed by the step. The user can select none, one, or multiple of the suggested

S2Rs. When a S2R is selected, BURT suggests additional S2Rs if any. When none are selected and BURT has more suggestions, BURT asks the user if she wants to get more suggestions. If so, BURT displays them. Otherwise, BURT prompts the user to describe the next S2R.

4.2.3.3 Collecting Input Values

User input from type-like steps (*e.g.*, "*I entered 5 gallons*") are extracted by BURT from the `object` or `object2` of the parsed S2Rs, by identifying literal values or quoted text. If the input value is missing or generic (*i.e.*, not a literal or "text"), BURT prompts the user to provide the input. This is only activated if the matched S2R is confirmed by the user as a correct S2R.

4.2.4 Report Processing Engine (RP)

BURT's RP Engine is composed of three sub-components: (i) the *App Execution Model*, (ii) the *Dialogue Quality Processor* which maps parsed bug report elements to app states from the model, and (iii) the *S2R Response Predictor* which infers likely next S2Rs, given an existing set of S2Rs already mapped to the execution model.

4.2.4.1 App Execution Model

The app execution model is a graph that stores sequential GUI-level app interactions (*e.g.*, taps, types, or swipes performed on screen GUI components) and the app response to those interactions (*i.e.*, app screens). These interactions and app responses are produced using two strategies: (1) by executing an automated systematic app exploration adapted from CRASHSCOPE's *GUI-ripping Engine* [162, 164], and (2) by recording (crowdsourced) app usage information from app end-users or developers. Both the systematic app exploration and app usage data are collected *before* BURT is deployed for use.

App Execution Model Data Collection. This is BURT's plat-form-specific part

and would be constructed differently for non-Android applications. BURT uses a version of CRASHSCOPE's *GUI-ripping engine* [162, 164] to generate app execution data in the form of sequential interactions. CRASHSCOPE enables dynamic analysis of Android apps that utilizes a set of systematic exploration strategies (*e.g.*, top-down and bottom-up) and has been shown to exhibit comparable coverage to other automated mobile testing techniques [162]. For a detailed description of the engine, we refer the readers to Moran *et al.*'s previous work [162, 164]. As in prior work [111, 65, 96, 64, 116], we instantiate data collection by recording low-level app event traces using the `getevent`, `sendevent`, `uiautomator` utilities included in the Android OS and SDK.

Collecting crowdsourced user app usage data serves two main purposes: (1) increase the coverage of app states and screens in BURT's execution model; and (2) augment the model with scenarios that are common during normal app usage. Section 4.2.5 describes the procedure that we implemented to collect the crowdsourced data. Crowdsourced data collection leads to the same types of app events as the automatic app exploration does.

App Execution Model Structure. The execution model is a directed weighted graph $G = (V, E)$, where V is the set of unique *app screens* with complete GUI hierarchies [13], and E is a set of *app interactions* performed on the screens' GUI components. In this model, two screens with the same number, type, size, and hierarchical structure of GUI components are considered a single vertex. E is a set of unique tuples of the form (v_x, v_y, e, c), where e is an application event (tap, type, swipe, *etc.*) performed on a GUI component c from screen v_x, and v_y is the resulting screen right after the interaction execution. Each edge stores additional information about the interaction, such as the textual data input (only for *type* events) and the interaction execution order dictated by the app usage (manual or automatic). The graph's starting node has only one outgoing interaction, which corresponds to the application launch. A GUI component is uniquely represented by a type (*e.g.*, a button or a text field), an identifier, a label (*'OK'* or *'Cancel'*), and its size/position in the screen. Additional information about a component is stored in the graph, for example, the component description given by the developer,

the parent/children components, and an annotated screen capture of the app highlighting the GUI component being interacted with. The screen captures are used in the screen suggestions made by Burt (see Sec. 4.2.4.3).

The graph edges have a weight which indicates the likelihood of a given app interaction represented as a state transition. The weights are utilized by the *S2R Response Predictor* (see Sec. 4.2.4.3), which aims to suggest S2Rs that end-users would perform when normally using given app features. To enable accurate predictions, Burt assigns higher weights to interactions executed by humans than those executed automatically by CrashScope. To accomplish this, Burt sets the weight of an edge to the number of times it was executed in the collected usage data. If an edge is not executed by a human, but was executed by CrashScope's systematic exploration, then edge weight is set to one, even if CrashScope executes the same interaction multiple times. While this weight assignment scheme is straightforward, it proved to be effective (see Sec. 6.3.6).

4.2.4.2 Dialogue Quality Processor

Based on prior work [78], Burt's quality definition is based on the ability to match a textual bug description (OB, EB, or S2R) to the screens (states) and interactions (edges) of the execution model. A textual description is considered to be high-quality if it can be precisely matched to the execution model, otherwise it is deemed low-quality. This definition and Burt's dialogue features that prompt users to improve low-quality descriptions aim to reduce the knowledge gap between the reporters, who are unfamiliar with app internals and may not know how to express a bug, and developers, who define and implement the vocabulary captured in Burt's execution model.

Assessing OB Quality. Burt first builds a query to the app execution model by concatenating the non-empty elements from the parsed description, namely the `subject`, `action`, `object`, and `object2`. Then, it preprocesses the query using lemmatization [157] and attempts to retrieve all matching GUI components via an adapted version of the matching procedure proposed by prior work [78]. This procedure computes the similarity

score between the query and the elements from a GUI component, namely the component label, the description, and the ID specified by the original developer. The similarity is computed based on a normalized length of the longest common substring between query and the component elements. If such similarity is greater than or equal to 0.5, then there is a match, otherwise there is a mismatch. If the initial query does not match an app screen, BURT runs a different query by using only the `subject`, since, based on our experience, it may indicate a key GUI component that is directly related with a bug.

BURT keeps a list of the app screens with at least one matching GUI component. Such a list is sorted in increasing order by the distance between the starting state in the execution model and the matched state. If this list is empty, it means the OB description does not use vocabulary from the app screens and needs to be rephrased. If this list contains only one element, it is used to show the user the potential buggy app screen, which the user has to confirm as correct or incorrect. Otherwise, if the list contains multiple elements, it is used to display the possible buggy app screens so that the user decides the appropriate screen. The selected OB screen by the user is tracked in the execution model and is used for (1) EB description matching, (2) the prediction of the next S2Rs, and (3) asking the user if the last provided S2R is the last step to replicate the bug.

Assessing EB Quality. BURT performs the matching approach described above using the parsed EB description against the OB screen confirmed by the user. BURT assumes the OB screen is the one that should work correctly, therefore, it attempts to match the EB description to it. If the user did not select an OB screen, the EB matching is bypassed and the EB description is saved for generating the bug report. If the EB description does not match the OB screen, it means the vocabulary used in the EB description is different from the OB screen, and the EB description should be rephrased. However, rather than prompting the user to rephrase it, BURT asks the user if the OB screen is the one that should work correctly.

Assessing S2R Quality. BURT adapts the step resolution/matching algorithm proposed by Chaparro *et al.* [78] and performs exploration of the execution model driven by

the matching of the reported S2Rs. By default, BURT assumes the first S2R performed by a user is launching the app and the current graph state is set to be the first app screen that results from this operation.

For a provided S2R description, starting from the current state, BEE traverses the graph in a depth-first manner and performs step resolution on each state. Step resolution is the process of determining the most likely app interactions that the S2R refers to in a particular state (*i.e.*, app screen). The result is a set of *resolved interactions* for the S2R on the selected states. If the S2R resolution fails for these states (either with a mismatch or a multiple-match result), then it means that either: (1) there are app states not present in the execution model, or (2) the S2R description is of low-quality.

The *resolved interactions* are matched against the interactions (*i.e.*, the edges) from the graph, by matching their source state v_x, the event e, and the component c. If a pair of interactions match, then they are considered to be the same interaction. The matching returns a set of interactions from the graph that match the resolved ones. If this set is empty, it means that the resolved interactions were not covered by the app exploration and the quality assessment returns a low-quality result with a mismatch. If the reason for the mismatch is because of multiple-component or -event match (*i.e.*, the S2R description matches multiple GUI components or map to multiple events), BURT considers the S2R as ambiguous, and BURT indicates that the S2R's `action` corresponds to multiple events, or the `object` or `object2` match multiple GUI components. If there is a no-match, BURT specifies the problematic vocabulary from the S2R elements: `action`, `object`, `object2`, or any combination of these.

Otherwise, if the set of *resolved interactions* is not empty, BURT proceeds with selecting the most relevant interaction that corresponds to the S2R description, by selecting the one whose source state is the nearest to the current execution state in the graph.

4.2.4.3 S2R Response Predictor

BURT predicts the next S2Rs that a user may have performed in practice. The prediction is executed during the following dialogue scenarios (see Fig. 4.3): (1) when an OB screen from the execution model has been selected/confirmed by the user, (2) when the S2R collection phase starts, (3) right after the user confirms a matched S2R for her S2R description, or (4) when the user has already selected one or more S2Rs suggestions.

BURT implements a shortest-path approach to predict the next S2Rs. First, BURT determines the paths between the current graph state and the corresponding OB state. Then, BURT computes the likelihood score based on the execution model edge weights.

BURT uses the equation below to compute the score S_p of an n-edge path $p = \{w_1, w_2, ..., w_n\}$, with w_k being edge k's weight:

$$S_p = \frac{1}{n}\sum_k w_k + \frac{1}{n}$$

The first term in the sum is the average weight among all path edges and the second term is a factor that favors shorter paths.

Once the paths are ranked by their scores (in descending order), these are modified to include loops, *i.e.*, steps that lead to the same app screen (*e.g.*, *types* for providing input values). Then, only the first five steps for each path are selected. With only the first five steps, all unique paths are kept and only the top-2 paths are saved for being presented to the user. The first one is always presented and if the user does not select any of the steps as being the next S2Rs and wants more suggestions, the second path is presented next. Every time the user selects a suggested step as being the next step, the prediction/suggestion process restarts with new predictions.

4.2.5 BURT Implementation

BURT is currently implemented as a web application with two major components: the front-end, developed with the React Chatbot Kit [28], and the back-end, developed with Spring Boot [30]. BURT's implementation is tailored for Android applications, however, its underlying techniques are generic enough to be easily implemented for other types of

software — the App Execution Model Data Collection is the only platform-specific part.

To collect the crowdsourced app usage traces for BURT, two computer science students, who did not have knowledge of our studied bugs, were instructed to use the apps' features as they typically would do, and recorded traces that exercise key app features. Additionally, two researchers recorded sequences simulating app developers who test the apps. These traces were converted and merged into app execution models for each of the studied apps as described in Sec. 4.2.4.1. In practice, developers can utilize recorded tests, crowdsourced data, or automated app exploration techniques with a "one-time" cost for building the app execution model.

4.3 Empirical evaluation design

We conducted two user studies to evaluate: (1) BURT's perceived usefulness and usability; (2) BURT's intrinsic accuracy in performing bug report element quality verification and prediction; and (3) the quality of the bug reports collected with BURT, compared with reports collected by a template-based bug reporting system. We aim to answer the following research questions (RQs):

RQ$_1$: *What* BURT *features do reporters perceive as (not) useful?*

RQ$_2$: *What* BURT *features do reporters perceive as (not) easy to use?*

RQ$_3$: *What is the accuracy of* BURT *in performing bug element quality verification and prediction during the bug reporting process?*

RQ$_4$: *What is the quality of the bug reports collected by* BURT *compared to reports collected by a template-based bug reporting system?*

To answer the RQs, we selected a set of Android app bugs used in prior research (Sec. 4.3.1), and asked bug reporters to report these bugs using BURT and to evaluate their experience (Sec. 4.3.2). We analyzed the conversations the reporters had with BURT

and measured how accurate BURT was during the reporting process (Sec. 4.3.3). Then, we asked additional participants to report the same bugs with a template-based bug reporting system (Secs. 4.3.4.1 and 4.3.4.2), and analyzed the collected bug reports to measure their quality based on bug element correctness (Sec. 4.3.4.3). We present and discuss the results in Sec. 6.3.6. Our user studies were approved by an Institutional Review Board (IRB) and conducted remotely due to restrictions related to COVID-19.

4.3.1 Apps and Bug Dataset

We selected 12 Android app bugs from the bug dataset provided by Cooper *et al.* [96]. The apps in the dataset support different app domains and have been studied in prior research [65, 78, 163, 162]. The apps are: AntennaPod (APOD) [14] – a podcast manager, Time Tracker (TIME) [32] – a time-tracking app, Android Token (TOK) [12] – a one-time-password generation app, GnuCash (GNU) [21] – a personal finances manager, GrowTracker (GROW) [22] – a plant monitoring app, and Droid Weight (DROID) [19] – a personal weight tracking app. This dataset provides, for each bug, the APK installer that contains the bug, the description of the incorrect (observed) app behavior (OB), the expected app behavior (EB), and the (minimal) list of the steps to reproduce the bug (S2Rs).

From the 60 bugs (35 crashes and 25 non-crashes) in Cooper *et al.*'s dataset [96], we selected 12 bugs (7 crashes, 1 handled error, and 4 non-crashes) using a stratified random approach (see Table 4.1). We randomly selected two bugs for each of the six apps, ensuring that the bugs represent a variety of bug types that manifest visually on the device (crashes, GUI issues, functional bugs, *etc.*) and have a diverse number and type of S2Rs (taps, types, swipes, *etc.*). Six bugs contain $5-9$ (minimal) S2Rs, and six bugs contain $10-16$ (minimal) S2Rs (see Table 4.1). The 12 bugs are reproducible on a specific web-based Android emulator configuration (virtual Nexus 5X with Android 7.0 configured via the Appetize.io [16] service).

Table 4.1: Apps and bug dataset

App	Bug ID	# of S2Rs	Bug type
APOD	CC3	11	Incorrect color in GUI component
	RB	5	Error message on screen
DROID	CC5	7	Crash
	CC6	12	Crash
GNU	CC9	13	Duplicated GUI component
	RC	5	Crash
GROW	CC5	10	Crash
	RC	8	Crash
TIME	CC1	16	GUI component disappears
	CC4	9	Crash
TOK	CC2	10	Crash
	CC7	6	GUI component does not appear

4.3.2 RQ$_1$ & RQ$_2$: BURT's User Experience

We asked participants to report a selected subset of bugs using Burt, and evaluate their experience via an online questionnaire.

4.3.2.1 BURT Bug Reporter Recruitment

We reached out to 36 potential participants with mixed experience in bug reporting from our personal network, who were not involved in or aware of the purpose of this work. They were offered a $15 USD gift card for participation. From these, 24 users completed the study and data from six participants was discarded due to low-effort answers, thus resulting in valid responses from 18 participants. Four of the six participants did not treat the study seriously, that is, they submitted incomplete reports (e.g., only the OB was reported) and answered all survey questions with the same response. The remaining two participants reported completely different bugs to the ones assigned. Five participants had not reported a software bug before, nine had reported five or fewer bugs, and the remaining four had reported more than five bugs. The participants were unfamiliar with Burt and the selected apps/bugs.

4.3.2.2 Bug Assignment and Reporting

Each of the 18 participants was randomly assigned to report three bugs (from the 12 selected) with BURT, each bug corresponding to a distinct app. The reporters were instructed to report the bugs in a given (random) order to account for potential learning biases. The bug reporting procedure consisted of five tasks which included the users: (i) watching a short instructional video that explained how to use BURT via an example; (ii) familiarizing themselves with the apps on the web-based emulator; (iii) watching a video demonstrating the observed and expected behavior for each assigned bug (with annotations to ensure proper understanding); (iv) reproducing the bugs on the web-based emulator; and (v) reporting each bug with BURT. We aimed to control for participant understanding of the bugs so that the effect of potential misunderstandings was minimized.

4.3.2.3 BURT's User Experience Assessment

After the participants reported the three assigned bugs, they answered an online questionnaire that was meant to assess BURT's usefulness and ease of use and to obtain feedback for potential improvements to BURT. Table 4.2 shows the questions asked to the participants, which are inspired by the PARADISE [114] evaluation framework.

To address RQ_1, we focused on evaluating BURT's four main features: (1) BURT's app screen suggestions for the OB, EB, and S2Rs; (2) BURT's ability to parse and match the OB, EB, and S2R descriptions provided by the user; (3) BURT's messages and questions given to the user; and (4) BURT's panel of reported S2Rs, which allows the user to visualize and edit the reported S2Rs. Questions Q1-Q5 in Table 4.2 aim to address RQ_1 and used a 5-level Likert scale [171]. We asked the participants to (optionally) provide a justification/rationale for their answers. Each bug involved multiple screen suggestions, OB/EB/S2R user descriptions, and BURT messages/questions. Questions Q1-Q3 refer to the frequency of these user interactions with BURT.

To address RQ_2, the reporters assessed BURT's overall ease of use (Q5) and indicated

Table 4.2: Questionnaire for evaluating BURT's user experience

ID	Question
Q1	How often were BURT's screen suggestions useful?
Q2	How often was BURT able to understand your OB/EB/S2Rs?
Q3	How often were you able to understand BURT's messages/questions?
Q4	Was BURT's panel of reported steps useful?
Q5	How easy to use was BURT overall?
Q6	Which of BURT's features did you find easy/difficult to use?
Q7	What additional functionality (if any) would you like to see in BURT?

BURT's specific features that were easy or difficult to use for them (Q6). Q5 used a used a 5-level Likert scale and Q6 requested an open-ended response. The reporters were also asked to indicate additional features they would like to see in BEE (Q7). Additional open-ended questions were asked (not shown in Table 4.2) to obtain feedback on how to improve BURT.

4.3.3 RQ_3: BURT's Intrinsic Accuracy

To answer RQ_3, we analyzed the conversations that the reporters had with BURT to determine: (1) how often BURT was able to correctly match OB/EB/S2R descriptions to the app execution model as confirmed by the reporters; and (2) how often the user selected one or more of the suggested app screens as being correct (*i.e.*, they match the reporters' OB/EB/S2R descriptions). We computed statistics on the (meta)data that BURT collected from the conversations, such as, the type of messages that BURT asked and the type of user responses (as defined by BURT's Dialogue Manager – see Sec. 4.2.3).

4.3.4 RQ_4: BURT's Bug Report Quality

We describe the methodology to answer RQ_4 in this section.

4.3.4.1 Itrac: A Web Form for Bug Reporting

We implemented a web/template-based bug reporting interface, called Itrac, using Qualtrics [36]. Itrac offers the same features of professional issue trackers (*e.g.*, GitHub Issues [20] or JIRA [24]), for reporting the OB, EB, and S2Rs. Specfically, Itrac provides text boxes with explicit prompts that ask for the bug summary/title and the OB, EB, and S2Rs. In addition, Itrac prompts the reporter to provide the S2Rs using a numbered list (via a given template). The reporters can write freely their own bug descriptions in the text boxes and also attach images/files. We use Itrac rather than an existing professional issue tracker to simplify the reporting process for the reporters because they can use Itrac without having to log into a service.

4.3.4.2 Bug Reporting with Itrac

Following the methodology described in Sect. 4.3.2.1, we recruited 18 more end-users, who did not participate in the Burt study, and asked them to report a subset of bugs using Itrac. These reporters did not know about Burt, Itrac, and the selected apps/bugs, and had a similar bug reporting experience to that of the group who reported the bugs with Burt. Five of the new reporters had not previously reported a software bug, eight had reported one to five bugs, and the remaining five had reported more than five bugs.

We assigned the same sets of three bugs used in the Burt study to the new users (trying to match the bug reporting experience) and instructed them to report the bugs using Itrac in the same order from before. Prior to reporting the bugs, the participants were instructed to: (i) familiarize themselves with the apps by using them on the web-based emulator; (ii) watch a video demonstrating the bugs (with annotations to ensure proper understanding); and (iii) reproducing the bugs on the web-based emulator.

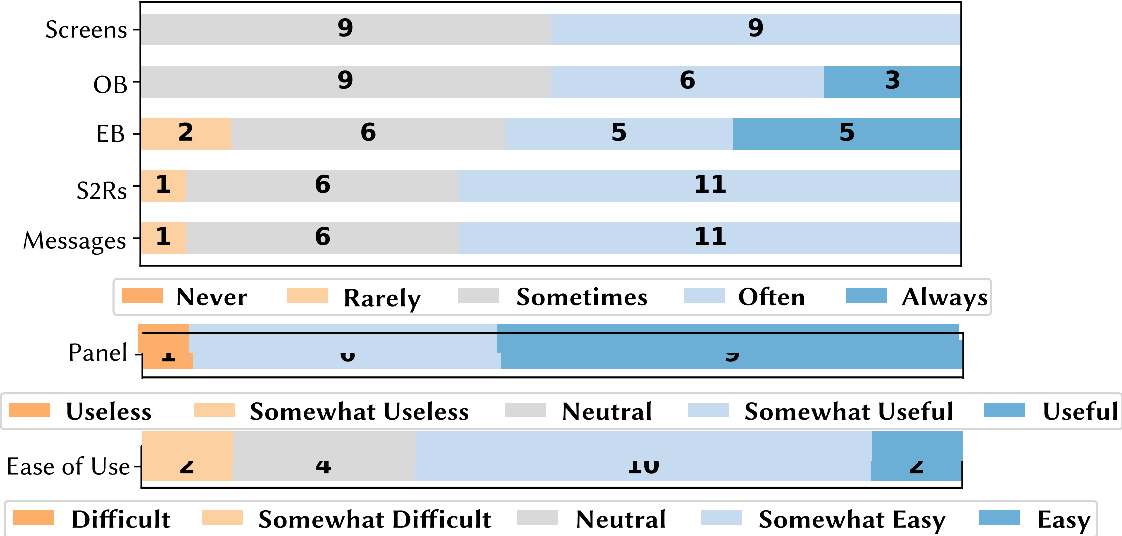

Figure 4.4: User experience results for BURT (Q1-Q5)

4.3.4.3 Measuring Bug Report Quality

We estimate the quality of the collected bug reports (via BURT and ITRAC) by assessing the correctness of the OB, EB, and S2Rs described in the reports, based on the quality model proposed by Chaparro *et al.* [78]. Three authors manually compared each collected report with the *ground truth scenarios* from Cooper *et al.*'s dataset [96], which included correct descriptions of the OB and EB and a minimum viable set of S2Rs. Using this methodology, we computed the following: (i) the number of incorrect OB/EB/S2R descriptions; and (ii) the number of missing S2Rs. To limit the effect of subjective assessments, two authors performed the bug report analysis independently and a third author reviewed the results, reaching consensus among all three in case of discrepancies. In order to determine how helpful BURT and ITRAC are for novices or more experienced reporters, we analyzed bug report quality across different levels of bug reporting experience.

4.4 Results and Analysis

We present and discuss the results of our evaluation for each RQ.

4.4.1 RQ_1: BURT's Perceived Usefulness

Fig. 4.4 summarizes the users' answers on: (i) their perceived usefulness of BURT's screen suggestions (row labeled *Screens*); (ii) BURT's ability to understand the user's OB, EB, and S2R descriptions (rows *OB*, *EB*, and *S2Rs*); (iii) how often they were able to understand BURT's messages and questions (row *Messages*); (iv) their perceived usefulness of BURT's panel of reported S2Rs (row *Panel*); and (v) BURT's overall ease of use (row *Ease of Use*).

App Screen Suggestions. Half of the 18 participants (9) agreed that BURT's app screen suggestions were *often* useful, and the other half (9) agreed they were *sometimes* useful. As for their rationales, one participant mentioned that the next S2R screen suggestions *"were useful because they shortened the time it took me to explain how to reproduce the bug"*. Other participants highlighted that the suggestions *"were helpful in making sure I was providing the exact steps I wanted to describe"*, or that BURT *"gave very good suggestions when it could figure out which screen had the bug based on the initial report"*. Some of the participants even hoped that BURT can provide suggestions more frequently. These results illustrate the usefulness of BURT's app screen suggestions.

Some participants noted, though, that *"the suggestions were a little inaccurate"*. We found that the inaccuracies stemmed from BURT not being able to recognize/match the user's OB description because of generic wording, without details (*e.g.*, *"the app crashed"*). Note that the BURT's S2R suggestions are not activated if the OB description is not matched to an app screen, which affected the reporters experience. Also, the participants recommended that it would be useful to have suggestions of *"bug triggering screenshots"*, as currently, BURT's screen captures may not show the bug that the user wants to report. The participants also found some suggestions confusing because the screen captures for the S2Rs highlight *"non-existent buttons"*. This stems from BURT's systematic GUI exploration technique, which can execute events on GUI components such as, layouts or views, which are often not visible to the user.

OB, EB, and S2R Understanding. The reporters have a positive overall impression

on how often BURT understood their OB, EB, and S2R descriptions. Specifically, BURT was able to *often* or *always* (*sometimes*) understand the OB/EB/S2R descriptions of 9/10/11 (9/6/6) participants (out of 18). Only two/one participant(s) felt that BURT *rarely* recognized their EB/S2Rs.

Our analysis of the open-ended answers also reveals that some participants were generally satisfied with BURT in terms of bug description understanding. This can be seen in comments such as *"I'm quite satisfied with the recognition rate [for the S2Rs], even better than talking to a real agent"*, *"It always understands my description of the OB/EB when I tried to use keywords from apps"*, *"it was kind of easy for burt to understand my (EB) description"*, and *"It can understand me to describe the error behavior"*. However, several participants had a less positive perception of BURT's bug description understanding stating that it is *"difficult to match* BURT*'s language"*, they *"need to follow specific pattern"* so that BURT is able to understand, and they *"usually had to paraphrase"* their descriptions. These comments stem from our design decision to limit the language that users could use to describe the OB, EB, and S2Rs, and inaccuracies in bug description matching. However, we observed, based on the reporters' comments and their conversations with BURT, that the participants learned how to describe the OB/EB/S2Rs using BURT's preferred formats after reporting the first bug. Still, the reporters' main recommendation was to improve BURT's ability to recognize additional vocabulary and ways of phrasing the OB/EB/S2Rs.

BURT's Messages and Questions. Eleven (of 18) participants *often* understood BURT's messages and questions, while six participants understood them *sometimes*. Only in one case, the reporter *rarely* understood the messages/questions.

The analysis of their rationales reveals that generally BURT's messages/questions were *"very easy to understand"*. One participant wrote that BURT's *"wording was always clear and I could always tell what* BURT *was asking for"*, also echoed by multiple participants. Some participants recommended to improve the messages and questions, as sometimes they were unclear and too similar to each other. For example, for BURT's question *"Was*

Table 4.3: Quality assessment results for bug reports (BRs) collected by BURT and ITRAC

App-Bug ID	# of BRs		Avg. # of S2Rs		Avg. # (%) of incorrect S2Rs		Avg. # (%) of missing S2Rs		# of BRs with incorrect OB		# of BRs with incorrect EB	
	ITRAC	BURT	ITRAC	BURT	ITRAC	BURT	ITRAC	BURT	ITRAC	BURT	ITRAC	BURT
APOD-CC3	5	5	4.6	7.4	0.6 (16.7%)	0.6 (9.7%)	6.4 (58.2%)	3.4 (30.9%)	0	1	0	3
APOD-RB	4	4	3.3	4.8	0.0 (0.0%)	1.8 (30.6%)	1.0 (20.0%)	1.0 (20.0%)	1	1	1	0
DROID-CC5	6	6	4	4.3	0.3 (11.1%)	0.5 (10.0%)	1.3 (19.0%)	0.7 (9.5%)	1	2	1	1
DROID-CC6	6	6	4.5	8.5	1.3 (44.4%)	1.2 (13.7%)	5.0 (41.7%)	2.0 (16.7%)	0	3	1	1
GNU-CC9	5	5	6.4	10.2	0.8 (26.7%)	0.2 (2.5%)	4.8 (36.9%)	3.2 (24.6%)	1	0	1	0
GNU-RC	3	3	4.7	4.3	0.0 (0.0%)	0.3 (6.7%)	0.0 (0.0%)	0.0 (0.0%)	0	1	0	0
GROW-CC5	4	4	4.8	7.5	1.3 (28.1%)	0.0 (0.0%)	4.5 (45.0%)	3.5 (35.0%)	0	0	0	0
GROW-RC	4	4	5.8	7.5	1.0 (30.0%)	0.5 (7.1%)	1.8 (21.9%)	1.5 (18.8%)	1	3	1	0
TIME-CC1	5	5	7.8	10.4	1.0 (24.0%)	0.2 (2.9%)	6.6 (41.3%)	6.2 (38.8%)	0	1	0	0
TIME-CC4	4	4	4.3	8	1.0 (24.4%)	0.3 (5.0%)	3.0 (33.3%)	1.3 (13.9%)	1	2	2	1
TOK-CC2	4	4	4.8	10.3	0.3 (8.3%)	0.8 (6.8%)	2.5 (25.0%)	0.5 (5.0%)	2	2	0	0
TOK-CC7	4	4	5	5.8	0.5 (16.7%)	0.3 (3.6%)	1.5 (25.0%)	0.8 (12.5%)	1	0	1	0
Overall	**54**	**54**	**5**	**7.5**	**0.7 (20.4%)**	**0.6 (8.3%)**	**3.4 (32.0%)**	**2.1 (19.4%)**	**8**	**16**	**8**	**6**

this the last S2R that you performed?", the participants suggested to clarify which last S2R BURT was referring to.

The Panel of Reported S2Rs. BURT's panel of reported S2Rs was deemed to be *useful* (*somewhat useful*) by 9 (6) participants. Only one participant found that the panel was *somewhat useless*. The participants commented that the panel was *"Very useful for visualizing a bug report"*, that *"It was good to see what was getting logged"*, and that it was useful *"as a way for me to review that the reproduction steps I entered are complete"*.

Summary of findings for RQ$_1$: Overall, reporters found BURT's screen suggestions and S2R panel useful. They also had a positive impression of BURT's OB/EB/S2Rs understanding and messages. Improvements are required for BURT to support additional wording of bug report elements and more accurate suggestions.

4.4.2 RQ$_2$: BURT's Perceived Ease of Use

Twelve reporters indicated BURT was either *easy* or *somewhat easy* to use. Four reporters were neutral, while two reporters expressed it was *somewhat difficult* to use (see *Ease of use* in Fig. 4.4).

We analyzed the reporter responses regarding which of BURT's features they found easy/difficult to use. In general, the participants expressed that BURT's GUI *"is really*

helpful", *"concise"*, and *"easy to use and understand"*. Multiple reporters indicated that selecting BURT's app screen suggestions was easy to use and some of them were very enthusiastic about them. One reporter mentioned that *"I liked the screenshots a lot, very easy to report the process to reproduce a bug"*. Other reporters expressed that *"The suggestions & confirmations were very easy to use. When it had the right idea, confirming it was just a matter of clicking a button"*, and that BURT *"guides the user to provide a "step-by-step" view"*. The panel of reported steps was easy *"to explore"* and it was easy to *"remove events"* from it.

The main reason behind usage difficulties was the limited vocabulary that BURT understands, also observed before for RQ_1. The reporters recommended to let the users upload their own screen captures when BURT is unable to attach screens to the user's bug descriptions, and the ability to delete/modify *any* step.

Finally, for both RQ_1 & RQ_2, we found no notable differences in BURT's perceived usefulness and ease of use between different levels of user's bug reporting experience.

4.4.3 RQ_3: BURT's Intrinsic Accuracy

We analyzed the 54 conversations that reporters had with BURT to determine how often BURT was able to correctly (1) match OB/EB/S2R descriptions to the execution model, and (2) suggest relevant OB/S2R app screens to the reporters.

OB Reporting. We found that in 3 of 54 conversations (5.5%), BURT was able to match the reporter's OB description to the correct screen that showed or triggered the bug, as confirmed by the reporter during the conversation. In 35 of 54 conversations (64.8%), BURT matched the OB description to multiple app screens. In those cases, BURT suggested the top-5 matched screens so that the reporter selected the one s/he was referring to. In 29 of these 35 reports (80%), the reporter selected one of the suggested screens, while in the remaining 6, the suggested screens were irrelevant. For the remaining 16 of the 54 conversations (29.6%), BURT was not able to match the OB description with any app screen because of incorrect OB wording from the user and inaccuracies in BURT's message

parser and processing. Overall, BURT was able to correctly match their OB descriptions in 32 of 54 of the conversations (59.3%).

EB Reporting. As described in Sect. 4.2.4.2, BURT can only match the reporter's EB description when there is a matched/selected OB screen. Otherwise, BURT collects the EB description from the user as is. In the 32 cases when BURT can verify EB quality, BURT was able to match the EB against the OB screen in 17 cases (53.1%) without having to ask the reporter for confirmation. In 6 of the 32 cases (18.8%), the users confirmed the matched OB screen when BURT asked them about that. In the remaining 9 cases (28.1%), BURT was not able to parse the provided EB description.

S2R Reporting. BURT matched a written S2R with a step from the execution model 205 times in total across the 54 conversations (3.8 times per conversation on avg.). In 157 of these cases (76.6%), BURT was able to match S2Rs correctly. BURT predicted and suggested the next S2Rs in 146 cases (4.6 times per conversation on avg.) for the 32 conversations where there was a matched/selected OB screen. We found that the reporters selected 1.6 of the 3.9 suggested S2Rs (on avg.) in 91 cases (62.3%). In 13 of the 32 conversations, the reporter always selected S2Rs from the suggested list, meaning at least one suggestion was correct. In all the 54 conversations, BURT asked the user to rephrase their S2Rs 176 times (3.9 times per conversation on avg.). We found that in at least 59 of these cases (33.5%), the user made a mistake or described the step incorrectly (*e.g.*, *"incorrect result"* or *"no more steps"*).

Summary of findings for RQ_3: The results support the users' ratings (RQ_1) on how often BURT's OB/S2R screen suggestion were useful and how often BURT was able to understand the user's OB/EB/S2R descriptions. The accuracy assessment revealed cases where BURT's struggles to parse and match the users' descriptions, however, BURT is able to continue with rephrasing prompts. The overall accuracy indicates that the techniques we used in building BURT's components are adequate. Improvements are planned for future work to improve BURT's accuracy.

4.4.4 RQ$_4$: Bug Report Quality

Table 4.3 summarizes the quality measures of the $54 \times 2 = 108$ bug reports, collected with ITRAC and BURT, for the 12 bugs in our dataset (each bug is reported in 3 to 6 reports).

S2R Quality. Overall, as shown in Table 4.3, BURT reports contain fewer incorrect S2Rs than ITRAC reports on avg. (8.3% vs. 20.4%) and fewer missing S2Rs (19.4% vs. 32%), compared to the ground-truth scenarios of the 12 bugs. We performed an analysis to verify whether there there statistically significant differences between BURT and ITRAC on the percentage of incorrect and missing S2Rs. We applied the Wilcoxon signed-rank test [120] and Cliff's delta (CD) [94] on the results, across the 12 bugs (at 95% confidence level), since we have paired ordinal measurements (for each bug) that do not necessarily follow normal distributions. We found that BURT's bug reports have fewer incorrect ($p = 0.0261$) and fewer missing steps ($p = 0.0025$) than ITRAC's reports, with a large effect size (CD = 0.5 and 0.527, respectively).

The main reasons for incorrect S2Rs are generic/unclear step wording (4 in BURT and 36 ITRAC reports), duplicate S2Rs (13 in BURT reports, zero in ITRAC reports), and extra S2Rs (10 in BURT and one in ITRAC reports). Examples of steps with unclear/generic wording include *"Add comment"* or *"I searched for tech"*, where the user either refers to high-level app features, which map to multiple steps that are not explicit, or does not specify which GUI components should be used and/or which action should be applied on them. Extra S2Rs are irrelevant reported steps (*e.g.*, "*I did nothing else*"). We identified two main reasons for duplicate S2Rs: (1) user mistakes; and (2) duplicate app screens suggested by BURT and selected by the users. The latter stems from the design of BURT's execution model that considers structural variations of the same screen as different screens (see Sec. 4.2.4.1). An example is when the users employ different keyboard layouts (*e.g.*, numeric vs. alphanumeric) to enter input values on the same screen.

OB/EB Quality. More BURT reports have an incorrect OB description compared to ITRAC reports (16 vs. 8 out of 54 reports), while a comparable number of BURT and ITRAC

reports have an incorrect EB description (8 vs. 6). We found that there is no statistically significant difference between the number of BURT and ITRAC bug reports with incorrect expected behavior ($p = 0.1586$), with a small effect size (CD = 0.222) in favor of BURT. Fewer ITRAC reports than BURT reports have an incorrect observed behavior ($p = 0.0352$), with a medium effect size (CD = 0.361).

The incorrect OB/EB descriptions (in 18 BURT reports and 10 ITRAC reports total) occurred either because the participants did not provide enough details about the bug (*e.g.*, *"the app crashed"*) or they described their inability to perform an action rather than describing the bug itself (*e.g.*, *"I can't add/delete a comment" vs. "Crash when trying to add/delete a comment"*).

For the 18 BURT reports, we found that, in 14 cases the users described the OB/EB incorrectly to begin with and BURT correctly prompted them to rephrase them. Nonetheless, they still reported an incorrect OB/EB. In four cases, BURT accepted the incorrect OB/EB, and in only three of the cases, BURT prompted incorrect OB/EB reporting after the user correctly described them. This is mainly due to BURT's current limitation on the OB/EB wording.

Summary of findings for RQ$_4$: Overall, BURT bug reports contain higher-quality S2Rs than ITRAC bug reports, and comparable EB descriptions. The results indicate that improvements to BURT are needed to better collect OB descriptions from the reporters.

Table 4.4: S2R quality by bug reporting experience

Reporting	**# of BRs**		**Avg. # of S2Rs**		**Avg. % of incorrect S2Rs**		**Avg. % of missing S2Rs**	
experience	ITRAC	BURT	ITRAC	BURT	ITRAC	BURT	ITRAC	BURT
Novice	15	15	3.5	6.7	33.6%	6.7%	45.6%	31.3%
Intermediate	24	27	5.2	7.5	20.1%	11.5%	35.5%	20.0%
Experienced	15	12	6.1	8.6	7.6%	3.2%	12.8%	3.2%
Overall	**54**	**54**	**5**	**7.5**	**20.4%**	**8.3%**	**32.0%**	**19.4%**

Novice vs. Experienced Bug Reporters. Our original expectation was that BURT would help novice reporters more than ITRAC, as the experienced reporters likely used template-based reporting systems before.

We compared the quality of the bug reports across different levels of user's bug reporting experience. While we did not observe notable differences in terms of OB/EB quality, we found differences in S2R quality, which we discuss. Table 4.4 shows the S2R quality results for three groups: novice bug reporters (with *no* prior reporting), intermediate reporters (who had reported 1-5 bugs), and experienced reporters (who had reported 6+ bugs).

Regarding incorrect S2Rs, experienced and intermediate reporters produced about twice as many incorrect S2Rs with ITRAC, compared to BURT (33.6% vs. 6.7%, and 20.1% vs. 11.5% on avg., respectively). At the same time, novices produced about five times more incorrect steps with ITRAC than with BURT (7.6% vs. 3.2% on avg.). This indicates that BURT helps novices most to avoid incorrect S2Rs.

Table 4.4 tells a different story for missing S2Rs. Novices and intermediate reporters missed ≈1.5 times fewer S2Rs with BURT, compared to ITRAC, while experienced reporters missed four times fewer S2Rs with BURT. Surprisingly, this indicates that BURT helps experienced reporters most to avoid missing steps.

We do not speculate on the reasons behind these observations, as more in-depth studies are needed for proper explanations.

4.5 Limitations and Threats to Validity

Before BURT is deployed for use, either systematic app exploration data or crowdsourced app usage data needs to be collected to construct the app execution model. The evaluation results indicate that BURT performs reasonably well with the data collected by CRASHSCOPE and only four people. However, we expect that additional data (more covered states and scenarios) would improve BURT's quality verification of reported elements and screen/step suggestions, enabling the reporting of different bug types, under a variety of reproduction scenarios. To confirm our expectations, additional studies are needed for future work.

BURT is evaluated in a lab setting where reporters were exposed to the bugs through videos, rather than letting them find the bugs while using the apps, as users would do in real life. As in prior studies [78, 163], we adopted this setting mainly to reduce participant effort and fatigue. To address the lack of knowledge about the apps/bugs, we instructed the users to get familiar with the apps by using them and with the bugs by reproducing them on the emulator before they reported the bugs. We addressed potential bug misunderstandings via 2/3-word annotations added to the videos.

A diverse group of reporters participated in the studies, who have different levels of bug reporting experience. Since we offered the reporters a monetary incentive for their participation and some of them are students from our institution(s), they may have been motivated to diligently provide high-quality bug reports, which may not necessarily be the case in a real-life scenario. However, we expect this factor to have a minimal impact on the results since (1) we used the same procedure to recruit both BURT and ITRAC users, and (2) the bug reporting experience in both reporter groups are almost the same (only two ITRAC users have a different experience).

Our evaluation did not consider how easy or difficult it is (for developers) to understand and reproduce the ITRAC and BURT bug reports. Instead, we focused on assessing bug report quality, as done by prior work [78]. Assessing bug report understanding and reproduction is in our plans for future work. Additionally, we did not account for the complexity of the bugs in our dataset. However, we selected bugs of diverse types and distributions of the S2Rs. Our future work will investigate how bug complexity affects BURT.

Finally, given the relatively expensive nature of our evaluation, we limited it to 12 bugs from six apps, reported by 36 participants, which affects the external validity of our conclusions. A larger evaluation, possibly performed on a larger sample of apps, bugs, and participants is in our plans for future work.

4.6 Related work

We discuss BURT's advancements in relation to prior work.

Issue/Bug Reporting Systems. A variety of systems currently enable end-users and developers to manually report software bugs, namely, issue/bug trackers (*e.g.*, GitHub Issues [20] or JIRA [24]), built-in bug reporting interfaces in desktop and web apps (*e.g.*, Google Chrome [29]), in-app bug reporting frameworks (*e.g.*, BugSee [33]), app stores [34, 15], and Q&A platforms [68]. These systems typically consist of web/GUI forms (with text-based templates) that allow reporters to provide bug descriptions, indicate bug/system metadata, and attach relevant files. Some of these systems collect technical information (*e.g.*, configuration parameters) and offer screen recording that enable graphical bug reporting.

While existing systems provide features that facilitate bug reporting, they offer limited guidance to bug reporters, lack quality verification of bug report information, and do not provide concrete feedback on whether this information is correct and complete. These are some of the main reasons for having low-quality bug reports, which have important repercussions for developers [238, 237].

Researchers have explored improving bug reporting interfaces, as we do in this work. Moran *et al.* [163] proposed FUSION, a web-based system that allows the user to report the S2Rs graphically by selecting (via dropdown lists) images of the GUI components and actions (taps, swipes, *etc.*) that can be applied on them. More recently, Fazzini *et al.* proposed EBUG [104], a mobile app bug reporting system similar to FUSION that suggests potential future S2Rs to the reporter while they are writing them. Record-and-replay tools [161, 111, 123, 174] offer the ability to record user actions during app usage (*e.g.*, when a bug is found) and replay them later.

BURT offers two main advancements over prior techniques like FUSION. First, BURT was designed to support end-users with little or no bug reporting experience. For example, FUSION was not created to specifically cater to end-users, as inexperienced users found

it *more difficult* to use as compared to alternatives [163]. Second, whereas past systems helped to provide structured mechanisms to facilitate the reporting process (*e.g.*, through drop-down selectors) they do not offer *interactive* assistance when reporting a bug. BURT offers such interactivity through its automated suggestions, real-time quality assessment, and prompts for information clarification.

Bug Report Quality Analysis. Surveys and interviews with developers and end-users [237, 143, 187] have identified the observed software behavior (OB), the expected behavior (EB), and the steps to reproduce (S2Rs) the bugs as essential bug report elements for developers during bug triage and resolution. Unfortunately, such elements are often missing, unclear, or ambiguous, as indicated by numerous studies and developers [101, 236, 74, 113, 237, 105, 136, 3], which have a negative impact on bug report management tasks.

In consequence, researchers have proposed techniques to better capture and manage high-quality information in bug reports. Prior work [67, 87, 192, 231] proposed ways to automatically identify different essential elements in bug reports (*e.g.*, S2Rs [229, 155]), analyze their quality, and give feedback to reporters about potential issues in them. In particular, Zimmermann *et al.* [237] proposed an approach to predict the quality level of a bug report based on factors such as readability or presence of keywords. Hooimeijer *et al.* [121] measured quality properties of bug reports (*e.g.*, readability) to predict when a report would be triaged. Zanetti *et al.* [221] proposed an approach based on collaborative information to identify invalid, duplicate, or incomplete bug reports. Imran *et al.* [126] proposed an approach to suggest follow-up questions for incomplete reports. Song *et al.* [192, 87] proposed a technique to detect when the OB, EB, and S2Rs are absent in submitted bug reports. Chaparro *et al.* [78] evaluated the quality of the S2Rs in bug reports through the EULER tool, which integrates dynamic app analysis, NLP and graph-based approaches.

Our work builds upon prior research for the automated quality verification of bug descriptions by developing quality checks for new types of bug elements (*i.e.*, OB/EB)

and by designing dialogue flows capable of guiding the user during the bug reporting process.

4.7 Conclusions

BURT is a task-oriented chatbot for interactive Android app bug reporting. Unlike existing bug reporting systems, BURT can guide end-users in reporting essential bug report elements (*i.e.*, OB, EB, and S2Rs), provide instant feedback about problems with this information, and produce graphical suggestions of the elements that are likely to be reported.

Eighteen end-users reported 12 bugs from six Android apps and reported that, overall, BURT's guidance and automated suggestions/clarifications are accurate, useful, and easy to use. The resulting bug reports are higher-quality than reports created via ITRAC, a template-based bug reporting system, by other 18 reporters. Specifically, BURT reports contain fewer incorrect and missing reproduction steps compared to ITRAC reports. We observed that BURT is most helpful to novice reporters for avoiding incorrect S2Rs. Surprisingly, BURT seems to be most useful to experienced reporters for avoiding missing reproduction steps.

The reporters provided feedback for refining the supported dialog, by including support additional wordings to describe the OB, EB, and S2Rs. The studies also revealed areas of improvement for BURT with respect to the verification of the reported elements.

4.8 Data-Availability Statement

We provide an online replication package [35] that contains a complete implementation of BURT, BURT's app execution data, code and data about BURT's evaluation, and documentation that enables the verification and validation of our work and future research on bug reporting systems.

Chapter 5

Recommending Bug Assignment Approaches for Individual Bug Reports

Multiple approaches have been proposed to automatically recommend potential developers who can address bug reports. These approaches are typically designed to work for any bug report submitted to any software project. However, we conjecture that these approaches may not work equally well for all the reports in a project. We conducted an empirical study to validate this conjecture, using three bug assignment approaches applied on 2,249 bug reports from two open source systems. We found empirical evidence that validates our conjecture, which led us to explore the idea of identifying and applying the best-performing approach for each bug report to obtain more accurate developer recommendations. We conducted an additional study to assess the feasibility of this idea using machine learning. While we found a wide margin of accuracy improvement for this approach, it is far from achieving the maximum possible improvement and performs comparably to baseline approaches. We discuss potential reasons for these results and conjecture that the assignment approaches may not capture important information about the bug assignment process that developers perform in practice. The results warrant future

research in understanding how developers assign bug reports and improving automated bug report assignment. This work has been published on arXiv [193].

5.1 Introduction

Many software projects receive hundreds of bug reports on a daily basis that need to be triaged and solved timely [239, 57, 197]. An important step in the bug triage process is to assign the bug reports to the developer who has the proper expertise to solve the reported bugs. Given the high volume of incoming reports, this process is often time-consuming [239, 57, 197]. Hence, researchers have proposed approaches to automatically recommend potential developers for a bug report (*e.g.*, [54, 58, 122, 199, 204, 214]). These approaches implement a variety of techniques, *e.g.*, machine learning (ML) [70] or information retrieval (IR) [159], and leverage bug report information such as past reports or bug fixes, or the developers who processed these artifacts.

Existing bug assignment approaches use different artifact information and ways to encode such information for automatically recommending potential developers. More importantly, these approaches are typically designed to be *general-purpose*, *i.e.*, they are designed to work for all the reports submitted to a software project. However, we conjecture that these approaches may not work equally well for all the reports in a software project, given their internal recommendation mechanisms, which may not account for particular nuances that each bug report has (*e.g.*, the bug type and specific information reported for such a type).

In this work, we first report an empirical study (*a.k.a.* Study 1) that aims to validate such conjecture. We executed three bug report assignment techniques,

that use distinct techniques (*e.g.*, ML or IR) and information from various software artifacts, on 4,929 bug reports from three open-source systems. We compared how often the three approaches perform best in recommending the ground-truth developers for the reports. We found that no single approach performed best for the majority of the bug

reports in each system. Instead, each of the three approaches performs better for some reports and worse for others, which means that these approaches do not work equally well for all the bug reports in a project.

The results motivated us to investigate the idea of automatically identifying the best approach for each individual bug report and running the predicted approach to produce more accurate developer recommendations. In this way, we would combine the advantages of the three approaches to produce better recommendations. To the best of our knowledge, this is the first work that investigates this idea for automated bug assignment (more details are given in Section 5.5).

We report an additional study (*a.k.a.* Study 2) to assess the feasibility of this idea by using machine learning (ML). We experimented with multiple ML models to recommend the best-performing bug assignment approach for each bug report. These models are trained and evaluated using the same bug report dataset from Study 1 and 23 features that encode possible relationships between the reports and the approaches. Once the models detect the best-performing approach for a report, this is applied to the report to obtain a list of potential developers who can address the reported bug. We call this approach Mix. The results show that, while there is a wide margin of accuracy improvement by perfectly recommending the best approach for individual reports, Mix is far from achieving the maximum possible improvement and performs comparably to using the individual assignment approaches on all the reports. We discuss potential reasons for these results and conjecture that existing approaches may not capture important information about the bug assignment process that developers perform in practice. The results warrant future research in understanding how developers assign bug reports in practice and improving automated bug report assignment.

5.2 Study 1: Bug Assignment on Individual Bug Reports

We conducted an empirical study to validate the conjecture that existing bug report assignment techniques do not work equally well for all the bug reports in a software project. Specifically, the study aimed to answer the following research question:

***RQ*$_1$**: *How do bug report assignment approaches perform and compare when applied on individual bug reports?*

5.2.1 Dataset

We collected bug report data for two open source projects, namely Angular.js (Angular) and WordPress Android (WordPress), which have been used in prior work [184, 87]. Angular is a web framework and WordPress is an Android app for website creation. These projects were selected because they are active, large (90kLOC+), support different domains, and involve a large number of developers (171 and 40 for Angular and WordPress, respectively).

We used the GitHub API to download all of the project issues submitted until Oct. 2020. For each issue, we collected the ID, the title/summary and description, the status, the date-time of submission, the labels, and the person who was assigned to solve the issue. We used the issue labels to identify the bug reports: those issues tagged with the label "bug" by the project maintainers. We only used those reports that were closed and had one or more commits that fixed the bug (read below). We collected 905 and 1,344 reports for Angular and WordPress, respectively.

As in prior work [184, 122], we built the set of developers who fixed the bug for each bug report (*i.e.*, the ground truth). We consider the developer assigned to solve the bug on the issue tracker as well as the developers who pushed the code changes to solve the bugs as the ground-truth developer set for each report. To build the developer sets, we identified the associated commits with each bug report by using regular expressions on the commit and bug report comments. From the identified commits, we extracted the

developers who authored or committed the code changes. The avg. number of developers that fixed the bug in the collected bug reports is 1.4 (Angular) and 1.1 (WordPress). We applied tokenization, stop word removal, and stemming on all the textual data (*e.g.*, bug reports).

5.2.2 Bug Assignment Approaches

We used different types of bug assignment approaches that leverage information from various artifacts to recommend potential developers. We used an ML-based approach (L2R [204]), an IR-based approach (Lucene), and another approach that leverages how frequent the developers solve bugs (Freq). These approaches recommend a ranked list of developers for a particular bug report (*a.k.a.* the query, formed by concatenating its title/summary and description). Developers ranked higher in the list are considered to have more expertise in addressing the report.

L2R is a state-of-the-art approach [204] (based on the Ranking SVM model) that learns to rank tuples composed of a bug report and developer, based on 16 features that represent the similarity/relevance between the report and the developer. For example, a set of features leverage the textual similarity between the bug report and the code files that a developer modified (via VSM [185] and BM25 [180]). Another set of features leverage the textual similarity between potentially-buggy code files (as given by a bug localization approach, based on the report) and the code files modified by the developer. Given that the original approach was not made publicly available, we implemented L2R. In our implementation, we use a Lucene-based bug localizer [77] as we were unable to obtain the original localizer [218].

Lucene [115] is an IR-based approach that implements a variation of the classical VSM [185] to compute the textual similarity between documents. We implemented a Lucene-based approach where the documents are bug reports (similarly to prior duplicate bug report detection work [181]). This approach finds the most similar past bug reports to a particular report (the query), retrieves the developers that addressed the reports,

and ranks them in the order in which the reports are ranked (from high to low textual similarity).

Freq is an approach (inspired by prior work [208]) that ranks the project developers based on the number of bug reports they have addressed. Developers that fixed a larger number of reports are ranked higher by this approach.

5.2.3 Metrics and Methodology

We measured the performance of the three approaches using standard metrics used in prior bug assignment studies [169, 204, 184].

HIT@k (H@k) is the percentage of bug reports (queries) for which at least one of the expected developers (*i.e.*, from the ground truth) is found in the top-k recommended developers. We report H@k for $k = 1 - 5$, as in prior work [169, 204]. **MRR** is the mean of the queries' reciprocal ranks. The reciprocal rank for a query is $1/rank$, where ***rank*** is the position of the first expected developer found in the ranked list of developers. **MAP** is the mean average precision (AP) over all queries. The AP for a query is the average of the precision values achieved at all the cutting points k of the ranked list of developers (precision@k). Precision@k is the proportion of the top-k recommended developers that are correct according to the ground truth. A higher H@k, MRR, and MAP indicate higher bug assignment performance.

To train/evaluate L2R (as done in [204]), we sort the bug reports chronologically by submission date-time and we split them up into 10 ten folds. The first x folds are used for training (with $x = 1...9$), and fold $x+1$ is used for model testing. L2R's overall performance is computed with the metrics defined above, applied on the set of queries from all the folds except the first one. The number of queries we used for measuring L2R's performance is 803 (Angular) + 1,208 (WordPress). We measured Lucene's and Freq's performance on the same query set to have a fair comparison among all the approaches.

The approach that achieved the lowest *rank* was considered the best approach for each individual report/query. We computed the % of queries for which each approach performs

best, including the cases when the approaches achieved the same (lowest) *rank*.

Table 5.1: Bug assignment performance for each system

Approach	MRR	MAP	H@1	H@2	H@3	H@4	H@5
Angular							
Freq	50.7%	43.6%	29.5%	49.6%	64.3%	75.5%	82.4%
Lucene	40.8%	36.0%	22.0%	37.6%	46.6%	56.4%	65.3%
L2R	52.0%	44.7%	31.4%	53.0%	65.3%	74.9%	80.5%
WordPress							
Freq	49.3%	48.8%	28.5%	45.7%	60.9%	74.4%	80.7%
Lucene	56.9%	56.2%	38.2%	56.8%	68.6%	77.6%	82.7%
L2R	57.0%	56.2%	38.7%	56.0%	69.4%	79.6%	84.1%

Table 5.2: % of reports for which the approaches perform best

System	L2R	LC	FR	L2R/LC	L2R/FR	LC/FR	All	Total
Angular	21.2%	16.7%	17.5%	4.7%	26.3%	4.1%	9.6%	813
WordPress	19.5%	20.9%	13.4%	11.6%	9.9%	4.1%	20.6%	1,208

LC: Lucene, FR: Freq, All: L2R/LC/FR

5.2.4 Results

Table 5.1 shows the bug assignment performance achieved by each approach. The approaches perform differently for each system. For Angular, Freq and L2R achieve a similar performance, while Lucene performs significantly lower. L2R's MRR and MAP are higher than those for Freq mainly because of the higher H@1-3. In terms H@4 and H@5, Freq performs better than L2R. For WordPress, Lucene and L2R outperform Freq on all metrics. Lucene and L2R perform similarly, L2R achieving slightly better MRR/ H@k (except H@2).

Table 5.2 shows the distribution of bug reports for which the three approaches performed best. A different distribution is observed for each system and the distributions are not skewed toward one of the approaches. The proportion of reports for which a single approach performs best ranges from 13.4% to 21.2%. For some queries, there are multiple best-performing approaches. For example, for Angular, both L2R and Freq perform best for 26.3% of the reports, while for WordPress, all three approaches perform best for 20.6%

of the reports. For each bug report, there is at least one approach that gives a more accurate developer recommendation than the other approaches. This means that no single approach performs equally well for all the reports, thus verifying our conjecture.

5.3 Study 2: Recommending the Best Performing Approach

The results from Study 1 motivated us to explore the idea of automatically recommending the best-performing approach for each bug report. In this way, we would combine the advantages of the three approaches to produce better developer recommendations.

We conducted an study to assess the feasibility this idea using machine learning (ML) and 23 features that aim to capture relationships between the bug reports and the bug assignment approaches. ML models recommend the best approach (Step 1), which is applied on the bug report to obtain a ranked list of developers that can address the report (Step 2). We coin this 2-step approach as Mix.

This study aimed to answer the following research question:

***RQ*$_2$:** *What is the bug assignment performance of* Mix *compared to that of baseline bug assignment approaches?*

5.3.1 Model Features

We used 23 features that aim to capture possible relationships between the bug reports and the bug assignment approaches. We selected 18 features from prior work on query quality assessment [160] since these can be used to measure textual and statistical properties of the bug report text and the textual corpora of past bug reports and past code changes made by the developers, information that is leveraged by L2R and Lucene. We defined additional five (5) features to capture the bug fixing distribution of developers in a project, information that is leveraged by Freq. We summarize the features.

Specificity features (11/23) measure how specific the bug report text is compared to the corpus of past bug reports (*a.k.a.* documents) [160] to differentiate relevant and

non-relevant documents. One specificity feature is the standard deviation of the inverse document frequency of the bug report terms (stdev. IDF). If the stdev. IDF for a bug report is low, L2R's and Lucene's recommendations may be impacted because the bug report may not have discriminatory information to identify the relevant past reports.

Similarity features (6/23) measure the degree of textual similarity between the query and the corpus of past bug reports or code files modified by the developers [160], information used by Lucene and L2R, respectively. Higher similarity may indicate the existence of many relevant documents to the report/query [160], potentially leading to better developer recommendations. One feature is the avg. Collection Query Similarity: the linear combination between the frequency of a report term in the collection of past code files and its IDF. This is computed by averaging over all query terms.

We used one (1/23) *coherency* feature that measures the average similarity between pairs of past bug reports that contain a bug report term [160]. This feature measures how focused a bug report is on a particular topic, expressed by its vocabulary. This information is used by both Lucene and L2R.

The remaining five (5/23) features measure the bug fixing activity of the developers. For example, some of these features measure the average/median/maximum # of reports solved by the developer, and the # of active developers (information leveraged by Freq).

5.3.2 Models and Methodology

We used well-known classifiers/models, used in prior bug assignment research [70], namely Random Forest, Naive Bayes, Decision Trees, and Logistic Regression, to automatically assign a bug report to one of three classes (*i.e.*, the approaches).

To train/evaluate the classifiers, we built ground-truth data based on the results presented in Table 5.2. For each bug report, we selected the approach with the lowest *rank* as the ground-truth approach for the report. In the case of *rank* ties, we opted for randomly selecting Freq, Lucene, or L2R as the ground-truth approach, to avoid data imbalance in our classes by selecting a single approach (as we tried to do in pilot experiments), which

would potentially bias the classifiers. To address the potential effects of the randomness, we repeat the experimental setting described in this subsection five times (*i.e.*, having five ground truth datasets).

To train the classifiers, we first sort the bug reports by submission date-time for each system. Next, we take the first 70% for training/validation, and the remaining 30% for testing. We use 5-fold cross validation to select the best parameters of each classifier using a chronological splitting where the first x folds (with $x = 1...4$) are used for training and the $x + 1$ fold is used for validation.

We found the best parameters for each classifier as those that led to the largest weighted avg. *F1-score* over the three classes on the validation sets. We measured the classification performance of each classifier (with the best parameters) on the test set using weighted avg. *F1-score*, *precision*, and *recall*. We average these values over the five runs for each classifier to obtain an overall performance

We experimented with the four classifiers by running them in Mix's 1st step. Then, in the 2nd step, the predicted approach (given by the classifier) for each report was executed to obtain a list of developers. Mix's performance is measured using the metrics from Sect. 5.2.3 on the test set when using each classifier. To compute the overall Mix's performance, we averaged these metrics over the five runs. We report Mix's best overall performance by selecting the classifier that leads to the lowest MRR, as this metric might better capture the scenario where the bug triager scans through the developer list until deciding the developer for the bug report.

5.3.3 Results and Discussion

Table 5.3 shows the bug assignment performance of Mix (on the test set) compared to that of the baseline approaches (L2R, Lucene, & Freq). The *max* row in Table 5.3 shows the maximum performance that Mix can achieve if it used a classifier that perfectly predicts the best approach for every bug report. The *max* results indicate that there is great potential for Mix to improve bug assignment performance, yet the Mix results show that

Table 5.3: Bug assignment performance on the test set

Approach	MRR	MAP	H@1	H@2	H@3	H@4	H@5
Angular							
L2R	56.9%	49.9%	32.8%	61.9%	79.9%	87.7%	90.6%
Lucene	43.8%	39.0%	21.5%	40.5%	54.9%	67.5%	78.1%
Freq	57.4%	50.2%	34.9%	57.6%	76.5%	89.5%	93.7%
MIX	56.5%	48.9%	32.8%	59.7%	77.0%	88.4%	92.2%
Max	72.2%	62.6%	56.6%	75.8%	86.5%	91.4%	94.7%
WordPress							
L2R	46.0%	45.3%	29.0%	41.7%	54.0%	62.4%	68.7%
Lucene	47.8%	47.2%	30.7%	44.3%	53.3%	61.4%	68.1%
Freq	40.4%	40.1%	23.4%	30.8%	41.9%	53.3%	63.2%
MIX	46.2%	45.7%	29.1%	41.1%	52.7%	61.8%	68.4%
Max	63.7%	62.8%	48.3%	61.8%	74.1%	81.3%	87.6%

it is still far from achieving the maximum performance, as are the baseline approaches.

For Angular, MIX (using Naive Bayes) achieves comparable performance to L2R/Freq's performance and superior performance to Lucene's. Similarly, for WordPress, MIX (using Decision Trees) achieves comparable performance to L2R/Lucene's performance and superior performance to Freq's. The results show that MIX does not perform more accurately than all the baseline approaches.

We investigated this phenomenon by analyzing the overall classification performance of the classifiers on the test set. We found that Naive Bayes achieves 30.6%, 36.7%, and 31.3% weighted avg. precision, recall, and F1, respectively, while Decision Trees achieve 32.9%, 34.4%, and 29.3% weighted avg. precision, recall, and F1, respectively. These are average values over the five experimental runs. It is important to note that the % of reports for each class (approach) range from 23% (min) to 40% (max) across the five runs, which rules out potential problems related to data imbalance. One likely reason for the classifier results is that the classifier features are not capturing enough information to distinguish among the approaches. One potential solution to explore in our future work is to use post-retrieval features [160] rather than only pre-retrieval features (as the ones we currently used).

However, since the features we used for classification encode information that L2R,

Lucene, and Freq leverage to make developer recommendations, one potential implication of the results is that these (and other) approaches may also lack additional information and factors that maintainers use in practice to assign bug reports to developers, that are not necessarily found in software repository data (in bug reports, commits, source code, *etc.*) [60]. For example, maintainers may assign reports to developers based on developer availability at a given moment, developer's technical background or experience (*e.g.*, in using specific technology), and different social structures and dynamics that can be found in open source projects [71, 239]. In other words, maintainers may perform bug assignment not entirely based on developer profiles defined by the past bug reports, code changes that developers have addressed, and other information, as existing bug assignment techniques attempt to model. We advocate for additional research to understand the way bug reports are assigned to developers in practice.

5.4 Threats to Validity

As in prior work [184], we used project repository data to create bug assignment ground-truth data. We minimized potential errors in bug reports and developer sets by (1) selecting issues labeled as "bugs" by the original maintainers, and (2) by manually curating the extracted data. For example, we manually merged GitHub accounts that referred to the same developer. We selected a diverse set of approaches according to the techniques they implemented and the information they used. The results may not generalize beyond the selected two open source projects. Expanding the studies with additional approaches/systems is in our plans for future work.

5.5 Related work

A variety of approaches have been proposed to automatically assign bug reports to developers [54, 58, 59, 63, 70, 73, 122, 127, 132, 134, 138, 145, 154, 158, 169, 183, 184, 186, 188, 189, 198, 199, 200, 204, 215, 214, 222, 224, 225]. These techniques leverage multiple

sources of information such as past bug reports, source code, and bug tossing information, and are typically designed to be applied to all bug reports in a project. In contrast, our work aims to recommend specific approaches to individual bug reports to improve developer recommendations. To the best of our knowledge, this work is the first to investigate this idea, yet similar work has been done to support other software engineering tasks [160, 166, 92].

5.6 Conclusions and Future Work

We conducted an empirical study that applied three bug assignment approaches on 2k+ bug reports from two open source projects. We found that such approaches do not perform equally well when applied on all the bug reports from a software project. This finding motivated us to explore the idea of automatically recommending and applying the best-performing approach on individual bug reports via machine learning (ML). We experimented with four ML models that learn from 23 features and found that this composite approach is far from achieving the maximum possible performance, while achieving comparable performance to that of the baselines approaches. We found that the features utilized by the ML models may not capture enough information to distinguish between approaches. A possible implication of this result is that bug assignment approaches do not capture factors (*e.g.*, developer availability) found in the way bug reports are assigned to developers in practice. The results warrant future research in (1) defining effective features to better distinguish assignment approaches, (2) understanding how developers perform bug assignment in practice, and (3) incorporating additional information on automated bug assignment approaches to better recommend developers.

Chapter 6

Automated Localization of Buggy Mobile App UIs from Bug Descriptions

Bug report management is a costly software maintenance process comprised of several challenging tasks. For mobile apps, one task that is central to fixing reported bugs is identifying the UI screens and components that cause and/or show a reported issue (*Buggy UI Localization*). Given the UI-driven nature of mobile apps, bugs typically manifest through the UI, hence the identification of buggy screens and components is important to localizing the buggy behavior and eventually fixing it. However, this task is challenging as developers must reason about bug descriptions, which are often low-quality, and the visual or code representations of UI screens.

This work is the first to investigate the feasibility of automating the task of Buggy UI Localization through a comprehensive study that evaluates the capabilities of two textual and three multi-modal deep learning (DL) techniques. We evaluate such techniques at two levels of granularity, Buggy UI *Screen* and *Component* localization. We evaluated such techniques in both zero-shot and fine-tuned settings, using a manually curated dataset of 228 real-life bug descriptions and corresponding UI screens/components to automate

those 2 tasks. Our results illustrate the individual strengths of models that make use of different representations, wherein models that incorporate visual information perform better on screen retrieval tasks, and models that operate on textual screen information perform better on component retrieval tasks – highlighting the need for a localization approach that blends the benefits of both types of techniques. Furthermore, we study whether Buggy UI Localization can improve traditional bug localization in source code, and find that incorporating UIs leads to improvements of 4.77%-16.69% in Hits@10.

6.1 Introduction

Bug report management is an essential, yet costly process for software projects, in particular for mobile apps [239]. It demands high developer effort [239, 237, 58, 103, 72, 203] due in part to the potential for large volumes of reported bugs and the varying quality of submitted bug reports. These reports are the central artifact in bug management [237, 239, 103, 86], as they directly impact downstream tasks such as bug triaging, reproduction, localization, program repair, and even regression testing. Bug reports typically describe defects found during software development and usage, and are expected to include, at minimum, the app's observed (incorrect) behavior (**OB**), the expected behavior (**EB**), and the steps to reproduce the bug (**S2Rs**) [67, 143, 237, 86, 190, 191].

One critical task in bug report management for mobile apps is the identification of UI screens and components (*e.g.*, buttons or text fields) that cause or display the reported incorrect behavior of the app (*i.e.*, the OB), a task that we term *Buggy UI Localization*. This task is essential but can be difficult for developers, especially when many incoming bug reports need to be addressed that may fail to include important details or graphical information (*e.g.*, buggy app screenshots [100]). In addition, mobile app developers are typically subject to constraints such as rapidly evolving platform APIs [153] and the need to support fragmented device ecosystems [210], making this task even more challenging. Despite the growing body of work on automating bug report management

tasks [117, 53, 144, 219, 167, 93, 212, 140, 108, 79, 81, 233, 83, 231, 213, 230, 196], prior work has not yet explored how to assist developers in Buggy UI Localization.

Compared to other types of software, mobile apps are inherently UI-centric, and past studies have illustrated that a large majority of real bugs reported for mobile apps manifest through the UI [131]. As such, an important first step toward understanding, diagnosing, and resolving underlying bugs is localizing the buggy behavior to both a UI screen and UI component(s). In fact, recent work has illustrated how UI information can be used to improve traditional bug localization to code [156]. However, in that work, the UI information was *manually* collected, and other prior research has not thoroughly explored the potential of automating the task of Buggy UI Localization. Furthermore, this task is cognitively demanding, as developers must reason about the (often poor) descriptions of buggy behavior, and visual or code representations of UI screens [163].

In this context, it is clear that automating the process of localizing a bug that is described in the OB of a report to both UI *screens* and *components*, would greatly assist developers during bug triaging and resolution, and may also improve the effectiveness of a wide range of existing techniques for automated bug report management and software testing tasks [231, 230, 105, 195, 78, 194, 156]. For example, recent techniques that leverage the S2Rs to automatically reproduce the reported bugs [231, 230] or generate test cases [105] may benefit from a Buggy UI Localization approach that suggests buggy UI screens, as these could be used to generate assertions that verify if the reported bugs can indeed be replicated. Additionally, a Buggy UI Localization approach can help existing techniques verify the quality of OB descriptions more accurately [195, 78, 194] by checking how well the description maps to UI screens/components and thus give more accurate feedback to the reporter about (un)informative or (un)clear bug descriptions. A Buggy UI Localization approach may also be useful in enhancing textual bug reports with UI screenshots so that it can assist developers in comprehending and resolving app issues more effectively [170, 220, 52, 74, 237]. This can further benefit existing techniques that detect duplicate bug reports using visual data [207].

In this work, we present the first empirical study that investigates *the feasibility of automatically localizing bug descriptions to UI screens/components of mobile apps*. Similar to other types of bug localization, we formulate Buggy UI Localization as a retrieval task, in which a bug description is used as query input to a retrieval engine that searches the space of UI screens/components of an app and recommends a ranked list of candidates that most likely correspond to a given bug description. Specifically, the study focuses on two retrieval tasks: ***screen localization (SL)***, which involves retrieving UI screens, and ***component localization (CL)***, which aims to retrieve UI components from a given screen, relevant to the bug description.

The study investigates how the textual and visual information from UI screens/components can be leveraged for Buggy UI Localization, and hence, explores the effectiveness of pre-trained textual and multimodal deep learning (DL) techniques. Specifically, we examine five models: two text-based models (SentenceBert or SBert [179] and OpenAI's embedding model or OpenAI-txt-embed), and three multimodal models (Clip [175], Blip [146] and GPT-4). We have made necessary adjustments and conducted extensive experiments on these models as they are not tailored to solve our problem. While large language models (LLMs) like GPT-4 show promising "zero-shot learning" abilities—generating the correct output without seeing any example of the task, the significance of fine-tuning remains, particularly for models smaller than LLMs, to align with the nuances and requirements of the buggy UI localization task. Additionally, the lack of a benchmark for model evaluation led us to create two novel datasets: a *synthetic dataset*, and a *real dataset*. We fine-tuned three models (SBert, Clip, and Blip) using the *synthetic dataset* and evaluated the effectiveness of the models in zero-shot and fine-tuned settings using the real dataset. Regarding the two OpenAI models (GPT-4 and OpenAI-txt-embed), we leveraged their APIs directly due to their exceptional zero-shot capabilities, as OpenAI has not made fine-tuning options available for these models. The *synthetic dataset* includes 2.8M OB descriptions automatically created for 23.9k mobile app UI screens, via textual templates designed/implemented for a variety of mobile app

bug types. The *real dataset* includes 228 OB descriptions and associated UI screens/components, manually tagged in 87 real bug reports from the AndroR2 dataset [211]. The dataset also includes associated buggy UI screens/components that we manually labeled from a UI corpus created by employing GUI app exploration techniques [162], for 39 mobile apps.

The results of our study illustrate that GPT-4 and OpenAI-txt-embed from OpenAI perform best for SL and CL localization tasks, respectively. The best-performing approaches suggest correct buggy UI screens (GPT-4) and components (OpenAI-txt-embed) in the top-3 recommendations for 73% and 71.6% of the bug descriptions, respectively. We also found the models tend to perform better for higher-quality bug descriptions and easier-to-retrieve cases as judged by humans. The results show the feasibility and effectiveness of using existing DL, particularly multimodal large language models (LLMs), for Buggy UI Localization.

To illustrate the practical usefulness of automated Buggy UI Localization, we further studied how identified buggy UI screens from the best-performing SL model can improve traditional bug localization approaches. To do this, we adapted the approach proposed by Mahmud *et al.* [156] to filter or boost code files retrieved by existing bug localization approaches using UI information, given an input bug description. That is, we implemented an end-to-end automated approach that receives a bug description and set of screens and (i) automatically identifies buggy UI screens from a corpus of automatically crawled UI screens for a given app, (ii) computes semantic similarity between the bug description and code files to retrieve potentially buggy files, and (iii) boosts the rankings of retrieved code files related to identified Buggy UI screens and filters out those that are unrelated. Using two bug localization tools applied to 79 bug reports we found that incorporating information from the automatically identified Buggy UI screens can lead to 4.77% to 16.69% improvement in Hits@10, compared to baseline techniques that do not use UI data. In summary, this work makes the following contributions:

1. A novel synthetic dataset of OB descriptions and associated UI screens/components that can be utilized for pre-training/fine-tuning Buggy UI Localization techniques.
2. A novel dataset of real-life OB descriptions and associated buggy UI screens/components that can be utilized for evaluating Buggy UI Localization techniques.
3. An empirical study that evaluates the effectiveness of five DL techniques of different kinds, used for Buggy UI Localization. The study reveals different performances achieved by the approaches for different types of OBs and retrieval tasks.
4. An empirical study that illustrates the usefulness of automated Buggy UI Localization for improving the effectiveness of techniques that localize buggy code.
5. A complete replication package that provides data, source code, results, and instructions to reproduce our studies [37].

6.2 Background, Problem, and Motivating Example

6.2.1 Bug Descriptions and App UI Screen/Components

In this work, a ***bug description*** is the observed or incorrect app behavior (OB) textually described in a sentence of a bug report. We focus on descriptions of bugs that manifest visually on the device screen, which is the majority of bugs reported for the Android ecosystem [69, 195]. Figure 6.1 shows an example of a real-life bug report for WiFi Analyzer [40], an app for monitoring the strength and channels of surrounding WiFi networks [39]. The bug/OB descriptions in the bug report are underlined and describe a bug in which the app fails to show the keyboard to enter the WiFi's SSID.

App ***UI screens*** implement one or more app features and represent the canvas upon which UI components (*a.k.a.* widgets) are drawn. ***UI components*** are elements rendered on a UI screen (*e.g.*, buttons, text fields, or checkboxes) that allow end-users to interact with the application. A screen is composed of a hierarchy of UI components and containers

Title: Can no longer enter text in SSID Filter TextView
Description: Cannot enter any text in the SSID Filter field.

Steps:
1. Click on Filter icon.
2. Click/tap on SSID Filter text field.
3. Keyboard does not pop up.

Expected Behaviour:
Should display keyboard and allow you to enter SSID filter text.

Figure 6.1: Bug report #191 from the WiFi Analyzer app [40]

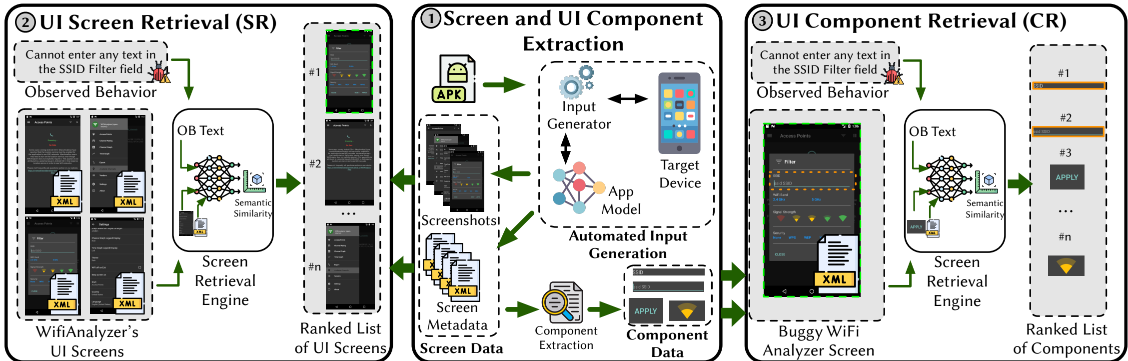

Figure 6.2: Example of the UI screen/component localization process for an OB/bug description of the WifiAnalyzer app [40].

(*a.k.a.* layouts) that group components together [13]. Figure 6.2 shows examples of UI screens and their components for the WiFi Analyzer app. In this work, a UI screen is represented as a *screenshot* and its corresponding *UI hierarchy* of components/containers described in metadata. Each UI component is represented by a set of *attributes*, including the component type (*e.g.*, Button [13]), its label or text shown on the screen, an ID, a description, and various visual properties such as the component's visibility and size.

Buggy UI Screens/Components display unexpected, incorrect behavior of an app.

6.2.2 Problem and Motivating Example

We envision a system that suggests to the developer the UI screens (*i.e.*, app screenshots) that display or is related to the buggy app behavior reported in the bug/OB descriptions of a bug report. The developer would then inspect the suggested UI screens and select one or more screens that s/he deems display the reported bug. The system would then identify (and highlight) the components in the selected buggy UI screens that are most related to the reported bug. The suggestions of this system can help developers automatically localize buggy UI screens/components [170, 220, 52, 74, 237], but also understand the reported bug, and assist them in other bug management activities (*e.g.*, bug reproduction). Additionally, this system can be useful for various bug report management tasks, as it can provide information to existing automated techniques that aim to reproduce bugs [231, 230], generate test cases [105], assess the quality of bug reports [195, 78, 194], and perform bug localization in code [156]. In fact, in Section 6.4, we demonstrate the usefulness of this system for augmenting techniques that perform traditional bug localization to functional code. We investigate different underlying approaches for our envisioned system to localize buggy screens/components from bug descriptions.

While bug reports provide information such as the steps to reproduce the bug (S2Rs), which can be used to identify the buggy screens/components, we focus on OB descriptions for at least two reasons [86]: (1) they convey the faults observed by the user, and (2) they are often written using different wordings (even for a single bug type—see fig. 6.1). The S2Rs (and other elements) in bug reports do not necessarily describe a bug and their discourse is more constrained compared to that of OBs [86].

We formulate automated buggy UI localization as two retrieval tasks (see fig. 6.2): *screen* and *component* localization. In ***screen localization (SL)***, a bug/OB description (*i.e.*, the ***query***) is the input to a retrieval engine that searches the space of UI screens of a given app and retrieves a list of UI screens ranked by their similarity to the bug description, which indicates the likelihood of a UI screen to show or be affected by the

bug described by the query. The left side of fig. 6.2 illustrates the screen localization process for one OB description from the bug report shown in fig. 6.1. The highlighted UI screen with the green border is the buggy screen (initially unknown to the developer). The two best approaches we studied (BLIP & SBERT) are able to retrieve the buggy screen as their first suggestion. In ***component localization (CL)***, the retrieval engine searches the space of UI components of a given screen and retrieves a list of components ranked by a similarity score that indicates the likelihood of the components to show or be affected by the bug. The right side of fig. 6.2 illustrates the component localization process for the buggy UI screen of the bug description. The highlighted components in orange are the ones that the bug description refers to, hence they are expected to be ranked higher by the component retrieval engine. The two best-performing approaches we studied (SBERT & BLIP), rank the buggy components in the first position.

We note that screen and component localization is impacted by the amount of information that a bug/OB description contains (*i.e.*, ***query quality***) and the difficulty in retrieving buggy UI screens/components (*i.e.*, ***retrieval difficulty***). As such, if the bug description is poorly written or does not provide enough information about the bug (which is common in bug reports [78, 195]), then a retrieval engine (or even a human) would have a hard time identifying the buggy UI screens/components (if not familiar enough with the app). This problem is exacerbated by the fact that the same bug can be described in a variety of ways [86] – see fig. 6.1. Even if the OB is clear and informative, identifying the buggy UI screens/components can be challenging when numerous similar UI screens/components exist in the app. As an example, consider the last OB/bug description from the Wifi Analyzer app shown in fig. 6.1: "Keyboard does not pop up". The best approaches we studied (BLIP and SBERT) retrieved the buggy screen and components in positions 21/16 and 6/12, respectively. This illustrates the difficulty of buggy UI localization, hence in our study, we assess the performance of various approaches considering bug descriptions of different quality levels and retrieval difficulty.

6.3 Study 1: Buggy UI Localization

This study aims to investigate different methods for automatically locating buggy UI screens/components based on bug/OB descriptions and measure their effectiveness for this problem. To that end, we investigate existing retrieval approaches that leverage textual and/or visual information from the bug descriptions and UI screen/components, to perform screen and component localization. With this in mind, we formulate three research questions (RQs):

RQ$_1$: *How effective are retrieval approaches at locating buggy UI screens (SL) from bug descriptions?*

RQ$_2$: *How effective are retrieval approaches at locating buggy UI components (CL) from bug descriptions?*

RQ$_3$: *How effective are retrieval approaches for different query quality and retrieval difficulty levels?*

To answer these RQs, we selected three supervised approaches of various kinds (section 6.3.1). Then, we constructed a real-world dataset for evaluating the effectiveness of the approaches (section 6.3.3). We executed the approaches (section 6.3.4) and measured their performance with standard retrieval metrics (section 6.3.5).

6.3.1 Retrieval Approaches

We investigated three deep learning (DL)-based approaches and GPT4 that support text-to-text or text-to-image retrieval for accomplishing OB-UI mapping tasks.

OpenAI's Embedding Models (or **OpenAI-txt-embed**) [50] The embedding model we used was released by OpenAI recently: *text-embedding-3-large*. The textual information is sent to the embedding API endpoint to get an embedding that can be compared with an embedding of another description to establish their semantic similarity. Accord-

ing to OpenAI, *text-embedding-3-large* is their best-performing embedding model and can create embeddings with up to 3072 dimensions.

SentenceBert (or **SBert**) [179] is a neural text-based language model, derived from a pre-trained model, BERT [99]. SBERT is utilized the same way OpenAI's embedding model is used for CL and SL. SBERT augments the traditional BERT model with siamese and triplet networks allowing for better support of tasks such as clustering and semantic search with less computational overhead. SBERT can be utilized for both screen and component localization using the textual information in the bug description and UI screens/components.

Generative Pre-trained Transformer 4 [48] (or **GPT-4**), created by OpenAI, is a multimodal large language model (LLM). Unlike LLMs limited to textual data, GPT-4 excels in handling both text and image inputs to produce text outputs. Multimodal LLMs are usually trained on large-scale datasets which include not just text but also various visual elements and can support a wide range of tasks with prompt engineering. By simply describing natural language prompts, LLMs can be invoked to perform specific tasks without requiring additional training or hard coding. In this work, we prompt GPT-4 for both screen and component localization. While zero-shot learning has been proven effective for simple tasks, its effectiveness diminishes when confronted with more complex tasks that require logical reasoning and a multi-step resolution process, such as arithmetic, commonsense, and symbolic reasoning [141, 176, 209]. To tackle this, researchers have developed a strategy where they use step-by-step explanations, or rationales, to help LLMs understand and solve these problems, known as chain-of-thought reasoning [209, 227]. Our SL/CL tasks are inherently complex, necessitating logical reasoning to interpret bug reports and visual data for identifying buggy UI elements. Consequently, we adopt the zero-shot chain-of-thought reasoning [141] to utilize GPT-4 to locate buggy UI elements.

Clip [175] is a neural multi-modal vision/language model that can learn semantic embeddings from text and images, via a contrastive architecture. Given a text-image pair, CLIP can determine the similarity between the text and the image, hence it can be

used for text-image or image-text retrieval. Clip was trained on a wide range of open-source text-image datasets for a variety of domains and has been evaluated on multiple downstream tasks under zero-shot settings, achieving similar performance as ResNet-50 [118] (*e.g.*, on ImageNet [98]). Clip can be utilized for both screen and component localization using textual information from the bug description and visual information from UI screens/components.

Blip [146] is a neural multi-modal model for vision-language understanding and generation tasks. It utilizes a multi-modal encoder-decoder component (MED) and a captioning-filtering (CapFilt) component, incorporating three types of losses to learn representations from text and images: contrastive, matching, and language modeling losses. Blip's CapFilt improves the quality of training examples by generating synthetic labels for a given image and by filtering noisy generated image-text pairs. Blip exhibited state-of-the-art performance on image-text retrieval, image captioning, and question-answering tasks. We used a Blip version optimized for text-image retrieval tasks, which implements the multi-modal contrastive and matching losses only. Blip can be utilized the same way Clip is used for screen and component localization.

While these models have been pre-trained with general-purpose data (*e.g.*, images of landscapes and corresponding captions/descriptions), they have been shown to perform well under zero-shot settings [179, 146, 175] and also have been fine-tuned for diverse tasks [223, 62, 95, 178]. This means they can learn rich visual and textual representations that can be transferred to solving a diverse set of problems.

Besides the five models, we considered models specifically designed for mobile app UI understanding tasks, including UIBert [61], VuT [151], and Screen2Vec [147]. However, UIBert and VuT's source code and pre-trained models are not available and Screen2Vec would require a significant adaptation effort for our task as the model is only designed for generating UI screen embeddings from screen text and UI hierarchies; extra modules would be required to adapt this model for Buggy UI Localization. We should note, though, that we experimented with it as a zero-shot encoder to represent UIs and

with an SBERT model for representing bug descriptions, computing the cosine similarity on both embeddings to establish similarity. Unfortunately, this led to poor performance for both CL and SL, hence we decided to not report their performance in this work. Our replication package contains those results [37].

Finally, we selected **Lucene** [115] as a baseline technique for text retrieval. LUCENE is an unsupervised approach that combines the classical vector space model (VSM), based on the TF-IDF representation, and the boolean text retrieval model, to compute the (cosine) similarity between a query and a document. LUCENE can be utilized for SR/CR using the textual information in bug descriptions and UI screens/components.

6.3.2 Synthetic Dataset Construction

We built a synthetic dataset of bug descriptions and associated UI screens/components to fine-tune the selected DL models, given that no dataset exists for the OB-UI mapping problem. We opted to create a synthetic dataset because the models require large amounts of training data, and it would be extremely expensive to build it manually (*e.g.*, manual issue analysis).

To build our dataset, we leveraged the RICO-SCA dataset by Wang *et al.* [206], which contains ≈24k UI screens (with screenshots, UI hierarchies, and metadata) for ≈6.4 Android apps of different domains. This dataset is a subset of the popular RICO dataset [38] and includes only UI screens with *complete* and *accurate* UI screenshots, hierarchies, and metadata.

We adopted a multi-step procedure to generate screen/component OBs and associated UI screens/components. From this data, we defined the queries, the ground-truth UI screens/components, and the corpus required for component and screen retrieval. Our procedure was designed to generate realistic OB/bug descriptions from the RICO-SCA UI screens/components, with the goal of generating a diverse and reasonable dataset that could be used to help the models learn associations between related bug descriptions and UI screens, even though the screens may not display a buggy app behavior.

6.3.2.1 Synthetic OB Generation

To generate OBs, we designed bug description templates for a variety of mobile app bug types [102] and specific UI component types. The templates were designed based on discourse patterns that prior work derived from real-life bug descriptions [86]. We designed and implemented heuristics to extract the information from applicable UI screens/components and fill in the templates. These heuristics were executed to generate bug descriptions. This process was iterative and included manual validation of the templates and generated OBs, which resulted in template and heuristic refinement to maximize data quality. Similar procedures have been adopted to generate synthetic UI-level actions in the context of command-UI grounding [61].

Table 6.2 provides examples of the OB templates we designed and the OB descriptions the templates generate. Each template was created based on one or more discourse patterns and is applicable to one bug type and different kinds of UI components. For space reasons, we provide examples of these elements, but the full catalog of templates (and their implementation) can be found in our replication package [37].

We now provide details about our OB generation procedure. Selecting Bug Types. Based on the mobile app bug taxonomy from Escobar *et al.* [102] and a manual analysis of bug reports, we selected nine (9) bug types that cover a variety of app problems, including crashes, cosmetic issues, and incorrect GUI behavior. Specifically, we selected seven UI bug types from Escobar *et al.*'s taxonomy [102] (*e.g.*, *Component with wrong dimensions* and *Wrong text in component* – see Table 6.2), which was derived from manual analysis of Android app bug reports, bug-fixing commits, technical forum discussions, Android documentation, prior bug taxonomies, and online app reviews. Additionally, we selected two additional UI bug types (*i.e.*, *Unclickable/uneditable components*), not found in Escobar *et al.*'s taxonomy, by analyzing the titles of sampled bug reports we collected to guide the design of component/screen OB templates (more details below).

Selecting Applicable UI Components. Next, we identified the UI components that (1)

could be affected by the bug types, and (2) contained potentially useful information to generate OB templates. For example, *Uneditable component* bugs are exclusive to EditText components (*i.e.*, text fields). Another example is a bug about *Elements not listed in the right order*, which refers to UI components displayed in the wrong order. Since these are bugs that affect regions of the screen (*i.e.*, they would correspond to screen OBs), we identified component containers such as ListView and ScrollView as potentially useful to define proper templates and the screens these templates apply to.

Assigning Discourse Patterns to Bug Types. Prior work identified 90 discourse/linguistic patterns that bug reporters recurrently use to express OB/bug descriptions [86], based on the manual analysis of ≈3k bug reports from nine software projects (including one Android app). Of these 90 patterns, 12 are commonly used by reporters and were found in ≈83% of the 3k reports. We assigned subsets of these 12 patterns to one or more bug types when applicable. For example, *incorrect text* bugs can be expressed using the discourse pattern NEG_AUX_VERB, which refers to OB sentences containing auxiliary verbs that convey a negative discourse (*e.g.*, "I cannot zoom in on the filter screen").

Defining OB Templates. For each bug type, we generated one or more templates applicable to corresponding UI components or screens, by creating a phrase that followed a discourse pattern. An OB template contains: (1) *slots* that represent either components or screens (noted with squared brackets in table 6.2), and are filled in from UI screen/-component metadata, (2) *variable phrases* are expressions that can be substituted with synonyms to increase OB diversity (noted with braces in table 6.2), and (3) *fixed phrases* are expressions that connect *slots* and *variable phrases*. An example of a template for *incorrect dimension* bugs is "[component] {dimension} is {wrong}", where [component] is the slot that represents a UI component expression and {dimension} and {wrong} are variable phrases that can be used as is or replaced with synonyms such as *size* and *incorrect*, respectively.

Since we initially struggled to define screen OB templates, we guided our template design based on a manual analysis of Android app bug reports. From the original 6,356

Table 6.1: Statistics of the synthetic dataset of OB/bug descriptions

	# of UI screens		# of UI components		# of Component OBs			# of Screen OBs			# of Components & Screen OBs		
	Per app	**Total**	**Per screen**	**Total**	**Per app**	**Per screen**	**Total**	**Per app**	**Per screen**	**Total**	**Per app**	**Per screen**	**Total**
Training	3.6 (2)	15,268	12.4 (10)	189,612	416.4 (222)	115.5 (85)	1,751,224	20.7 (12)	7.1 (6)	61,631	252.4 (66)	76.0 (37)	1,812,855
Validation	3.8 (2)	3,685	12.5 (9)	45,915	429.6 (226)	114.2 (84)	417,538	22.2 (13)	7.1 (6)	15,747	257.9 (64)	73.8 (35)	433,285
Testing	3.9 (3)	4,949	12.6 (10)	62,444	449.6 (241)	116.1 (85)	570,953	22.3 (13)	7.1 (6)	20,749	269.1 (65)	75.5 (73)	591,702
Total	**3.7 (45)**	**23,902**	**12.5 (10)**	**297,971**	**424.9 (226)**	**115.4 (85)**	**2,739,715**	**21.3 (13)**	**7.1 (6)**	**98,127**	**256.5 (65)**	**75.6 (37)**	**2,837,842**

app/screen numbers are average values (median values in parenthesis)

bug reports mined by Wendland *et al.* [211], from which the AndroR2 dataset was built, we took a representative random sample of reports (362 in total at a 95% confidence level), analyzed their titles, and found that 59 titles describe screen-related bugs (*i.e.*, those affecting the entire screen or regions of it). From these titles, we designed screen OB templates, after we identified the corresponding discourse patterns from Chaparro *et al.* [86].

Since screen OBs are more general than component OBs, as they refer to app features or screen-level app behavior rather than specific components, we designed the screen OB templates for screens with certain characteristics rather than templates applicable to every screen in our dataset. With this strategy, we also aimed to help the models learn better the associations between the OBs and screen characteristics. For example, the S2 template in table 6.2 refers to updates that should happen in settings screens (identified by a search for keywords such as *settings* or *configuration* in the screen/component metadata).

In total, we designed 38 templates, 21 for generating component OBs, and 17 for generating screen OBs.

Designing Heuristics. We defined a predefined set of phrases to fill in the *variable phrases*, by selecting one phrase randomly every time the template was applied.

To fill in template *slots*, we defined rules to compose phrases corresponding to components and screens. These rules would select UI component metadata, particularly, the component text (*i.e.*, label), ID, description, type, and location, if/when available. For

components, we selected the first available attribute among the text, description, and ID (in that order), as the component phrase for the slot. To increase diversity, we randomly appended the component type and/or the component location on the screen (based on their coordinates and size). For example, for a text field with attributes (text: null, description: "SSID", ID=null, type="EditText") possible created component phrases are *SSID*, *SSID text field*, *SSID text field at the top left corner*, or *SSID at the top left corner*. Similar rules were defined for generating screen phrases (*i.e.*, extract screen names from pre-processed activity names).

Generating OB Descriptions. The defined heuristics were implemented via algorithms that applied the templates on the ≈24k UI screens. Only the visible UI components that are leaf nodes in the UI hierarchy were processed to generate OB descriptions, as we consider them as common candidates for reporters to include in their bug descriptions. The algorithms generated unique OBs and avoided cases where the OBs were excessively long (*e.g.*, from long component metadata).

In total, we generated ≈2.8M OB descriptions (75.6/256.5 per screen/app on average)—see table 6.1. From these, ≈2.7M are component OBs (115.4/424.9 per screen/app on average) and ≈98k are screen OBs (7.1/21.3 per screen/app on average).

OB Template/Heuristic Validation and Refinement. The templates, heuristics, and generated OBs were validated in multiple review sessions where a sample of OBs, generated by different templates, was manually inspected by two or more researchers. The researchers reviewed grammar, semantics, and how realistic the bug descriptions were, suggesting adaptations to improve the templates and heuristics.

6.3.2.2 Synthetic Retrieval Data

We kept track of the UI screen used to generate each screen OB and the UI screen/components used to generate each component OB. Based on this information, we defined the queries, ground truth, and corpus for both screen (SR) and component retrieval (CR).

For SR, the queries are *both* component and screen OBs (≈2.8M total), the ground

Table 6.2: Examples of templates used to generate synthetic OB/bug descriptions

Bug Type [102]	Template	Generated OB (App name)	Screens & Components	Discourse Pattern [86]
Templates for Screen OBs				
Elements not listed in the right order	**S1**: After {filtering} the [list], the order of [list item] did not change	After filtering the view items list, the order of new cars did not change (Autoportal)	Screens with ListViews, ScrollViews, etc.	AFTER_NEG
UI refresh issue	**S2**: {Changes} in the [screen] setting {will not apply immediately}	Changes in the app settings will not apply immediately (smartChord)	Settings screen	NEG_AUX_VERB
Issues in view animation	**S3**: I cannot {zoom on} the [screen]	I cannot pinch on the home wiki screen (FANDOM)	All screens	NEG_AUX_VERB
Templates for Component OBs				
Component with wrong dimensions	**C1**: [Component] {size} does not {match the expected size}	The More options button does not match the expected size (Daily Reflections)	All components	NEG_AUX_VERB
Wrong text in component	**C2**: {Wrong} text in [component]	Incomplete text in Invite your friends textview (Chat Rooms)	Buttons, TextViews, etc.	VERB_TO_BE_NEG
Unsupported style in component	**C3**: [Component] {shows} {incorrect} {color}	ECONOMY button shows incorrect text color when clicked (PlayCast)	Radio buttons, TextViews, etc.	NEG_ADV_ADJ

truth is their corresponding UI screen (one per query), and the retrieval corpus comprises the screens that belong to a specific app (there are 3.7 screens per app, on average). SR retrieval is performed per app (*i.e.*, rather than across apps), as it would be executed in practice.

For CR, the queries are only component OBs (≈2.7M total), the ground truth is their corresponding UI component (one per query), and the retrieval corpus is the UI components for a given screen (there are 12.5 components per screen, on average).

6.3.3 Real Dataset Construction

We built a dataset of real-life bug descriptions and relevant buggy UI screens/components to assess the effectiveness of the models in a realistic setting. We collected OB descriptions from a set of bug reports. Then, for each report, we collected a corpus of UI screens used for screen localization, identifying the screens and components displaying the reported

bug. Finally, for each screen, we extracted the UI components visible to the user as the corpus for component localization.

6.3.3.1 Bug Report Selection

Since one of our goals, later in this work (see section 6.4), is to assess the usefulness of Buggy UI Localization models for Buggy Code Localization (*i.e.*, in short, bug localization), we took a pragmatic approach to select the bug reports for this study. We started selecting the 80 bug reports included in the bug localization dataset provided by Mahmoud *et al.* [156]. In this way, we could reuse this data for this study and the bug localization study reported in section 6.4.

Mahmoud *et al.*'s dataset was created based on the AndroR2 dataset [211, 130], which consists of 180 manually reproduced bug reports for popular open-source Android applications hosted on GitHub. These reports were systematically collected from the project's issue tracker following a rigorous procedure (see [211, 130] for details). The dataset provides APK files, buggy and bug-fixing commits, GitHub issue links, and scripts that replicate the reported bugs (crash-, output-, navigation-, and cosmetic-related bugs [211, 130]).

To construct the bug localization dataset, Mahmoud *et al.* employed rigorous manual procedure to collect ground-truth buggy Java files for the 180 AndroR2 reports. They reproduced the reported bugs in a Pixel 2 Android emulator and for the reproducible bugs, they inspected the bug-fixing commits to identify the Java files and lines of code that were changed to fix the reported bug. From the 180, 100 reports were discarded because: (1) their bugs are not reproducible, (2) included fixed non-Java files only, (3) are no longer publicly available, or (4) included ambiguous code changes or commit IDs. The 80 remaining reports were used for bug localization experiments by Mahmoud *et al.*, and hence, we used them for our study. However, when collecting ground truth data for Buggy UI Localization (see section 6.3.3.4), we discarded one bug report (from the GnuCash app [**?**]) because we were unable to reproduce the reported bug, thus leaving 79 reports. To expand the set of bugs usable for this study, we selected 14 extra bug reports

from the 100 discarded ones whose bug fixes were in XML resource files as opposed to Java code and discarded 6 reports because we obtained errors trying to collect the retrieval corpus for those reports (see section 6.3.3.3). This resulted in 8 extra bug reports, for a total of 87.

From the 87 bug reports (1 to 8 per app), 32 describe an output problem, 23 report an app crash, 23 describe a UI cosmetic issue, and 9 report a navigation problem. The bug reports correspond to 39 Android apps (*e.g.*, GnuCash [21], Mozilla Focus [17], K-9 Mail [25], WiFi Analyzer [40], Images to PDF [23]) of different domains (*e.g.*, finance tracking, web browsing, emailing, WiFi network diagnosis, and image conversion) and UI layouts.

6.3.3.2 Bug Description Annotation

To collect bug descriptions, two researchers inspected and annotated the OB sentences in the title and descriptions of the 80 bug reports. Based on the definition of OB and the criteria to annotate OB sentences defined by Chaparro *et al.* [86], one author marked every sentence of the bug reports that conveyed a bug manifesting in the UI. The second author verified the first author's annotations, marking missed OB sentences and incorrect annotations, resulting in an agreement for 1774/1807 (≈98%) sentences (Cohen's kappa [18] of 0.91). The researchers solved disagreements via discussion and consensus. Reasons for disagreement included mostly mistakes and misinterpretations (*e.g.*, when sentences described root causes in the code, rather than UI faults). In total, 228 sentences were identified as OB/bug descriptions, which represent the queries for retrieval.

6.3.3.3 Retrieval Corpus Collection

To build the retrieval corpus for each bug description, we require the set of UI screens/-components of the apps, including the buggy screens/components. To collect these data, we employed: (1) a record-and-replay methodology used in prior studies [96, 195], and (2) an automated app exploration methodology used by Chaparro *et al.* [78].

The goal of the record-and-replay methodology was to collect the buggy UI screens for each bug report and the UI screens navigated while reproducing the bugs. Two researchers manually reproduced the reported bugs by executing the reproduction steps from the bug report on a Pixel 2 Android emulator (the same emulator used by Mahmoud *et al.* [156]). While reproducing the bugs, the researchers used the AVT tool [96, 195] to collect UI-event traces and a video showing the user interactions of the app and the bug itself [164, 162]. These traces were replayed on the emulator via the TraceReplayer tool [156], which is able to automatically collect app screenshots and UI hierarchies/metadata for the exercised app UI screens.

The purpose of the automated app exploration methodology was to collect as many UI screens for building the corpus. We executed a version of the CrashScope tool [164] that implements multiple exploration strategies to interact with the UI components of app screens in a comprehensive way, trying to exercise as many screens as possible. In the process, CrashScope collects app screenshots and XML-based UI hierarchies/metadata for the exercised app UI screens, in the same way TraceReplayer does it.

Since these two approaches can generate duplicate UI screens, we employed the approach by Chaparro *et al.* [78] to produce a unique set of UI screens for each of the 87 bug reports. This approach parses the hierarchies of the collected UI screens for an app and establishes uniqueness between two screens if they have the same hierarchical structure (based on component types, sizes, and parent-children relationships). This implies that two UI screens with the same structure but different text shown on the screen are considered the same. Unique UI screens for each report were identified using a SHA hash created from the hierarchy structure of the UI screens.

To create UI component corpora for each of the buggy UI screens we parsed the UI hierarchy of the screen and identified the visible leaf components, which are typically the ones shown to the user on the mobile device. However, we discard layouts and other containers, thus focusing on labels, buttons, text fields, and other UI components that users typically interact with.

This procedure resulted in UI screen corpora containing ≈26 UI screens per bug report on average, which will used for screen localization. We also collected UI component corpora containing ≈17 UI components per screen on average, which will be used for component localization.

6.3.3.4 Ground Truth Construction

During multiple annotation sessions, four researchers (*a.k.a.* annotators) first read and understood the reported bugs, watching (if needed) the bug reproduction video collected during the corpus collection step. Then, the annotators inspected the app screens from the corpus to identify the buggy screens shown in the video and marked them as such in a spreadsheet. The annotators identified and marked the buggy UI components in the same spreadsheet. Each bug report was assigned to two annotators, making sure the annotators had an even number of bug reports to annotate. For each bug report, the first annotator identified the buggy UI screens/components and then the second annotator validated the annotation by the first annotator. Both annotators would follow the procedure described above, marking potential disagreements in a shared spreadsheet. At the end of each annotation session, the annotators would meet to discuss disagreements (mostly because of misinterpretation of the bugs), and reach a consensus to produce the final set of buggy UI screens/components.

Besides identifying the buggy UI screens/components, the annotators rated the quality of the bug descriptions based on the amount of information they provided to understand the bug. Since the Buggy UI Localization tasks focus on individual bug descriptions, the annotators judged the quality of single OBs in isolation. The annotators agreed on a quality rating on a 1-5 discrete scale. A rating of 1 means the bug description does not contain useful information to understand the problem. Conversely, a rating of 5 means the description contains complete information to understand the bug. A rating of between 1 to 5 indicates that there is missing information in the OB that hinders bug comprehension. Additionally, the annotators marked each bug description as *easy* or *hard* to localize, based

Table 6.3: Dataset statistics for SL and CL

Statistic	SL	CL
# of retrieval tasks/queries	228	254
# of hard-to-retrieve tasks	111	130
# of easy-to-retrieve tasks	117	124
Avg. # of buggy UI screens/comp.	2.06 (2)	1.86 (1)
Avg. of corpus size	25.97 (22)	17.11 (14)

Average (median) values per query/retrieval task

on the difficulty they encountered in identifying the buggy screens and components. A common reason why bug descriptions were judged as *hard to retrieve* was that multiple UI screens/components were similar, yet one or a few were really displaying the reported bug. During the reconciliation sessions, disagreements were discussed and solved to produce the final query quality and retrieval difficulty category for each bug description.

6.3.3.5 Summary of the Collected Retrieval Data

For screen localization (SL), the dataset contains 228 queries (see table 6.3) with 2.1 buggy UI screens on avg. as ground truth and 26 screens in the corpus on avg. Each query represents a unique screen retrieval ask.

For component localization (CL), the queries can be reused for multiple retrieval tasks. Since the OBs may be associated with multiple buggy screens, each screen represents a task. This means we created 254 queries (or retrieval tasks), with 1.9 buggy UI components as ground truth and 17.11 components in the corpus on avg.

In summary, we collected: OB descriptions (*i.e.*, the queries), the retrieval corpus of UI screens/components for each query (including app UI screenshots and cropped component images, and their UI hierarchy with associated metadata: component text, ID, *etc.*), and identified buggy UI screens/components (*i.e.*, the ground truth).

6.3.4 Approach Execution

In this research, we analyze five DL models: two text-based (OpenAI's embedding model and SBert) and three multimodal (GPT-4, Clip, and Blip). We fine-tuned three of them—SBert, Clip, and Blip, which are supervised models—using the synthetic dataset to assess their performance in both zero-shot and fine-tuned scenarios. For the two OpenAI models, we leveraged their APIs for zero-shot evaluations due to their exceptional zero-shot capabilities: the *text-embedding-3-large* model and *gpt-4-vision-preview* model, which represent the most performant model of their respective domains. OpenAI has not provided access for fine-tuning these two models.

6.3.4.1 GPT-4 Customized Prompting and Execution

We leverage the sophisticated image comprehension capabilities of GPT-4, aiming to pinpoint buggy elements or screens based on textual descriptions of bugs. Through a trial-and-error process, We craft effective prompts for both Component Localization (CL) and Screen Localization (SL) tasks. Out of a selection, five bugs were chosen for prompt construction, with the remainder allocated for evaluation. This sample covers 5 outputs, 3 report an app crash, 3 describe a UI cosmetic issue, and 1 report a navigation problem, which mirrors the diversity encountered within the whole real dataset.

The prompt framework for the **screen localization**, detailed below fig. 6.3, begins with defining "Observed Behavior" and presenting the corresponding behavior, and then the image of the screen. Subsequently, GPT-4 is instructed to compute a similarity score reflecting the degree to which the UI screen shows the observed behavior or could potentially manifest it. To ensure GPT-4's success in this task, it was crucial to direct its analysis based on specific guidelines distilled from our trial-and-error testing. These guidelines include:

- Analyze only the elements that are immediately actionable and in the user's current focus. Background elements, even if related to the bug, should be disregarded if

they are not in the active foreground and cannot be interacted with in the present state of the UI.

- Directly relate the observed behavior with visible elements in the screenshot, without speculating on what isn't shown or inferring additional functionality. You should not lower the score just because the bug is not actively occurring.

- Avoid assumptions about what is not explicitly shown.

These principles are designed to direct GPT-4 towards concentrating on visible and active UI components, overlooking background details, and preventing speculative reasoning about unobserved functionalities or unnecessary assumptions. This guidance is essential to ensure that GPT-4 does not unfairly reduce scores due to bugs not actively occurring, a frequent observation in our pilot study. Given that images, unlike videos, cannot depict ongoing actions, it's necessary to underscore this point. For instance, in the case of a bug report from Focus-android [7] that describes the observed behavior as *"URL bar text sometimes cleared when pressing near the URL bar"*, responses such as *"However, it does not capture the bug occurring or provide evidence that it could occur in this state."* illustrate the need for this specific guideline.

The prompt template for **component localization**, is outlined in the fig. 6.4, initiating similarly to the SL prompt by defining and presenting the observed behavior. However, given the necessity for component localization to distinguish buggy components within a screen's component space (as outlined in section Section 6.2.2), a key distinction for CL is we provide not just the image of the component but also the originating screen image for contextual understanding. This dual imagery enables GPT-4 to more accurately interpret the component's image. This difference necessitates a unique prompt structure for CL, considering its focus on the single component image, unlike SL's focus on the entire screen. Through a process of trial and error with the same five bugs used for SL, we developed a set of four tailored guidelines for CL:

- Focus solely on the cropped image's UI component, assessing its visible functionality and interaction elements (e.g., buttons, checkboxes) as they relate directly to the observed behavior. High scores require clear, visible evidence of direct capability to enact the described behavior, without assumptions or context extrapolation.

- Focus strictly on the UI component shown in the cropped image. Consider if this exact component could be where the observed behavior directly occurs.

- Do not factor in indirect interactions or the possibility of a component affecting the observed behavior from another location.

- Ignore whether the action described in the observed behavior is actively happening in the cropped image. The focus is on the potential relevance of the UI component to the described behavior.

These instructions are crucial in guiding GPT-4 to avoid errors such as overgeneralization and inappropriate associations, by emphasizing the importance of focusing on the functionality and location of the UI component without unnecessary speculation.

Additionally, for both SL and CL, we ask GPT-4 to provide its thought process in a step-by-step manner before computing the final similarity score. Inspired by literature on Zero-shot Chain of Thought (Zero-shot-CoT) reasoning [141], we found that LLMs can effectively perform zero-shot reasoning by prefacing with "Let's think step by step" during our trial-and-error process. Thus, we incorporate *"Break down your reasoning into sequential steps before returning the similarity score"* into our prompt, opting for a Zero-shot-CoT approach rather than CoT or Few-shot-CoT is due to the practical constraints of embedding images within GPT-4 API prompts, and we verified its efficiency through evaluation.

Finally, we instruct GPT-4 to present the similarity score in a standardized format: "Similarity score: 60", enabling the use of regular expressions for score extraction and subsequent evaluation. Given the non-determinism (*i.e.*, inconsistent results with identical

prompts) [1] of generative AI models such as GPT-4, we call the GPT-4 API three times for every query and calculate the average similarity score. To handle scenarios where we encounter errors such as "too many requests" or receive unhelpful responses like "I cannot help," we limit our attempts to a maximum of ten requests. We cease further requests after the tenth attempt, regardless of whether we've received satisfactory responses. Thus, if the first three requests yield valid responses, we discontinue any additional calls to the API.

[1] In the literature, other terms like inconsistency, randomness, and instability are also used to describe the concept of non-determinism.

The sentence below is the Observed Behavior extracted from a bug report for the mobile application {app_name}. The Observed Behavior is the incorrect application behavior manifested visually on the device screen when the user was using the application.

Observed Behavior: {bug_report}

I am also providing a screenshot of {app_name} that shows one UI screen.

Based on the Observed Behavior and the screenshot provided, please compute and return a similarity score in the range 0 to 100 in such a way that:

- A score closer to 100 means that it is more likely that the provided UI screen shows the observed behavior or can cause the observed behavior to manifest, even though the bug is not actively occurring in the image.
- A score closer to 0 means that it is more likely that the provided UI screen does not show the observed behavior or can cause the observed behavior to manifest.

Guidelines for analysis:

- **UI Layer Clarity**: Analyze only the elements that are immediately actionable and in the user's current focus. Background elements, even if related to the bug, should be disregarded if they are not in the active foreground and cannot be interacted with in the present state of the UI.
- **Visual Relevance Evaluation**: Directly relate the observed behavior with visible elements in the screenshot, without speculating on what isn't shown or inferring additional functionality. You should not lower the score just because the bug is not actively occurring.
- **Assumption Exclusion**: Avoid assumptions about what is not explicitly shown.

If you cannot provide a similarity score, please return 0. You need to return the similarity score in this format: "Similarity score: 60". Break down your reasoning into sequential steps before returning the similarity score.

Figure 6.3: **The prompt template for screen localization**

The sentence below is the Observed Behavior extracted from a bug report for the mobile application {app_name}. The observed behavior is the incorrect application behavior manifested visually on the device screen when the user was using the application.

Observed Behavior: {bug_report}

You are provided with two images:

- A cropped image of a UI component.
- A full-screen screenshot of {app_name} for context, illustrating where the cropped component image was taken from.

Your task is to calculate a similarity score based on the observed behavior and the first cropped component image provided in the range 0 to 100 in such a way that:

- A score closer to 100 means that it is more likely that the provided UI component shows the observed behavior or can cause the observed behavior to manifest.
- A score closer to 0 means that it is more likely that the provided UI component does not show the observed behavior or cannot cause the observed behavior to manifest.

Guidelines for analysis:

- **Direct Component Functionality**: Focus solely on the cropped image's UI component, assessing its visible functionality and interaction elements (e.g., buttons, checkboxes) as they relate directly to the observed behavior. High scores require clear, visible evidence of direct capability to enact the described behavior, without assumptions or context extrapolation
- **Specific UI Component Focus**: Focus strictly on the UI component shown in the cropped image. Consider if this exact component could be where the observed behavior directly occurs.
- **Exclusion of Indirect Interactions**: Do not factor in indirect interactions or the possibility of a component affecting the observed behavior from another location.
- **Behavior-Relevance Assessment**: Ignore whether the action described in the observed behavior is actively happening in the cropped image. The focus is on the potential relevance of the UI component to the described behavior.

If you cannot provide a similarity score, please return 0. You need to return the similarity score in this format: "Similarity score: 60". Break down your reasoning into sequential steps before returning the similarity score.

Figure 6.4: **The prompt template for component localization**

6.3.4.2 Model Fine-tuning and Execution

We split the ≈6.4K apps in our synthetic dataset into training (65%), validation (15%), and test sets (20%). The training set (*i.e.*, the OBs and their ground truth for the training apps) was used for fine-tuning the models, the validation set was used for hyper-parameter tuning, and the test set for assessing model performance. We denote the fine-tuned models by their name followed by the ft subscript; for example, we denote fine-tuned CLIP as CLIP_{ft}.

For fine-tuning **SBert**, we required building sets of positive and negative pairs of OB descriptions and textual documents of UI screens/components. UI component documents were defined by concatenating the component text, description, and ID. UI screen documents were defined by concatenating the UI component documents. We experimented with different strategies for building positive and negative pairs, but the one that led to the highest performance was the following. Half of the training OBs and their corresponding UI screen/components were used as positive pairs and the remaining half of OBs were used to create negative pairs. To create these pairs we randomly selected screens from other apps (for SR), or other components from the screen (for CR). Our replication package describes the data creation strategies we experimented with and the results [37].

For fine-tuning **Clip**, we required pairs of OB descriptions and images of the ground-truth UI screens/components. We experimented with different datasets created from the training set. Using the entire training set for fine-tuning proved to be ineffective since a single UI screen/component can have multiple OBs associated with it, which makes it difficult for CLIP to learn the distinction between different pairs containing the same image (given its contrastive architecture). We experimented by using subsets of 1, 5, 10, and 15 OBs per image (selected from different bug types), all leading to better performance on the synthetic test set. In the end, the best performance was obtained by using unique pairs of OBs and images (*i.e.*, one OB per image), preserving the distribution of component/screen OBs of the training data, and using OBs of different bug types across different images.

The performance on this data can be explained by CLIP's contrastive architecture.

Fine-tuning **Blip** requires the same information sources as CLIP. Given BLIP's similarity to CLIP regarding architecture, we experimented with the same subsets of pairs and the unique pairs of OBs and images as used for CLIP.

Once fine-tuned, the models were tested on the synthetic test set as well as on the real dataset. **Lucene**, being an unsupervised approach, was only tested on the aforementioned data, following the same procedure as for SBERT to create textual documents from UI screens/components. Only for LUCENE, we performed standard pre-processing on the queries and documents (lemmatization, stop word removal, *etc.*).

6.3.5 Evaluation Metrics

We used standard retrieval metrics, widely used in prior studies [53, 144, 108, 96], to measure the effectiveness of the models:

- **Mean Reciprocal Rank (MRR)**: it gives a measure of the average ranking of the first buggy UI screen/component in the candidate list given by a model. It is calculated as: $MRR = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{rank_i}$, for N queries ($rank_i$ is the rank of the first buggy UI screen/component for query i).
- **Mean Average Precision (MAP)**: it gives a measure of the average ranking of all the buggy UI screens/components for a query. It is computed as: $MAP = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{BU}\sum_{b=1}^{BU}P_i(rank_b)$, where BU is the set buggy UI screens/components for query i, $rank_b$ is the rank of the buggy UI screen/component b, and $P_i(k) = \frac{buggy_elements}{k}$ is the number of buggy UI screens/components in the top-k candidates.
- **Hits@K (H@K)**: it is the percentage of queries for which a buggy UI screen/component is retrieved in the top-K candidates.

All metrics give a normalized score in [0, 1]—the higher the score, the higher the retrieval performance of the models. We executed the models and the baseline on the query sets for SL and CL and computed/compared the metrics between these approaches.

Table 6.4: Screen/component retrieval results on synthetic data

Approach	Screen Retrieval					Component Retrieval				
	MRR	**H@1**	**H@2**	**H@3**	**H@4**	**MRR**	**H@1**	**H@2**	**H@3**	**H@4**
SBert	0.538	0.355	0.526	0.642	0.726	0.610	0.482	0.599	0.671	0.725
SBert$_{ft}$	0.590	0.418	0.587	0.699	0.774	0.747	0.667	0.740	0.783	0.818
Clip	0.492	0.304	0.470	0.591	0.679	0.474	0.334	0.449	0.522	0.582
Clip$_{ft}$	0.619	0.453	0.626	0.730	0.798	0.704	0.605	0.701	0.754	0.793
Blip	0.543	0.373	0.530	0.633	0.709	0.436	0.253	0.376	0.563	0.629
Blip$_{ft}$	0.628	0.475	0.634	0.724	0.788	0.698	0.612	0.688	0.733	0.768

6.3.6 Results

We present and discuss the effectiveness of the results of the approaches for both screen (SL) and component localization (CL). We focus our discussion on MRR since the other metrics show similar trends to the MRR results for all the models. Our replication package contains the results of all the experiments we conducted [37]. In our pilot study, we used five bugs to create effective prompts for both Component Localization (CL) and Screen Localization (SL) tasks. To ensure fairness in our evaluation, we excluded these five bugs. Consequently, this left us with 215 retrieval tasks/queries for SL and 229 retrieval tasks/queries for CL for the evaluation.

6.3.6.1 RQ$_1$: Screen Retrieval Results

Table 6.5 shows the screen localization performance of the approaches for 215 queries. Only considering zero-shot for all models, the results reveal that GPT-4 performs the highest (0.661 MRR), outperforming the second best OpenAI-txt-embed (0.565 MRR) and the third best Blip (0.438 MRR) with a relative improvement of 17% and 50.9% respectively. This suggests that GPT-4 has a strong ability to locate the most relevant screens. SBert displayed moderate MRR score with 0.423 MRR. The remaining models including Clip and Lucene achieve a similar MRR (Clip: 0.363, Lucene: 0.354). In terms of H@K, GPT-4 consistently outperformed other approaches across all hit rates by a considerable margin, with scores ranging from 0.544 at H@1 to 0.805 at H@5. For

example, it outperforms the models with a maximum relative improvement of 33% H@1 (compared to OpenAI-txt-embed). The three models other than GPT-4 and OpenAI-txt-embed achieve a similar H@1 from 0.197 (Clip) to 0.26 (SBert and Blip).

The exceptional performance of the GPT-4 model in all metrics can be attributed to its advanced natural language and image understanding and reasoning capabilities. It can sufficiently comprehend the intricate details within bug reports and extracts pertinent details from mobile app screens. This ability enables it to effectively bridge the gap between written bug reports and their corresponding UI elements on screens, thus accurately pinpointing errors on screens as described in bug reports.

Surprisingly, The OpenAI-txt-embed approach also performed well, with MRR and MAP significantly higher than those of the SBert, Clip, and Blip models, falling just behind the performance of GPT-4. This trend was consistent across H@1 through H@5, indicating the robustness of the OpenAI-txt-embed approach. Notably, it excels beyond other text-based models like SBert and Lucene. One possible factor for this success could be the utilization of OpenAI's best-performing *text-embedding-3-large* embedding model. Despite the closed-source nature of OpenAI's model that limits insight into its technical details, the outcomes suggest that a sufficiently advanced embedding model could serve as a more cost-efficient yet effective alternative to GPT-4.

Additionally, as the results show, fine-tuning affects the models' performance, except for Clip. It is important to note that the fine-tuning of Blip was done with text-image pairs containing duplicate images across pairs (since there are multiple OBs per UI screen). We attempted to train Blip with the same unique text-image dataset used for Clip, consisting of 14.3k pairs. This was motivated mainly by the Blip's contrastive/matching losses, which might benefit from the unique dataset. However, Blip was not able to learn from this dataset as we did not observe a steady, decreasing training loss. We attribute this to the relatively low size of this dataset. Since Blip is a larger model than Clip (224M vs 151M parameters), given its more robust architecture, it needs more data for training.

Table 6.5: Screen Localization (SL) Results

Approach	**MRR**	**MAP**	**H@1**	**H@2**	**H@3**	**H@4**	**H@5**
SBert	0.423	0.427	0.260	0.417	0.472	0.535	0.559
SBert_{ft}	0.351	0.357	0.220	0.315	0.378	0.425	0.465
Clip	0.363	0.352	0.197	0.315	0.449	0.496	0.583
Clip_{ft}	0.391	0.369	0.276	0.331	0.402	0.441	0.520
Blip	0.438	0.440	0.260	0.449	0.528	0.583	0.661
Blip_{ft}	0.395	0.397	0.205	0.402	0.528	0.567	0.614
Lucene	0.354	0.357	0.228	0.323	0.417	0.472	0.512
OpenAI-txt-embed	0.565	0.530	0.409	0.572	0.647	0.712	0.744
GPT-4	0.661	0.613	0.544	0.642	0.730	0.791	0.805

Further, while the fine-tuning vs zero-shot results seem counter-intuitive at first, this trend can be explained by the fact that 96.5% of the synthetic OBs are component OBs which describe issues with a UI component rather than issues with regions or the entire screen. As explained in section 6.3.2, we opted to design screen OB templates only applicable for selected screens, rather than templates that apply to all the screens, as the latter would likely make it difficult for the models to learn the relationships between the OBs and screens. Given this, we expect the fine-tuned models to perform higher for component OBs than for screen OBs (both can be used for screen retrieval), as the models would learn more from training component OBs for screen retrieval. To validate our expectation, we analyzed the MRR results of the approaches across the two query types in the data: screen and component OBs. We found that all the approaches (except Clip) perform higher on the component OBs than on the screen OBs, which confirms our expectation. However, the non-tuned models show the same trends. Despite the models learn from the synthetic data, the learned patterns seem to not follow the same patterns in the real data. Our future work will investigate if improving the synthetic dataset with extra screen OBs can improve model effectiveness.

6.3.6.2 RQ$_2$: Component Retrieval Results

Table 6.6 shows the component localization results of all the approaches for 229 retrieval tasks. Note that although the number of OBs is 216, some OBs may have a different component corpus for retrieval, one for each buggy screen in the ground truth (*i.e.*, one OB/bug description may correspond to multiple buggy UI screens).

When focusing solely on a zero-shot setting for all models, all DL approaches (besides CLIP) perform substantially higher (0.432+ MRR) than the baseline (LUCENE), which achieves 0.416 MRR and fails to retrieve UI component suggestions in 14 cases. The OPENAI-TXT-EMBED approach exhibits superior performance with the highest MRR of 0.622, indicating its robustness in accurately ranking the correct component at the top of the list. The OPENAI-TXT-EMBED method's Hit Rates also show outstanding results, progressively increasing from 0.467 at H@1 to 0.79 at H@5, suggesting that it not only locates the correct component but also consistently ranks it highly.

GPT-4 shows second-highest performance with an MRR of 0.559. It maintains strong Hit Rates across the board, culminating in an H@5 of 0.799, which is even better than OPENAI-TXT-EMBED model, demonstrating its effectiveness in component localization. However, GPT-4 significantly trails the embedding model in terms of MRR and the metrics H@1 to H@3. Through a detailed manual review of the responses, we found multiple factors contributing to its inability to accurately locate the correct buggy components. A notable issue is GPT-4's unstable capability to comprehend and accurately describe component images within the given context. This inconsistency was evident as GPT-4 occasionally failed to process images, responding like *Unfortunately, I cannot assist with this request.*, or *I'm sorry but I cannot provide a similarity score between the observed behavior and the component image without the actual images to analyze. If you can provide the images I would be able to assist you further.*, among others. Sometimes, GPT-4 inaccurately focused on describing the function of the buggy component instead of the component image itself, leading to erroneously high similarity scores for components not

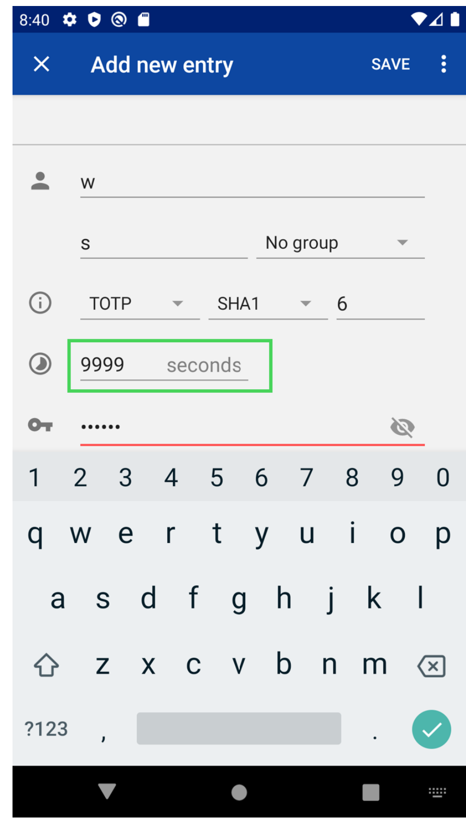

Figure 6.5: Buggy screen of a bug from Aegis App [5]

related to the bug, particularly when dealing with small-sized component images such as icons. For example, for a bug report from Aegis app [5], the OB is *"If code refresh time is set to 9999 - Aegis crashes"*, and the buggy screen Figure 6.5 has 20 components. GPT-4 misinterprets 12 out of 20 components, those 12 unrelated components are described as a component showing a numeric input field with 9999 entered, which is marked by a green square in Figure 6.5–the actual buggy component. Moreover, one component got all responses like *"I'm sorry but I cannot assist with requests that involve processing or analyzing specific images for content or context"*, further highlighting its inconsistency. In contrast, OpenAI-txt-embed model, leveraging direct metadata extracted from XML, easily identifies the faulty component by associating them with relevant information like "EditText" and "9999," thus proving to be more effective in locating buggy components.

Notably, SBert achieves a slightly lower MRR of 0.542, trailing GPT-4 by a slim margin of 3%. Moreover, it remains consistently strong across Hit Rates, achieving an H@5 score of 0.773, which is also comparable with 0.799 H@5 of GPT-4. These results indicate that text-only models are effective at suggesting buggy UI components based on textual descriptions compared with multimodal models such as Clip and Blip, which are at 0.416 and 0.432 MRR, respectively.

Additionally, as the results show, Blip's fine-tuned version (SBert_{ft}) is the most effective of all fine-tuned approaches, achieving a 0.501 MRR with a relative improvement of 16% of Blip. However, the fine-tuned versions of the SBert and Clip models (SBert_{ft} and Clip_{ft}) did not show the expected improvements over their respective non-fine-tuned versions, even though Clip_{ft} achieves a slightly higher H@3 than Clip (0.493 and 0.48, repestively), which indicates there is still room for improving the quality of the synthetic dataset. This issue is further discussed in the Section 6.3.6.4

Several observations can be derived from these results. First, the superiority of the DL models compared to Lucene suggests that DL models are better for component localization. Second, the textual information present in the UI components of the screens seems to be highly effective in performing localization, as indicated by OpenAI-txt-embed and SBert results. Third, Blip's superiority over Clip seems to stem from architectural differences, as Blip is specifically designed for text-image matching (via two losses: contrastive and matching), unlike Clip, which aims to learn joint representation for text and images via contrastive learning without an explicit matching loss. Fourth, while it may be counter-intuitive that text-only models outperform the multi-modal approaches, we generally observed that OBs tend to describe the buggy components using a language that is more similar to the component text observed by the user, which a text embedding model like OpenAI-txt-embed is specifically designed for. While Blip also leverages textual information from components, it does so based on the pixel data mainly, rather than the actual component text extracted from the UI metadata. Finally, despite GPT-4's exceptional performance in image and natural language understanding, the inherent

Table 6.6: Component Localization (CL) Results

Approach	MRR	MAP	H@1	H@2	H@3	H@4	H@5
SBert	0.542	0.527	0.367	0.546	0.633	0.729	0.773
SBert_{ft}	0.415	0.406	0.253	0.393	0.502	0.555	0.590
Clip	0.416	0.403	0.249	0.384	0.480	0.581	0.629
Clip_{ft}	0.399	0.381	0.227	0.380	0.493	0.533	0.594
Blip	0.432	0.412	0.258	0.419	0.511	0.572	0.616
Blip_{ft}	0.501	0.472	0.349	0.472	0.546	0.629	0.703
Lucene	0.416	0.374	0.323	0.459	0.502	0.528	0.537
OpenAI-txt-embed	0.622	0.603	0.467	0.664	0.716	0.755	0.790
GPT-4	0.559	0.528	0.384	0.550	0.672	0.725	0.799

instability of generative models limits its effectiveness in the CL task. This observation underscores that for component OBs, text-based models may be more appropriate. This finding further reinforces our conclusion.

6.3.6.3 RQ_3: Results vs. query qualities & retrieval difficulties

Query Quality. Figure 6.6 shows the screen localization results (based on MRR) across different query quality ratings (from 1 to 5, 5 meaning most informative). Notably, GPT-4 achieves the highest MRR on the least informative queries (*i.e.*, rating 1) among all models, a comparable performance with the most informative queries (*i.e.*, rating 5). This situation indicates that GPT-4 is robust across various levels of query quality. Additionally, the figure shows that while different approaches perform differently across the quality ratings, all models achieve the best performance for the most informative queries (*i.e.*, rating 5) except Clip. Moreover, the performance trend is similar for all the models on the queries with quality ratings 4 & 5 besides OpenAI-txt-embed.

Figure 6.7 shows the component localization results (based on MRR) across different query quality ratings. The figure shows a general upward trend: most models (besides GPT-4 and Clip) tend to perform better for higher-quality queries (rating 5) than lower-quality queries (rating 1 & 2). Interestingly, those models do not display a steady increase

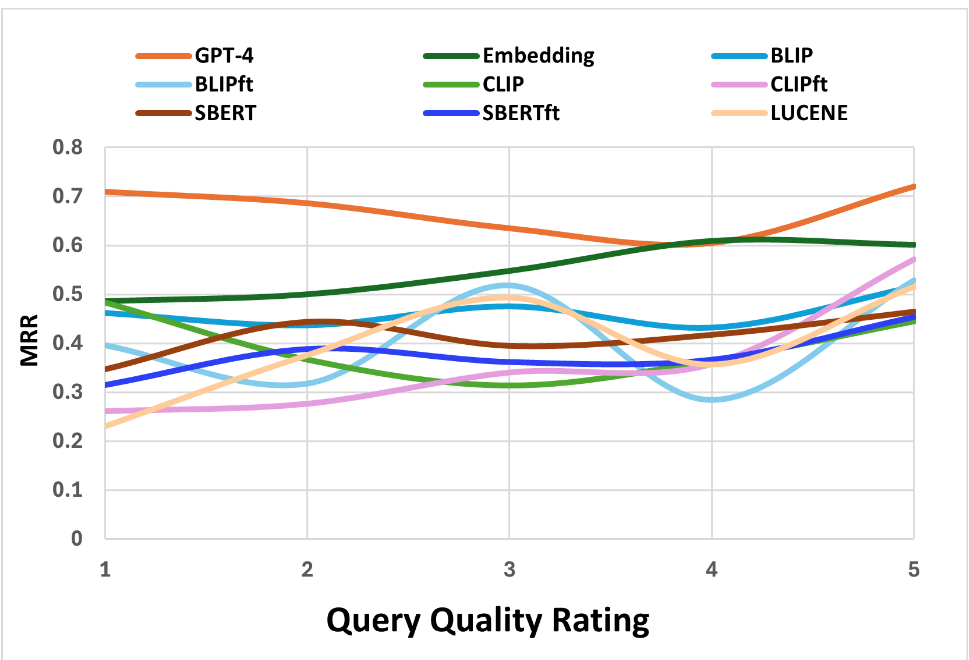

Figure 6.6: SL results for different query quality levels

in MRR with the improvement of query quality; instead, there's a notable peak at a quality rating of 3, followed by a decline and then an eventual ascent, particularly obvious for OpenAI-txt-embed which shows the highest MRR at rating of 3. Notably, GPT-4 shows a fluctuating pattern, peaking at a quality rating of 2, which is an unusual trend compared to others like Lucene, which appears to increase in performance with higher quality queries. Lucene mostly fails to retrieve any buggy components for the queries with ratings 1 & 2 (43/66 cases). Of the 17 queries with a rating of 1, Lucene fails to retrieve any component for 14 queries. For the remaining 3 queries, it cannot retrieve any relevant component resulting in a 0 MRR.

For SL, we found a middle to high positive correlation between the OB quality and the MRR results: a Spearman's correlation of 0.7 to 1 across all models except GPT-4, Blip (0.3 for both zero-shot and fine-tuned) and Clip. For CL, we found a high correlation: Spearman's correlation of 0.7 to 1 across all models except GPT-4 and Clip. The results show that the models tend to perform better for higher-quality queries than lower-quality

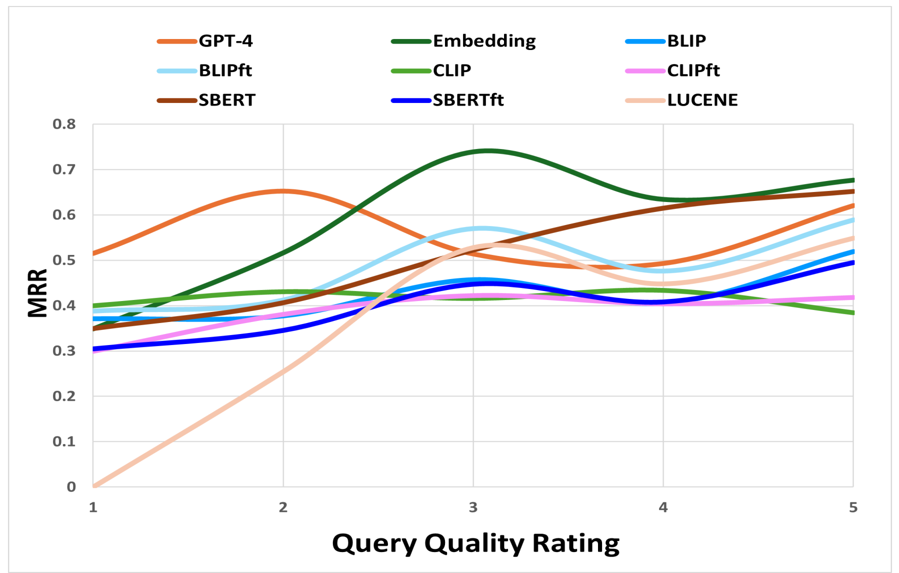

Figure 6.7: CL results for different query quality levels

queries for both SL & CL. Our replication package contains the # of queries per quality ratings [37].

Retrieval Difficulty. Figures 6.8 and 6.9 show the retrieval results (based on MRR) for easy- and hard-to-retrieve, for SL and CL respectively. For SL, all models perform higher on easy-to-retrieve tasks, which is expected. The same results are found for CL, except for BLIP, CLIP (zero-shot and fine-tuned), and SBERT$_{ft}$. Regardless of the difficulty of the tasks, GPT-4 performs highest for SR, and OPENAI-TXT-EMBED performs highest for CL. To recap, the results suggest a correlation between the difficulty of retrieval by humans and the retrieval performance of the models: they tend to perform higher/lower for easier/harder cases.

6.3.6.4 Discussion

SL *vs.* CL Our analysis revealed discrepancies in performance between SL (with MRR ranging from 0.351 to 0.661) and CL (showing MRR from 0.399 to 0.622). Furthermore,

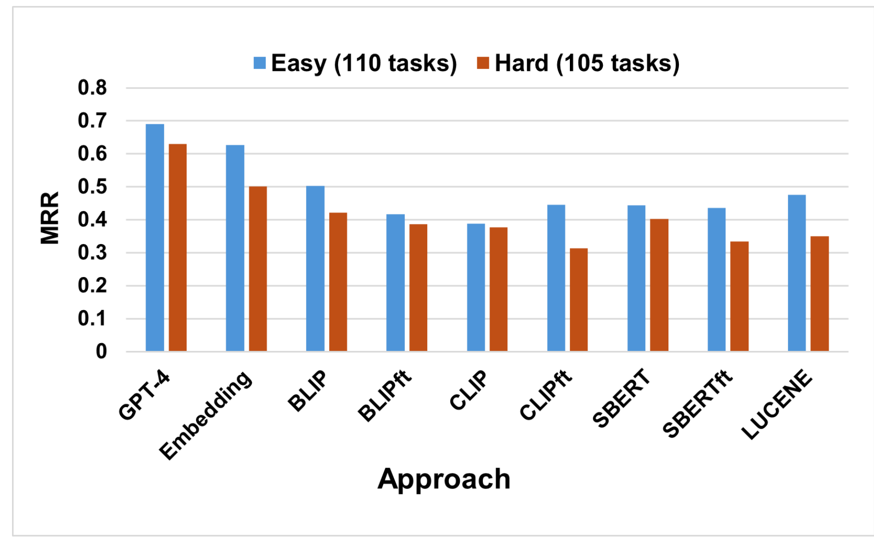

Figure 6.8: SL results for easy- and hard-to-retrieve tasks

excluding the two best-performing models: GPT-4 and OpenAI-txt-embed, the MRR values adjust to 0.351-0.438 and 0.399-0.542 for CL. Several factors make SL more challenging than CL. First, the corpus size is larger for SL than for CR (25.97 screens per app vs. 17.11 components per screen on avg.). Second, SL is more abstract or general than CL as the scope of SL is broader (all screens of the application vs. all components of a screen). Additionally, OBs are generally written focusing on the component level as the user interacts with the component while reproducing the bug. Third, the quality of the OBs has an important impact on the results. For instance, *"The color is unset."* is an OB with a quality rating of 2, which is from the bug report [4] in GnuCash Android app [21]. The best SL model, GPT-4, identified the relevant screen (shown in Figure 6.10) for this OB in the 21st place. However, the best CL model, OpenAI-txt-embed, identified the relevant component in the 1st place. However, the exceptional capabilities of GPT-4 and OpenAI-txt-embed help to bridge the gap. For example, GPT-4 accurately pinpoints the relevant screen by recognizing that *each account entry has a color strip which suggests that colors are used to distinguish or categorize accounts. If the color is unset it would likely affect these color strips*, leading to the conclusion that *it is the correct context where*

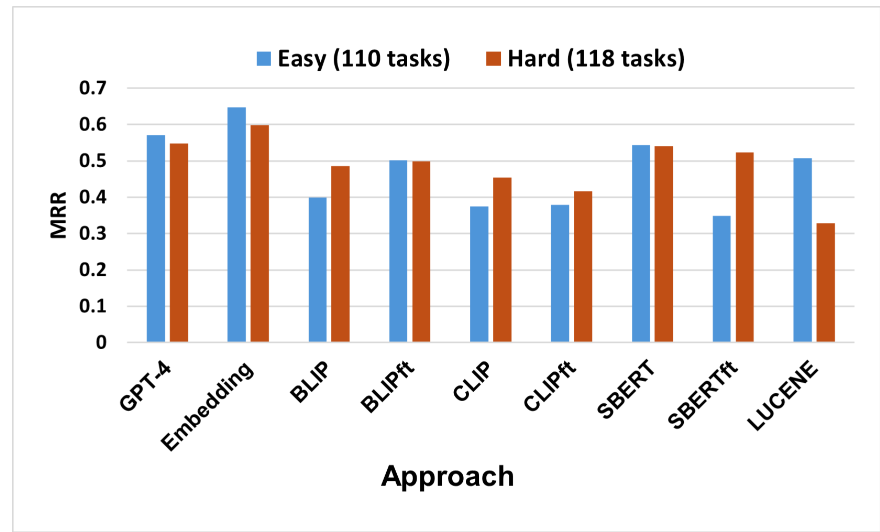

Figure 6.9: CL results for easy- and hard-to-retrieve tasks

such a bug could manifest, and thereby improving BLIP's rank to the 10th rank. GPT-4 could not rank it higher because GPT-4 observed *the bug is not actively visible*, despite being instructed to overlook whether the bug is actively occurring in the prompt.

Zero-shot *vs* fine-tuned models. For SL, CLIP_{ft} achieves a better performance than their zero-shot model. For CL, BLIP_{ft} achieves a better performance than their zero-shot model. Despite expectations, this improvement in performance with fine-tuning was not observed across all models. The anticipation was that fine-tuned models, at least for CL, would show enhanced performance because the synthetic dataset mainly contains component OBs with few screen OBs, which should make the models adequately learn the relationships between component OBs and images for CL. The primary reason behind this lies in the training dataset, especially caused by non-negligible discrepancies between synthetic OBs including both component and screen OBs. One possible reason is that our generated synthetic OBs are limited in their wording variety and fail to capture the broader spectrum of expressions found in actual bug reports. Moreover, despite efforts to cover a wide range of bug types, discourse patterns, and variable templates in the creation of synthetic OBs, they fall short of replicating the true complexity found in real bug

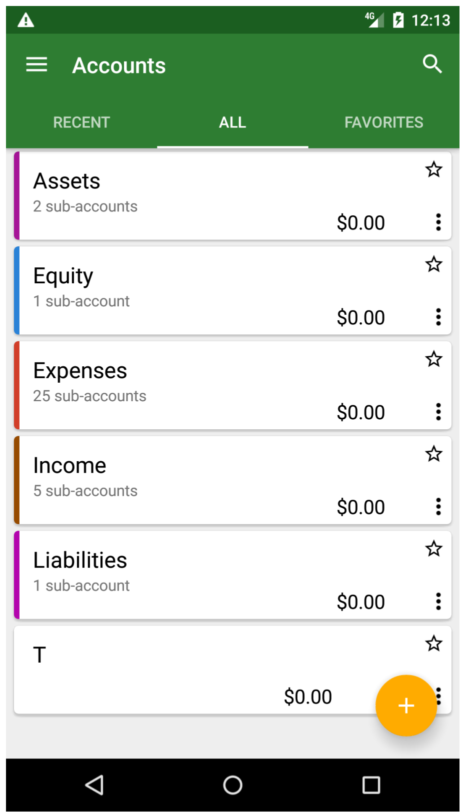

Figure 6.10: Buggy screen of a bug from GnuCash App [4]

reports. Additionally, our synthetic OBs tend to focus more on cosmetic issues, whereas real datasets encompass a broader array of bug types, including crash and navigation bugs. Consequently, there's a significant gap between the synthetic OBs, encompassing both component and screen OBs, and their real-world counterparts, highlighting an area for future enhancement of the synthetic dataset.

Textual *vs* multi-modal models. While we found that both textual and textual-visual models achieve a reasonable performance for Buggy UI Localization, no single type of model seems to stand out. GPT-4 and OpenAI-txt-embed were the best-performing approaches, yet no single model was the best for both tasks. The results indicate that both types of information, textual and visual, can be leveraged for Buggy UI Localization, yet textual data seems to be more useful for CL, while visual data seems to be useful for

SL.

Design requirements for Buggy UI Localization approaches. The results suggest that both textual and visual information alone are helpful for Buggy UI Localization. However, there is still room for improving the localization performance and specialized models may be required for this. We believe that both visual and textual information of the UI screens should be blended to build a more sophisticated model to increase localization performance. Other sources that can be explored are UI hierarchy information, which has shown promising results for command/instruction UI grounding [149, 151]. Moreover, for a successful localization approach, we may require potentially distinct models for each task: SL & CL.

Finally, while our study showed that it is feasible to leverage the pre-trained models for Buggy UI Localization, fine-tuning may be required to increase the performance of these models. However, creating or obtaining a comprehensive dataset for model fine-tuning is challenging because it should include OB descriptions of different types of bugs and wordings similarly found in real bug reports, with corresponding ground truth data. At the same time, such a dataset should include a variety of mobile apps and should be sufficiently large for the models to effectively learn patterns from the data. At the same time, creating a global model that applies to any mobile app and bug description is challenging. Future work should explore the possibility of comparing global *vs.* local models that work for specific apps. However, this brings an additional challenge: collecting sufficiently large ground truth data for individual apps.

6.4 Study 2: Improving Bug Localization

To illustrate the practical usefulness of automated Buggy UI Localization, we conducted an additional study that investigates how identified buggy UI screens from Blip, our best-performing SL model, can improve traditional bug localization approaches. We aim to answer the following research question:

RQ$_4$: *Can the identified buggy UI screens by* Blip *lead to improved bug localization performance?*

To answer this RQ, we adapted the approach proposed by Mahmud *et al.* [156] (section 6.4.1) as an end-to-end automated bug localization technique (section 6.4.2), which retrieves potentially buggy code files by leveraging the information from Blip's suggested buggy UI screens, given a bug description. We defined different pipelines that combine Buggy UI Localization and traditional Bug Localization to source code (section 6.4.2) and compared their performance with baseline techniques that do not use UI information (section 6.4.3).

6.4.1 UI-based Bug Localization in Code

Mahmud *et al.* [156] demonstrated that mobile app UI interaction data can improve the performance of four IR-based bug localizers that rely on bug reports (*e.g.*, BugLocator [234]). Their approach consists of modifying the initial ranking of potentially buggy code files produced by a bug localizer for a given bug report, by **boosting** relevant files and/or **filtering** out irrelevant files, or by performing **reformulation**. These operations (*a.k.a.* augmentations) leverage information extracted from the UI screen that shows the reported bug and the preceding 1-3 screens in a bug reproduction trace.

The information extracted from UI screens is *UI terms*, such as the activity and window names of the screens, which are then matched against code file names to produce a set of *UI-related files*. The UI terms and UI-related files are used by two augmentation methods: (1) **Reformulating queries:** Two query reformulation strategies were defined based on UI terms: query expansion by appending UI terms to bug reports, and query replacement by using UI terms as the query; and (2) **File re-ranking:** three re-ranking strategies were defined: filtering, boosting, and a combination of the two. Filtering involves removing files that do not match UI-related files from the search corpus. Boosting elevates the ranking of files in the search corpus that match UI-related files during the search. The filtering and boosting strategy combines both techniques: ranking boosts files higher in the filtered

file corpus.

To re-rank files in the search corpus, Mahmud *et al.* extracted three types of UI-related files using UI terms: (1) **Mapping UI screen terms to files:** the screen activity and window names were extracted from screen metadata and code files were matched against the activity/window terms; (2) **Mapping UI component terms to files:** components in UI screens were mapped to files by finding references of the IDs associated with those components within the file content; and (3) **mapping exercised UI component terms to files:** the components of the screens that were exercised by the user during bug reproduction were mapped to files by finding references of the IDs associated with those components within the file content.

Mahmud *et al.* employed four main configuration parameters to integrate UI information into the IR bug localizers: (1) the number of UI screens in a reproduction trace, specifically the last 2-4 screens, including the buggy one; (2) five types of UI information sources (*e.g.*, UI screen, exercised UI components) (3) query reformulation strategies; (4) re-ranking strategies. In total, 657 configurations were defined and evaluated for each bug localizer.

6.4.2 Using Buggy UIs for Bug Localization

Mahmud *et al.*'s approach [156] requires as input a trace of the UI screens and components that the user interacted with to reproduce the bug described in a bug report. The trace and the buggy UI screen in the trace are meant to be manually collected/identified by the developer. Mahmud *et al.*'s approach then uses the metadata information from the buggy screen and the 3-4 prior screens/components in the reproduction scenario as input to their augmentation approaches that filter and boost potentially buggy code files retrieved by an IR-based bug localization technique.

Our goal is to eliminate the manual effort of Mahmud *et al.*'s approach and define a fully automated end-to-end pipeline of bug localization in code that leverages the buggy UI screens recommended by a screen localizer. To that end, we adapted Mahmud *et*

al.'s approach by using our best screen localization approach (*i.e.*, GPT-4) to suggest the top 3-4 buggy UI screens as the only source of information needed by the bug localization pipeline. In this way, no reproduction scenario and buggy screen need to be collected/identified by the developers, thus eliminating the manual effort.

As such, we defined an approach that integrates both the screen localization and bug localization pipelines since the ultimate goal is to produce a ranked list of potentially buggy code files for a given bug report. The challenge in defining this combined approach is that a bug report can contain multiple OB descriptions. If we execute GPT-4 on each OB description, it would produce multiple lists of potentially buggy UI screens. Therefore, this challenge is to decide which buggy screens should be given as input to the localization pipeline, to produce a single ranking of buggy files.

To address this problem, we considered two options: (1) produce and provide a single ranking of UI screens for the bug report, or (2) provide each ranking of UI screens (for each OB description in the bug report) to the bug localization pipeline, to produce multiple code file ranking, and then combine these rankings into a final code file ranking. For option #1, we explored two strategies: (i) concatenate the OB descriptions in a bug report and use the resulting query as input to GPT-4, and (ii) select only the first OB description found in the bug report as a query to GPT-4. These two strategies (which we call "CONCAT OBS" and "FIRST OB", respectively) would produce a single UI screen ranking, which can be used by the bug localization pipeline to suggest a single code file ranking for the bug report. As for option #2, to produce a single code file ranking, we first averaged the similarity scores of each code file found in all the buggy file rankings to produce a single similarity score for the file. Then, these combined similarities, for all the files in the rankings, are used to produce a final code file ranking (*i.e.*, sorting by these similarities). We call this strategy "INDIVIDUAL OB".

6.4.3 Approach Execution, Dataset, and Metrics

We selected the two best IR-based bug localization techniques from Mahmud *et al.*'s study [156], namely Lucene [115] (adapted for the bug localization task) and BugLocator [234], and executed them in our combined pipeline for bug localization. We experimented with all 70 feasible configurations comprising the different augmentation methods and UI information defined in the prior work. We also experimented by proving the top 3 and 4 buggy UI screens suggested by Blip, following the best number of screens found by Mahmud *et al.*, for Lucene (3 screens) and BugLocator (4 screens).

We executed the three combined pipelines defined above (*i.e.*, Concat OBs, First OB, and Individual OB) using both bug localizers. However, we could not execute Individual OB with BugLocator because the tool provided by the original authors [234] does not provide the code file rankings, which are needed by Individual OB. The pipelines were executed on 79 of the 80 bug reports from the bug localization benchmark provided by Mahmud *et al.* [156]. As mentioned in section 6.3.3.1, we excluded one bug report because we could not reproduce the bug. We did not use the full set of 87 bug reports used in the Buggy UI Localization because 8 bugs do not have any Java files as the ground truth for bug localization.

The performance of the combined pipelines, using all possible configurations and IR bug localizers, was measured and compared using Hits@k and its relative improvement (RI), in line with the methodology followed by Mahmud *et al.* [156]. We used as baselines the original IR bug localizers, without using any UI information. We aim to test if the combined bug localization pipelines outperform the baselines. If so, we can conclude that automated Buggy UI Localization is useful to improve bug localization in source code. Note that, for the experiments with 4 screens, we used 77 bug reports as 2 bug reports have only 3 screens in the SL corpus.

6.4.4 Results

Table 6.7 shows the bug localization results for both IR bug localizers and the best configurations we obtained among all configurations. These results are obtained when GPT-4 suggests the top 3 and 4 buggy screens. Complete results, obtained from all configurations and experiments we conducted, are found in our replication package [37].

For each pipeline, IR bug localizer, and number of buggy screens recommended by GPT-4, we consistently found that the best configuration (*i.e.*, the highest H@10 improvement compared to the baselines) is when filtering with GUI Screen Components (SC), boosting with GUI Screen (GS), and an optional query expansion with GS (for both Lucene and BugLocator). Like Mahmud *et al.* [156], we obtained the best results with 3 screens for Lucene and 4 screens for BugLocator.

Table 6.7 reveals that all the combined pipelines for bug localization lead to performance improvement compared to the baselines, by 4.77% to 16.69% H@10. When using Lucene, Concat OBs pipeline achieves the highest performance boost of 11.17%. Likewise, with BugLocator, Concat OBs also records the best performance gain at 16.69%, which translates into retrieving the buggy code files in the top-10 results for seven more bug reports, compared to the baseline. Moreover, when using BugLocator with Concat OBs pipeline, there is a 16.69% increase in H@10, equivalent to 9 more successful retrieval tasks.

We compare our results from table 6.7 with the results achieved by the best configurations obtained for Lucene and BugLocator by Mahmud *et al.* [156], since those results represent a perfect identification of the buggy UI screen, along with the reproduction scenario. However, we must cautiously compare these results since the bug reports used in both studies are not exactly the same. Significantly, our best configurations show the comparable performance with the manual bug localization approach by Mahmud *et al.* [156] (0.89 vs 0.9 H@10 for Lucene, and 0.82 vs 0.84 for BugLocator). These results demonstrate the effectiveness of our fully automated way of localizing buggy code

files via Buggy UI Localization, eliminating the need for need manual effort in collecting reproduction traces and the buggy screen that the prior work requires.

Given the results, we conclude that Buggy UI Localization can be useful to improve the performance of UI-based bug localization in source code in a fully automated end-to-end way.

Table 6.7: Bug Loc. Performance via Buggy UI Localization

Bug Localizer	Approach	# Screens	H@5	H@10	RI of H@10	#Bug Top10
Lucene	Baseline	3	0.75	0.80	-	63
	Concat OBs	3	0.80	0.89	11.17%	70
	First OB	3	0.77	0.87	9.54%	69
	Individual OB	3	0.73	0.84	4.77%	67
BugLocator	Baseline	4	0.58	0.70	-	54
	Concat OBs	4	0.73	0.82	16.69%	63
	First OB	4	0.69	0.79	12.98%	61

6.5 Threats to Validity

Construct Validity. There may be subjectivity introduced in the dataset construction when identifying the OB descriptions in the bug reports, their quality rating, retrieval difficulty levels, and the ground truth buggy screens/components. We mitigated this threat by adopting a rigorous methodology to label and curate the data during joint sessions of bug understanding, replication, and analysis among four researchers, reaching consensus in all cases. **Internal Validity.** The selection of models affects the internal validity of our results/conclusions. To mitigate this we covered both uni-modal (SBert and OpenAI-txt-embed) and multi-modal (Clip, Blip and GPT-4) DL models for automating Buggy UI Localization. For buggy code localization study, we conducted various experiments with all feasible configurations on two localizers (Lucene and BugLocator) to obtain the best-performing configuration. **External Validity.** The conclusions of our study may not generalize to other retrieval models, bug descriptions, and apps. To improve the

generalization, we selected different types of models and built a real dataset containing a variety of bug types, and apps that implement different GUIs for multiple domains.

6.6 Related Work

Mobile App Bug Report Management. Recent research [106, 195, 104, 226, 107] has explored the use of bug reports of mobile apps to automate various bug report management tasks. Researchers [231, 226] have proposed approaches to reproduce Android bugs/crashes or generate test cases based on bug reports. However, these techniques lack verification that the generated reproduction steps can trigger the reported bugs. A Buggy UI Localization approach that identifies the buggy screen/components can help generate assertions to this end. Song *et al.* [195] proposed a chatbot to help end-users report Android bugs via visual guidance and automated quality verification of bug descriptions. This chatbot can benefit from a Buggy UI Localization approach by accurately assessing how the description corresponds or relates to UI screens/components. Despite the growing body of research on automating bug report management tasks (*e.g.*, bug reporting [195, 104], reproduction [231, 106, 226, 231, 230], localization [53, 212, 140, 108, 79, 81, 83], and others [107, 233]), prior work has not explored how to automatically localize buggy UIs as we do.

UI Representation Learning and Applications. UI representation learning aims to represent UI elements or text via embeddings [147, 119, 61, 151] for downstream tasks such as image captioning [206, 165, 90] and UI component labeling [89, 150, 90]. One application of UI representation learning is mapping (*a.k.a.* grounding) textual instructions to UI action/elements [173, 149, 217]. Pasupat *et al.* [173] evaluated three models to ground natural language commands to web elements. Li *et al.* [149] utilized transformers models for this task, based on three synthetic datasets for training. Although this grounding task may appear similar to Buggy UI Retrieval, there are significant differences that make it difficult to adapt those models to our problem. For example, Li *et al.*'s approach [149]

requires a sequence of screens where the instructions are performed, and then locating the corresponding UI component for each instruction. In contrast, our work focuses on identifying the buggy UI screens and components without any prior information about which screens are relevant. Furthermore, our study deals with bug descriptions, whose language is considerably more complex than that of UI instructions [86].

6.7 Conclusions

This work reported the results of the first empirical study that investigated the effectiveness of textual/visual neural models for automatically localizing buggy UI screens and components from the bug descriptions of mobile apps. We evaluated the approaches for screen and component retrieval, using a large-scale synthetic dataset and a real-life dataset of manually-curated OB descriptions and ground truth UI screens/components.

The study revealed that the best-performing approaches can suggest correct UI screens and components in the top-3 recommendations for 73% and 71.6% of the bug descriptions, respectively. Our findings suggest that there is potential for improvement through more refined model fine-tuning with enhanced synthetic datasets, and the models tend to perform better for higher-quality bug descriptions. We also showed that Buggy UI Localization can be useful to automate and improve buggy code localization approaches.

Chapter 7

Conclusion and Future Work

This dissertation discusses significant challenges in bug management across three areas: **Bug Reporting**, **Bug Assignment**, and **Bug Localization**. Traditional bug reporting is manual, inefficient, and prone to errors, leading to low-quality reports that hinder the resolution process. Even though there has been advancement in automated bug reporting systems, challenges remain, notably their complexity and a lack of interactive support. In the realm of bug assignment within complex software systems, manual methods are error-prone and automated techniques using machine learning or information retrieval face challenges in catering to the unique characteristics of individual reports, affecting their effectiveness. Furthermore, the process of bug localization, particularly in mobile applications, faces significant challenges due to a semantic gap between bug reports and source code, especially when identifying buggy UI elements, highlighting the need for more sophisticated bug localization methods that consider graphical user interface information.

The dissertation aims to address these challenges by exploring methods to automate the bug reporting process, assessing the feasibility and efficiency of automated bug assignment techniques, and investigating ways to enhance bug localization through integration with UI data. By addressing the inefficiencies and inaccuracies inherent in current practices, this dissertation seeks to advance the understanding, design, and application of automated systems, ultimately improving software development efficiency and bug resolution

effectiveness.

7.1 Main Contributions

Our research contributions can be summarized as follows:

1. We proposed and evaluated Bee, an innovative tool that enhances GitHub's issue tracker capabilities. It provides vital feedback to reporters and developers about OB, EB, and S2R in bug reports. Utilizing machine learning models, Bee analyzes submitted issues, distinguishing bug reports from enhancement suggestions or questions. The results reveal that Bee can identify sentences describing OB, EB, and S2R and detect the omission of these elements, and inform reporters about missing elements so that they can provide the information timely.
2. We proposed and evaluated Burt, a novel interactive Android app bug-reporting system, designed to provide real-time feedback for each element of a bug description, guide corrections where needed, and bridge the knowledge gap between end-users and developers. The proposed system has been evaluated empirically and has been found to improve the quality of bug reports. This work paves the way for a new approach to end-user bug reporting, transitioning from static to interactive bug reporting systems, and is expected to serve as a foundation for a new generation of interactive bug reporting systems.
3. We proposed and evaluated Mix, a novel approach that can automatically recommend and apply the best-performing approach on individual bug reports via machine learning (ML). Our results revealed that while these models perform comparably to baseline approaches, they fall short of achieving maximum potential effectiveness, indicating the selected features might not sufficiently differentiate between various assignment strategies. Our result suggests a potential gap in how bug assignment methods account for real-world considerations, such as developer availability, which could explain the observed performance limitations. This highlights the need for

research focused on improving feature selection for assignment strategies, understanding actual bug assignment practices, and enhancing automated systems with more comprehensive information for accurate bug assignment.

4. We proposed and evaluated a fully automated end-to-end bug localization approach for GUI-related bugs of mobile applications by automating identifying the UI screens and components that cause and/or show a reported issue from textual bug descriptions(Buggy UI Localization). For Buggy UI Localization, we explore the application of textual and multi-modal (visual-textual) deep learning or large language model (LLM) techniques to Buggy UI Localization. Our results demonstrate the effectiveness of DL models, particularly LLMs, for automatically localizing buggy UI screens and components from the bug descriptions of mobile apps, while also highlighting Buggy UI Localization can be useful to automate and improve buggy code localization approaches.

7.2 The Vision for Future Work

The vision for future work is to develop an AI-powered bug tracking system. The advancements in AI, especially in Large Language Models (LLMs) and AI agents, present an opportunity to transform traditional bug tracking systems. This system will feature intelligent bug detection capabilities that are customized to support various software systems, significantly reducing the manual effort required in identifying issues. Moreover, it will provide a more user-friendly interface to improve accessibility for developers of all skill levels. For example, it will allow users to customize the workflows. The system would also support automated detection of duplicate bugs, as well as automated bug reproduction and fixing. Additionally, the proposed system will integrate seamlessly with popular development and project management tools, such as Slack, facilitating better communication and workflow integration. This integration will not only streamline the bug tracking process but also enhance collaboration across development teams.

In conclusion, this dissertation achieves important milestones towards improving the automated bug report management process, specifically targeting **Bug Reporting**, **Bug Assignment**, and **Bug Localization**. These four works illustrate the dissertation's substantial contributions to the field of automated bug report management. Furthermore, it points out the challenges and potential directions for future research, underlining the ongoing need for innovation in automated bug report management.

Bibliography

[1] Bug report #95598 from eclipse submitted on github. https://github.com/ysong10/fast_ALS/issues/645, 2005.

[2] Original bug report #95598 from eclipse. https://bugs.eclipse.org/bugs/show_bug.cgi?id=95598, 2005.

[3] An open letter to github from the maintainers of open source projects. Available online: https://github.com/dear-github/dear-github, 2016.

[4] Gnucash's bug report #620. https://tinyurl.com/y3pw69ac, 2019.

[5] Aegis's bug report #500. https://github.com/beemdevelopment/Aegis/issues/500, 2020.

[6] Default labels on github issues. https://help.github.com/en/github/managing-your-work-on-github/about-labels, 2020.

[7] Focus-androi's bug report #3152. https://github.com/mozilla-mobile/focus-android/issues/3152, 2020.

[8] Github developer: Using the github api in your app. https://developer.github.com/apps/quickstart-guides/using-the-github-api-in-your-app/, 2020.

[9] Issue-label bot. https://github.com/marketplace/issue-label-bot, 2020.

[10] Replication package of Bee's evaluation. https://github.com/sea-lab-wm/bee-tool, 2020.

[11] 's installation website. https://github.com/apps/bee-tool, 2020.

[12] Android token. https://f-droid.org/en/packages/uk.co.bitethebullet.android.token/, 2021.

[13] Android's layout and layout validation. https://developer.android.com/studio/debug/layout-inspector, 2021.

[14] Antennapod. https://play.google.com/store/apps/details?id=de.danoeh.antennapod&hl=en_US&gl=US, 2021.

[15] App store: Ratings, reviews, and responses. https://developer.apple.com/app-store/ratings-and-reviews/, 2021.

[16] Appetize.io. https://appetize.io/, 2021.

[17] Bugzilla issue tracker - https://bugzilla.mozilla.org, 2021.

[18] Cohen's kappa. https://en.wikipedia.org/wiki/Cohen%27s_kappa, 2021.

[19] Droid weight. https://fossdroid.com/a/droidweight.html, 2021.

[20] Github issue tracker - https://github.com/features, 2021.

[21] Gnucash. https://play.google.com/store/apps/details?id=org.gnucash.android&hl=en_US&gl=US, 2021.

[22] Growtracker. https://f-droid.org/en/packages/me.anon.grow/, 2021.

[23] Images2pdf. https://play.google.com/store/apps/details?id=imagetopdf.pdfconverter.jpgtopdf.pdfeditor&hl=en_US&gl=US, 2021.

[24] Jira bug reporting system - https://www.atlassian.com/software/jira, 2021.

[25] K-9. https://play.google.com/store/apps/details?id=com.fsck.k9&hl=en_US&gl=US, 2021.

[26] Microsoft azure chatbot architecture. https://docs.microsoft.com/en-us/azure/architecture/reference-architectures/ai/conversational-bot, 2021.

[27] Mileage. https://fossdroid.com/a/mileage.html, 2021.

[28] React chatbot kit. https://fredrikoseberg.github.io/react-chatbot-kit-docs/, 2021.

[29] Report an issue or send feedback on chrome. https://support.google.com/chrome/answer/95315, 2021.

[30] Spring boot. https://spring.io/projects/spring-boot, 2021.

[31] Stanford dependencies. https://nlp.stanford.edu/software/stanford-dependencies.html, 2021.

[32] A time tracker. https://f-droid.org/en/packages/com.markuspage.android.atimetracker/, 2021.

[33] https://www.bugsee.com, 2021.

[34] Write a review on google play. https://support.google.com/googleplay/answer/4346705, 2021.

[35] Online replication package. 2022.

[36] Qualtrics: Survey software. https://www.qualtrics.com/core-xm/survey-software/, 2022.

[37] replication package. https://anonymous.4open.science/r/bug_report_mapping_anonymized, 2023.

[38] Rico dataset. https://interactionmining.org/rico, 2023.

[39] Wifi analyzer. https://github.com/VREMSoftwareDevelopment/WiFiAnalyzer, 2023.

[40] Wifi analyzer's issue #191. https://github.com/VREMSoftwareDevelopment/WiFiAnalyzer/issues/191, 2023.

[41] bugherd bug reporting system - https://bugherd.com, 2024.

[42] bugsnag bug reporting system - https://www.bugsnag.com/?utm_source=aw&utm_medium=ppcg&utm_campaign=SEM_Bugsnag_PR_NA_ENG_EXT_Prospecting&utm_term=bugsnag&utm_content=536025543533&&gclsrc=aw.ds&&campaignid=14051841305&adgroupid=128124629231&adid=536025543533&gad_source=1&gclid=Cj0KCQjw-_mvBhDwARIsAA-Q0Q5aNk_ZGXKfF8Tp6rclKpGZPY2dk1vCuktVTQYRKv675SuUBvioN50aAnjIEALw_wcB&gclsrc=aw.ds, 2024.

[43] claude-3. https://www.anthropic.com/news/claude-3-family, 2024.

[44] clickup bug reporting system - https://clickup.com/templates/bug-report-t-2z6mm47, 2024.

[45] embrace bug reporting system - https://embrace.io/product/crash-reporting/, 2024.

[46] gemini. https://gemini.google.com/, 2024.

[47] gleap bug reporting system - https://www.gleap.io/in-app-bug-reporting, 2024.

[48] Gpt-4. https://openai.com/research/gpt-4, 2024.

[49] instabug bug reporting system - https://www.instabug.com/product/bug-reporting, 2024.

[50] Openai embedding models. `https://platform.openai.com/docs/guides/embeddings`, 2024.

[51] shake bug reporting system - `https://www.shakebugs.com/bug-reporting/`, 2024.

[52] Vishakha Agrawal, Yong-Han Lin, and Jinghui Cheng. Understanding the characteristics of visual contents in open source issue discussions: a case study of jupyter notebook. In *EASE'22*.

[53] Shayan A Akbar and Avinash C Kak. A large-scale comparative evaluation of ir-based tools for bug localization. In *MSR'20*.

[54] Ethem Utku Aktas and Cemal Yilmaz. Automated issue assignment: results and insights from an industrial case. *Empirical Software Engineering*, 25(5):3544–3589, 2020.

[55] Jean-Baptiste Alayrac, Jeff Donahue, Pauline Luc, Antoine Miech, Iain Barr, Yana Hasson, Karel Lenc, Arthur Mensch, Katie Millican, Malcolm Reynolds, Roman Ring, Eliza Rutherford, Serkan Cabi, Tengda Han, Zhitao Gong, Sina Samangooei, Marianne Monteiro, Jacob Menick, Sebastian Borgeaud, Andrew Brock, Aida Nematzadeh, Sahand Sharifzadeh, Mikolaj Binkowski, Ricardo Barreira, Oriol Vinyals, Andrew Zisserman, and Karen Simonyan. Flamingo: a visual language model for few-shot learning, 2022.

[56] Giuliano Antoniol, Kamel Ayari, Massimiliano Di Penta, Foutse Khomh, and Yann-Gaël Guéhéneuc. Is It a Bug or an Enhancement?: A Text-based Approach to Classify Change Requests. In *Proceedings of the Conference of the Center for Advanced Studies on Collaborative Research: Meeting of Minds*, pages 304–318, 2008.

[57] John Anvik, Lyndon Hiew, and Gail C Murphy. Coping with an open bug

repository. In *Proceedings of the 2005 OOPSLA workshop on Eclipse Technology eXchange (ETX'05)*, pages 35–39, 2005.

[58] John Anvik, Lyndon Hiew, and Gail C Murphy. Who should fix this bug? In *Proceedings of the 28th International Conference on Software Engineering (ICSE'06)*, pages 361–370, 2006.

[59] John Anvik and Gail C. Murphy. Reducing the effort of bug report triage: Recommenders for development-oriented decisions. *Transactions on Software Engineering and Methodologies (TOSEM)*, 20(3):10:1–10:35, 2011.

[60] Jorge Aranda and Gina Venolia. The Secret Life of Bugs: Going Past the Errors and Omissions in Software Repositories. In *Proceedings of the 31st International Conference on Software Engineering*, pages 298–308, 2009.

[61] Chongyang Bai, Xiaoxue Zang, Ying Xu, Srinivas Sunkara, Abhinav Rastogi, Jindong Chen, et al. Uibert: Learning generic multimodal representations for ui understanding. *arXiv preprint arXiv:2107.13731*, 2021.

[62] Alberto Baldrati, Marco Bertini, Tiberio Uricchio, and Alberto Del Bimbo. Conditioned and composed image retrieval combining and partially fine-tuning clip-based features. In *CVPR'22*.

[63] Olga Baysal, Michael W Godfrey, and Robin Cohen. A bug you like: A framework for automated assignment of bugs. In *Proceedings of IEEE 17th International Conference on Program Comprehension (ICPC'09)*, pages 297–298. IEEE, 2009.

[64] Carlos Bernal-Cárdenas, Nathan Cooper, Madeleine Havranek, Kevin Moran, Oscar Chaparro, Denys Poshyvanyk, and Andrian Marcus. Translating video recordings of complex mobile app ui gestures into replayable scenarios. *IEEE Transactions on Software Engineering*, page to appear, 2022.

[65] Carlos Bernal-Cárdenas, Nathan Cooper, Kevin Moran, Oscar Chaparro, Andrian Marcus, and Denys Poshyvanyk. Translating video recordings of mobile app usages into replayable scenarios. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE'20)*, pages 309–321, 2020.

[66] Nicolas Bettenburg, Rahul Premraj, Thomas Zimmermann, and Sunghun Kim. Duplicate bug reports considered harmful ... really? In *Proceedings of the International Conference on Software Maintenance (ICSM'08)*, pages 337–345, 2008.

[67] Nicolas Bettenburg, Rahul Premraj, Thomas Zimmermann, and Sunghun Kim. Extracting Structural Information from Bug Reports. In *Proceedings of the International Working Conference on Mining Software Repositories (WCRE'08)*, pages 27–30, 2008.

[68] Aaditya Bhatia, Shaowei Wang, Muhammad Asaduzzaman, and Ahmed E Hassan. A study of bug management using the stack exchange question and answering platform. *IEEE Transactions on Software Engineering*, 48(2):502–518, 2022.

[69] Pallab Bhattacharya, Liudmila Ulanova, Iulian Neamtiu, and Sai Charan Koduru. An empirical analysis of bug reports and bug fixing in open source android apps. In *Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on*, pages 133–143, 2013.

[70] Pamela Bhattacharya, Iulian Neamtiu, and Christian R. Shelton. Automated, highly-accurate, bug assignment using machine learning and tossing graphs. *Journal of Systems and Software*, 85(10):2275–2292, 2012.

[71] Christian Bird, David Pattison, Raissa D'Souza, Vladimir Filkov, and Premkumar Devanbu. Latent social structure in open source projects. In *Pro-*

ceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering (FSE'08), pages 24–35, 2008.

[72] T. F. Bissyandé, D. Lo, L. Jiang, L. Réveillère, J. Klein, and Y. L. Traon. Got issues? who cares about it? a large scale investigation of issue trackers from github. In *Proceedings of the 24th International Symposium on Software Reliability Engineering (ISSRE'13)*, pages 188–197, 2013.

[73] Gerald Bortis and Andre Van Der Hoek. Porchlight: A tag-based approach to bug triaging. In *Proceedings of the 35th International Conference on Software Engineering (ICSE'13)*, pages 342–351. IEEE, 2013.

[74] Silvia Breu, Rahul Premraj, Jonathan Sillito, and Thomas Zimmermann. Information Needs in Bug Reports: Improving Cooperation Between Developers and Users. In *Proceedings of the Conference on Computer Supported Cooperative Work (CSCW'10)*, pages 301–310, 2010.

[75] Gemma Catolino, Fabio Palomba, Andy Zaidman, and Filomena Ferrucci. Not all bugs are the same: Understanding, characterizing, and classifying bug types. *Journal of Systems and Software*, 152:165–181, 2019.

[76] Gavin C Cawley and Nicola LC Talbot. On over-fitting in model selection and subsequent selection bias in performance evaluation. *Journal of Machine Learning Research*, 11(Jul):2079–2107, 2010.

[77] Oscar Chaparro, Carlos Bernal-Cárdenas, Jing Lu, Kevin Moran, Andrian Marcus, Massimiliano Di Penta, Denys Poshyvanyk, and Vincent Ng. Assessing the quality of the steps to reproduce in bug reports. In *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'19)*, pages 86–96, 2019.

[78] Oscar Chaparro, Carlos Bernal-Cárdenas, Jing Lu, Kevin Moran, Andrian Marcus, Massimiliano Di Penta, Denys Poshyvanyk, and Vincent Ng. Assessing the quality of the steps to reproduce in bug reports. In *Proceedings of the 27th ACM Joint Meeting on the Foundations of Software Engineering (ESEC/FSE'19)*, pages 86–96, 2019.

[79] Oscar Chaparro, Juan Manuel Florez, and Andrian Marcus. Using bug descriptions to reformulate queries during text-retrieval-based bug localization. *ESE'19*.

[80] Oscar Chaparro, Juan Manuel Florez, and Andrian Marcus. Using observed behavior to reformulate queries during text retrieval-based bug localization. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME'17)*, pages 376–387. IEEE, 2017.

[81] Oscar Chaparro, Juan Manuel Florez, and Andrian Marcus. Using observed behavior to reformulate queries during text retrieval-based bug localization. In *Proceedings of the 33rd IEEE International Conference on Software Maintenance and Evolution (ICSME'17)*, pages 376–387, 2017.

[82] Oscar Chaparro, Juan Manuel Florez, and Andrian Marcus. Using bug descriptions to reformulate queries during text-retrieval-based bug localization. *Empirical Software Engineering*, 24(5):2947–3007, 2019.

[83] Oscar Chaparro, Juan Manuel Florez, Unnati Singh, and Andrian Marcus. Reformulating queries for duplicate bug report detection. In *SANER'19*.

[84] Oscar Chaparro, Juan Manuel Florez, Unnati Singh, and Andrian Marcus. Reformulating queries for duplicate bug report detection. In *Proceedings of the IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER'19)*, pages 218–229. IEEE, 2019.

[85] Oscar Chaparro, Jing Lu, Fiorella Zampetti, Laura Moreno, Massimiliano Di Penta, Andrian Marcus, Gabriele Bavota, and Vincent Ng. Detecting missing information in bug descriptions. In *Proceedings of the 11th ACM Joint Meeting on the Foundations of Software Engineering (ESEC/FSE'17)*, pages 396–407, 2017.

[86] Oscar Chaparro, Jing Lu, Fiorella Zampetti, Laura Moreno, Massimiliano Di Penta, Andrian Marcus, Gabriele Bavota, and Vincent Ng. Detecting missing information in bug descriptions. In *Proceedings of the 11th Joint Meeting on the Foundations of Software Engineering (ESEC/FSE'17)*, pages 396–407, 2017.

[87] Oscar Chaparro, Jing Lu, Fiorella Zampetti, Laura Moreno, Massimiliano Di Penta, Andrian Marcus, Gabriele Bavota, and Vincent Ng. Detecting missing information in bug descriptions. In *Proceedings of the 11th Joint Meeting on the Foundations of Software Engineering (ESEC/FSE'17)*, pages 396–407, 2017.

[88] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, 16:321–357, 2002.

[89] Jieshan Chen, Chunyang Chen, Zhenchang Xing, Xiwei Xu, Liming Zhu, Guoqiang Li, and Jinshui Wang. Unblind your apps: Predicting natural-language labels for mobile gui components by deep learning. In *ICSE'20*.

[90] Jieshan Chen, Amanda Swearngin, Jason Wu, Titus Barik, Jeffrey Nichols, and Xiaoyi Zhang. Towards complete icon labeling in mobile applications. In *CHI'22*.

[91] Jieshan Chen, Amanda Swearngin, Jason Wu, Titus Barik, Jeffrey

Nichols, and Xiaoyi Zhang. Extracting replayable interactions from videos of mobile app usage, 2022.

[92] Qiuyuan Chen, Xin Xia, Han Hu, David Lo, and Shanping Li. Why my code summarization model does not work: Code comment improvement with category prediction. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 30(2):1–29, 2021.

[93] Agnieszka Ciborowska and Kostadin Damevski. Fast changeset-based bug localization with bert. In *ICSE'22*.

[94] Norman Cliff. *Ordinal methods for behavioral data analysis.* Psychology Press, 2014.

[95] Marcos V Conde and Kerem Turgutlu. Clip-art: Contrastive pre-training for fine-grained art classification. In *CVPR'21*.

[96] Nathan Cooper, Carlos Bernal-Cárdenas, Oscar Chaparro, Kevin Moran, and Denys Poshyvanyk. It takes two to tango: Combining visual and textual information for detecting duplicate video-based bug reports. In *Proceedings of the 43rd IEEE/ACM International Conference on Software Engineering (ICSE'21)*, pages 160–161, 2021.

[97] Steven Davies and Marc Roper. What's in a Bug Report? In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM'14)*, pages 26:1–26:10, 2014.

[98] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *CVPR'09*.

[99] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[100] Mona Erfani Joorabchi, Mehdi Mirzaaghaei, and Ali Mesbah. Works for me! characterizing non-reproducible bug reports. In *MSR'14*.

[101] Mona Erfani Joorabchi, Mehdi Mirzaaghaei, and Ali Mesbah. Works for Me! Characterizing Non-reproducible Bug Reports. In *Proceedings of the Working Conference on Mining Software Repositories (MSR'14)*, pages 62–71, 2014.

[102] Camilo Escobar-Velásquez, Mario Linares-Vásquez, Gabriele Bavota, Michele Tufano, Kevin Moran, Massimiliano Di Penta, Christopher Vendome, Carlos Bernal-Cárdenas, and Denys Poshyvanyk. Enabling mutant generation for open-and closed-source android apps. *TSE'20*.

[103] Yuanrui Fan, Xin Xia, David Lo, and Ahmed E Hassan. Chaff from the wheat: Characterizing and determining valid bug reports. *TSE'18*.

[104] Mattia Fazzini, Kevin Patrick Moran, Carlos Bernal-Cardenas, Tyler Wendland, Alessandro Orso, and Denys Poshyvanyk. Enhancing mobile app bug reporting via real-time understanding of reproduction steps. *IEEE Transactions on Software Engineering*, page to appear, 2022.

[105] Mattia Fazzini, Martin Prammer, Marcelo d'Amorim, and Alessandro Orso. Automatically translating bug reports into test cases for mobile apps. In *Proceedings of the 27th ACM International Symposium on Software Testing and Analysis (ISSTA'18*, pages 141–152, 2018.

[106] Sidong Feng and Chunyang Chen. Gifdroid: an automated light-weight tool for replaying visual bug reports. In *ICSE'22*.

[107] Sidong Feng, Mulong Xie, Yinxing Xue, and Chunyang Chen. Read it, don't watch it: Captioning bug recordings automatically. *arXiv preprint arXiv:2302.00886*, 2023.

[108] Juan Manuel Florez, Oscar Chaparro, Christoph Treude, and Andrian Marcus. Combining query reduction and expansion for text-retrieval-based bug localization. In *SANER'21*.

[109] Jianfeng Gao, Michel Galley, and Lihong Li. Neural approaches to conversational ai. *Foundations and Trends in Information Retrieval*, 13(2-3):127–298, 2019.

[110] Gregory Gay, Sonia Haiduc, Andrian Marcus, and Tim Menzies. On the use of relevance feedback in ir-based concept location. In *2009 IEEE international conference on software maintenance*, pages 351–360. IEEE, 2009.

[111] Lorenzo Gomez, Iulian Neamtiu, Tanzirul Azim, and Todd Millstein. Reran: Timing- and touch-sensitive record and replay for android. In *Proceedings of the 35th International Conference on Software Engineering (ICSE'13)*, pages 72–81, 2013.

[112] Don Goodman-Wilson. Automating issue triage with github and recast.ai, 2018. `https://github.blog/2018-10-31-automating-issue-triage-with-github-and-recastai/`.

[113] Philip J. Guo, Thomas Zimmermann, Nachiappan Nagappan, and Brendan Murphy. Characterizing and predicting which bugs get fixed: An empirical study of Microsoft Windows. In *Proceedings of the International Conference on Software Engineering (ICSE'10)*, pages 495–504, 2010.

[114] Melita Hajdinjak and France Mihelič. The paradise evaluation framework: Issues and findings. *Computational Linguistics*, 32(2):263–272, 2006.

[115] Erik Hatcher and Otis Gospodnetic. *Lucene in Action*. Manning Publications, 2004.

[116] Madeleine Havranek, Carlos Bernal-Cárdenas, Nathan Cooper, Oscar Chaparro, Denys Poshyvanyk, and Kevin Moran. V2s: a tool for translating video recordings of mobile app usages into replayable scenarios. In *Proceedings of the IEEE/ACM 43rd International Conference on Software Engineering (ICSE'21)*, pages 65–68, 2021.

[117] Jianjun He, Ling Xu, Meng Yan, Xin Xia, and Yan Lei. Duplicate bug report detection using dual-channel convolutional neural networks. In *ICPC'20*.

[118] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR'16*.

[119] Zecheng He, Srinivas Sunkara, Xiaoxue Zang, Ying Xu, Lijuan Liu, Nevan Wichers, Gabriel Schubiner, Ruby Lee, and Jindong Chen. Actionbert: Leveraging user actions for semantic understanding of user interfaces. In *AAAI'21*.

[120] Myles Hollander, Douglas A Wolfe, and Eric Chicken. *Nonparametric statistical methods*. John Wiley & Sons, 2013.

[121] Pieter Hooimeijer and Westley Weimer. Modeling Bug Report Quality. In *Proceedings of the 22nd International Conference on Automated Software Engineering (ASE'07)*, pages 34–43, 2007.

[122] Md Kamal Hossen, Huzefa Kagdi, and Denys Poshyvanyk. Amalgamating source code authors, maintainers, and change proneness to triage change requests. In *Proceedings of the 22nd International Conference on Program Comprehension (ICPC'14)*, pages 130–141, 2014.

[123] Yongjian Hu, Tanzirul Azim, and Iulian Neamtiu. Versatile yet lightweight record-and-replay for android. In *Proceedings of the 2015 ACM SIGPLAN Interna-*

tional Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'15), pages 349–366, 2015.

[124] Dorota Huizinga and Adam Kolawa. *Automated defect prevention: best practices in software management.* John Wiley & Sons, 2007.

[125] Da Huo, Tao Ding, Collin McMillan, and Malcom Gethers. An empirical study of the effects of expert knowledge on bug reports. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME'14)*, pages 1–10, 2014.

[126] Mia Mohammad Imran, Agnieszka Ciborowska, and Kostadin Damevski. Automatically selecting follow-up questions for deficient bug reports. In *In proceedings of the 18th IEEE/ACM International Conference on Mining Software Repositories (MSR'18)*, pages 167–178, 2021.

[127] Gaeul Jeong, Sunghun Kim, and Thomas Zimmermann. Improving bug triage with bug tossing graphs. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the Symposium on The Foundations of Software Engineering (ESEC/FSE'09)*, pages 111–120, 2009.

[128] Thorsten Joachims. Making large-scale svm learning practical. LS8-Report 24, Universität Dortmund, LS VIII-Report, 1998.

[129] Thorsten Joachims. Text categorization with support vector machines: Learning with many relevant features. In *Proceedings of the European Conference on Machine Learning*, pages 137–142. Springer, 1998.

[130] Jack Johnson, Junayed Mahmud, Tyler Wendland, Kevin Moran, Julia Rubin, and Mattia Fazzini. An empirical investigation into the reproduction of bug reports for android apps. In *SANER'22*.

[131] Jack Johnson, Junayed Mahmud, Tyler Wendland, Kevin Moran, Julia Rubin, and Mattia Fazzini. An empirical investigation into the reproduction of bug reports for android apps. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 321–322. IEEE, 2022.

[132] Leif Jonsson, Markus Borg, David Broman, Kristian Sandahl, Sigrid Eldh, and Per Runeson. Automated bug assignment: Ensemble-based machine learning in large scale industrial contexts. *Empirical Software Engineering*, 21(4):1533–1578, 2016.

[133] Armand Joulin, Edouard Grave, Piotr Bojanowski, and Tomas Mikolov. Bag of tricks for efficient text classification. *arXiv preprint arXiv:1607.01759*, 2016.

[134] Huzefa Kagdi, Malcom Gethers, Denys Poshyvanyk, and Maen Hammad. Assigning change requests to software developers. *Journal of Software: Evolution and Process*, 24(1):3–33, 2012.

[135] Rafael Kallis, Andrea Di Sorbo, Gerardo Canfora, and Sebastiano Panichella. Ticket tagger: Machine learning driven issue classification. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME'19)*, pages 406–409, 2019.

[136] Gün Karagöz and Hasan Sözer. Reproducing failures based on semiformal failure scenario descriptions. *Software Quality Journal*, 25(1):111–129, 2017.

[137] Gün Karagöz and Hasan Sözer. Reproducing failures based on semiformal failure scenario descriptions. *Software Quality Journal*, 25(1):111–129, 2017.

[138] K. Kevic, S. C. Müller, T. Fritz, and H. C. Gall. Collaborative bug triaging using textual similarities and change set analysis. In *Proceedings of the 6th*

International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE'13), pages 17–24, 2013.

[139] Amy J. Ko and Brad A. Myers. Debugging reinvented: Asking and answering why and why not questions about program behavior. In *Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*, page 301–310, 2008.

[140] Pavneet Singh Kochhar, Yuan Tian, and David Lo. Potential biases in bug localization: Do they matter? In *Proceedings of the Conference on Automated Software Engineering (ASE'14)*, pages 803–814, 2014.

[141] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners, 2023.

[142] Hiroki Kuramoto, Masanari Kondo, Yutaro Kashiwa, Yuta Ishimoto, Kaze Shindo, Yasutaka Kamei, and Naoyasu Ubayashi. Do visual issue reports help developers fix bugs? a preliminary study of using videos and images to report issues on github. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, ICPC '22, page 511–515, New York, NY, USA, 2022. Association for Computing Machinery.

[143] Eero I. Laukkanen and Mika V. Mäntylä. Survey reproduction of defect reporting in industrial software development. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement*, ESEM'11, pages 197–206, 2011.

[144] Jaekwon Lee, Dongsun Kim, Tegawendé F Bissyandé, Woosung Jung, and Yves Le Traon. Bench4bl: reproducibility study on the performance of ir-based bug localization. In *ISSTA'18*.

[145] Sun-Ro Lee, Min-Jae Heo, Chan-Gun Lee, Milhan Kim, and Gaeul Jeong. Applying deep learning based automatic bug triager to industrial projects. In

Proceedings of the 11th Joint Meeting on foundations of software engineering (ES-EC/FSE'17), pages 926–931, 2017.

[146] Junnan Li, Dongxu Li, Caiming Xiong, and Steven Hoi. Blip: Bootstrapping language-image pre-training for unified vision-language understanding and generation. In *ICML'22*.

[147] Toby Jia-Jun Li, Lindsay Popowski, Tom Mitchell, and Brad A Myers. Screen2vec: Semantic embedding of gui screens and gui components. In *CHI'21*.

[148] Wei Li, Qingan Li, Yunlong Ming, Weijiao Dai, Shi Ying, and Mengting Yuan. An empirical study of the effectiveness of ir-based bug localization for large-scale industrial projects. *Empirical Software Engineering*, 27(2):47, 2022.

[149] Yang Li, Jiacong He, Xin Zhou, Yuan Zhang, and Jason Baldridge. Mapping natural language instructions to mobile ui action sequences. *arXiv preprint arXiv:2005.03776*, 2020.

[150] Yang Li, Gang Li, Luheng He, Jingjie Zheng, Hong Li, and Zhiwei Guan. Widget captioning: Generating natural language description for mobile user interface elements. *arXiv preprint arXiv:2010.04295*, 2020.

[151] Yang Li, Gang Li, Xin Zhou, Mostafa Dehghani, and Alexey Gritsenko. Vut: Versatile ui transformer for multi-modal multi-task user interface modeling. *arXiv preprint arXiv:2112.05692*, 2021.

[152] Bennett P Lientz and E Burton Swanson. *Software maintenance management*. Addison-Wesley Longman Publishing Co., Inc., 1980.

[153] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. Api change and fault proneness: A threat to the success of android apps. In *Proceedings*

of the 9th Joint Meeting on Foundations of Software Engineering (FSE'13), pages 477–487, 2013.

[154] Mario Linares-Vásquez, Kamal Hossen, Hoang Dang, Huzefa Kagdi, Malcom Gethers, and Denys Poshyvanyk. Triaging incoming change requests: Bug or commit history, or code authorship? In *Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM'12)*, pages 451–460. IEEE, 2012.

[155] Hui Liu, Mingzhu Shen, Jiahao Jin, and Yanjie Jiang. Automated classification of actions in bug reports of mobile apps. In *Proceedings of the 29th ACM International Symposium on Software Testing and Analysis (ISSTA'20)*, pages 128–140, 2020.

[156] Junayed Mahmud, Nadeeshan De Silva, Safwat Ali Khan, Seyed Hooman Mostafavi, SM Mansur, Oscar Chaparro, Andrian Marcus, and Kevin Moran. On using gui interaction data to improve text retrieval-based bug localization. *ICSE'24*, 2024.

[157] Christopher D Manning, Mihai Surdeanu, John Bauer, Jenny Rose Finkel, Steven Bethard, and David McClosky. The stanford corenlp natural language processing toolkit. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL'14)*, pages 55–60, 2014.

[158] D. Matter, A. Kuhn, and O. Nierstrasz. Assigning bug reports using a vocabulary-based expertise model of developers. In *Proceedings of the 6th International Working Conference on Mining Software Repositories (MSR'09)*, pages 131–140, 2009.

[159] Dominique Matter, Adrian Kuhn, and Oscar Nierstrasz. Assigning bug reports using a vocabulary-based expertise model of developers. In *MSR'09*, pages 131–140, 2009.

[160] Chris Mills, Gabriele Bavota, Sonia Haiduc, Rocco Oliveto, Andrian Marcus, and Andrea De Lucia. Predicting query quality for applications of text retrieval to software engineering tasks. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 26(1):3, 2017.

[161] Kevin Moran, Richard Bonett, Carlos Bernal-Cárdenas, Brendan Otten, Daniel Park, and Denys Poshyvanyk. On-device bug reporting for android applications. In *Proceedings of the IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft'17)*, pages 215–216, 2017.

[162] Kevin Moran, Mario Linares-Váquez, Carlos Bernal-Cárdenas, Christopher Vendome, and Denys Poshyvanyk. Automatically Discovering, Reporting and Reproducing Android Application Crashes. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST'16)*, pages 33–44, 2016.

[163] Kevin Moran, Mario Linares-Vásquez, Carlos Bernal-Cárdenas, and Denys Poshyvanyk. Auto-completing Bug Reports for Android Applications. In *Proceedings of the Joint Meeting on Foundations of Software Engineering (FSE'15)*, pages 673–686, 2015.

[164] Kevin Moran, Mario Linares-Vasquez, Carlos Bernal-Cardenas, Cristopher Vendome, and Denys Poshyvanyk. Crashscope: A practical tool for automated testing of android applications. In *Proceedings of the 39th IEEE/ACM International Conference on Software Engineering (ICSE'17)*, pages 15–18, 2017.

[165] Kevin Moran, Ali Yachnes, George Purnell, Junayed Mahmud, Michele Tufano, Carlos Bernal Cardenas, Denys Poshyvanyk, and Zach H'Doubler. An empirical investigation into the use of image captioning for automated software documentation. In *SANER'22*.

[166] Laura Moreno, Gabriele Bavota, Sonia Haiduc, Massimiliano Di Penta, Rocco Oliveto, Barbara Russo, and Andrian Marcus. Query-based configuration of text retrieval solutions for software engineering tasks. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'15)*, pages 567–578, 2015.

[167] Laura Moreno, John Joseph Treadway, Andrian Marcus, and Wuwei Shen. On the use of stack traces to improve text retrieval-based bug localization. In *Proceedings of the Conference on Software Maintenance and Evolution (ICSME'14)*, pages 151–160, 2014.

[168] K. Morik, P. Brockhausen, and T. Joachims. Combining statistical learning with a knowledge-based approach – a case study in intensive care monitoring. In *International Conference on Machine Learning (ICML)*, pages 268–277, Bled, Slowenien, 1999.

[169] H. Naguib, N. Narayan, B. Brügge, and D. Helal. Bug report assignee recommendation using activity profiles. In *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR'13)*, pages 22–30, 2013.

[170] Maleknaz Nayebi. Eye of the mind: Image processing for social coding. In *ICSE'20*.

[171] Abraham Naftali Oppenheim. *Questionnaire Design, Interviewing and Attitude Measurement*. Pinter Publishers, 1992.

[172] Kai Pan, Sunghun Kim, and E James Whitehead. Toward an understanding of bug fix patterns. *Empirical Software Engineering*, 14:286–315, 2009.

[173] Panupong Pasupat, Tian-Shun Jiang, Evan Zheran Liu, Kelvin Guu, and Percy Liang. Mapping natural language commands to web elements. *arXiv preprint arXiv:1808.09132*, 2018.

[174] Zhengrui Qin, Yutao Tang, Ed Novak, and Qun Li. MobiPlay: A Remote Execution Based Record-and-replay Tool for Mobile Applications. In *Proceedings of the 38th International Conference on Software Engineering (ICSE'16)*, pages 571–582, 2016.

[175] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. Learning transferable visual models from natural language supervision. In *ICML'21*.

[176] Jack W. Rae, Sebastian Borgeaud, Trevor Cai, Katie Millican, Jordan Hoffmann, Francis Song, John Aslanides, Sarah Henderson, Roman Ring, Susannah Young, Eliza Rutherford, Tom Hennigan, Jacob Menick, Albin Cassirer, Richard Powell, George van den Driessche, Lisa Anne Hendricks, Maribeth Rauh, Po-Sen Huang, Amelia Glaese, Johannes Welbl, Sumanth Dathathri, Saffron Huang, Jonathan Uesato, John Mellor, Irina Higgins, Antonia Creswell, Nat McAleese, Amy Wu, Erich Elsen, Siddhant Jayakumar, Elena Buchatskaya, David Budden, Esme Sutherland, Karen Simonyan, Michela Paganini, Laurent Sifre, Lena Martens, Xiang Lorraine Li, Adhiguna Kuncoro, Aida Nematzadeh, Elena Gribovskaya, Domenic Donato, Angeliki Lazaridou, Arthur Mensch, Jean-Baptiste Lespiau, Maria Tsimpoukelli, Nikolai Grigorev, Doug Fritz, Thibault Sottiaux, Mantas Pajarskas, Toby Pohlen, Zhitao Gong, Daniel Toyama, Cyprien de Masson d'Autume, Yujia Li, Tayfun Terzi, Vladimir Mikulik, Igor Babuschkin, Aidan Clark, Diego de Las Casas, Aurelia Guy, Chris Jones, James Bradbury, Matthew Johnson, Blake Hechtman, Laura Weidinger, Iason Gabriel, William Isaac, Ed Lockhart, Simon Osindero, Laura Rimell, Chris Dyer, Oriol Vinyals, Kareem Ayoub, Jeff Stanway, Lorrayne

Bennett, Demis Hassabis, Koray Kavukcuoglu, and Geoffrey Irving. Scaling language models: Methods, analysis insights from training gopher, 2022.

[177] Sebastian Raschka. Model evaluation, model selection, and algorithm selection in machine learning. *arXiv preprint arXiv:1811.12808*, 2018.

[178] Hanoona Rasheed, Muhammad Uzair Khattak, Muhammad Maaz, Salman Khan, and Fahad Shahbaz Khan. Fine-tuned clip models are efficient video learners. *arXiv preprint arXiv:2212.03640*, 2022.

[179] Nils Reimers and Iryna Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks. *arXiv preprint arXiv:1908.10084*, 2019.

[180] Stephen Robertson and Hugo Zaragoza. *The probabilistic relevance framework: BM25 and beyond*. Now Publishers Inc, 2009.

[181] Per Runeson, Magnus Alexandersson, and Oskar Nyholm. Detection of duplicate defect reports using natural language processing. In *Proceedings of the 29th International Conference on Software Engineering (ICSE'07)*, pages 499–510, 2007.

[182] Swarup Kumar Sahoo, John Criswell, and Vikram Adve. An empirical study of reported bugs in server software with implications for automated bug diagnosis. In *Proceedings of the International Conference on Software Engineering (ICSE'10)*, pages 485–494, 2010.

[183] Ali Sajedi Badashian, Abram Hindle, and Eleni Stroulia. Crowdsourced bug triaging: Leveraging q&a platforms for bug assignment. In *International Conference on Fundamental Approaches to Software Engineering*, pages 231–248, 2016.

[184] Ali Sajedi-Badashian and Eleni Stroulia. Guidelines for evaluating bug-assignment research. *J. of Software: Evolution and Process*, 32(9):e2250, 2020.

[185] G. Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. *Communications of the ACM*, 18(11):613–620, 1975.

[186] Aindrila Sarkar, Peter C Rigby, and Béla Bartalos. Improving bug triaging with high confidence predictions at ericsson. In *Proceedings of the 35th International Conference on Software Maintenance and Evolution (ICSME'19)*, pages 81–91, 2019.

[187] Tommaso Dal Sasso, Andrea Mocci, and Michele Lanza. What Makes a Satisficing Bug Report? In *Proceedings of the International Conference on Software Quality, Reliability and Security (QRS'16)*, pages 164–174, 2016.

[188] Ramin Shokripour, John Anvik, Zarinah M. Kasirun, and Sima Zamani. Why so complicated? simple term filtering and weighting for location-based bug report assignment recommendation. In *Proceedings of the Working Conference on Mining Software Repositories*, pages 2–11, 2013.

[189] Ramin Shokripour, John Anvik, Zarinah M Kasirun, and Sima Zamani. A time-based approach to automatic bug report assignment. *Journal of Systems and Software*, 102:109–122, 2015.

[190] Mozhan Soltani, Felienne Hermans, and Thomas Bäck. The significance of bug report elements. *ESE'20*.

[191] Yang Song and Oscar Chaparro. Bee: a tool for structuring and analyzing bug reports. In *FSE'20*.

[192] Yang Song and Oscar Chaparro. Bee: a tool for structuring and analyzing bug reports. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'21)*, pages 1551–1555, 2020.

[193] Yang Song and Oscar Chaparro. Recommending bug assignment approaches for individual bug reports: An empirical investigation, 2023.

[194] Yang Song, Junayed Mahmud, Nadeeshan De Silva, Ying Zhou, Oscar Chaparro, Kevin Moran, Andrian Marcus, and Denys Poshyvanyk. BURT: A Chatbot for Interactive Bug Reporting. In *ICSE'23*.

[195] Yang Song, Junayed Mahmud, Ying Zhou, Oscar Chaparro, Kevin Moran, Andrian Marcus, and Denys Poshyvanyk. Toward interactive bug reporting for (Android app) end-users. In *FSE'22*.

[196] Yanqi Su, Zheming Han, Zhenchang Xing, Xin Xia, Xiwei Xu, Liming Zhu, and Qinghua Lu. Constructing a system knowledge graph of user tasks and failures from bug reports to support soap opera testing. In *ASE'22*.

[197] Chengnian Sun, David Lo, Siau-Cheng Khoo, and Jing Jiang. Towards more accurate retrieval of duplicate bug reports. In *Proceedings of the 26th International Conference on Automated Software Engineering (ASE'11)*, pages 253–262, 2011.

[198] Xiaobing Sun, Hui Yang, Xin Xia, and Bin Li. Enhancing developer recommendation with supplementary information via mining historical commits. *Journal of Systems and Software*, 134:355–368, 2017.

[199] A. Tamrawi, T. T. Nguyen, J. Al-Kofahi, and T. N. Nguyen. Fuzzy set-based automatic bug triaging. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE'11)*, pages 884–887, 2011.

[200] Ahmed Tamrawi, Tung Thanh Nguyen, Jafar M. Al-Kofahi, and Tien N. Nguyen. Fuzzy set and cache-based approach for bug triaging. In *Proceedings of the 19th Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE '11)*, pages 365–375, 2011.

[201] Lin Tan, Chen Liu, Zhenmin Li, Xuanhui Wang, Yuanyuan Zhou, and Chengxiang Zhai. Bug characteristics in open source software. *Empirical Software Engineering*, 19(6):1665–1705, 2014.

[202] Ferdian Thung, David Lo, and Lingxiao Jiang. Automatic defect categorization. In *Proceedings of the 19th Working Conference on Reverse Engineering (WCRE'12)*, pages 205–214. IEEE, 2012.

[203] Yuan Tian, David Lo, Xin Xia, and Chengnian Sun. Automated prediction of bug report priority using multi-factor analysis. *Empirical Software Engineering*, 20(5):1354–1383, 2015.

[204] Yuan Tian, Dinusha Wijedasa, David Lo, and Claire Le Goues. Learning to rank for bug report assignee recommendation. In *Proceedings of the 24th IEEE International Conference on Program Comprehension (ICPC'16)*, pages 1–10. IEEE, 2016.

[205] Davor Čubranić. Automatic bug triage using text categorization. In *Proceedings of the 16th International Conference on Software Engineering and Knowledge Engineering (SEKE'04)*, pages 92–97. KSI Press, 2004.

[206] Bryan Wang, Gang Li, Xin Zhou, Zhourong Chen, Tovi Grossman, and Yang Li. Screen2words: Automatic mobile ui summarization with multimodal learning. In *UIST'21*.

[207] Junjie Wang, Mingyang Li, Song Wang, Tim Menzies, and Qing Wang. Images don't lie: Duplicate crowdtesting reports detection with screenshot information. *IST'19*.

[208] Song Wang, Wen Zhang, and Qing Wang. Fixercache: Unsupervised caching active developers for diverse bug triage. In *Proceedings of the 8th ACM/IEEE In-*

ternational Symposium on Empirical Software Engineering and Measurement, pages 1–10, 2014.

[209] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models, 2023.

[210] Lili Wei, Yepang Liu, and Shing-Chi Cheung. Taming android fragmentation: Characterizing and detecting compatibility issues for android apps. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE'16)*, pages 226–237, 2016.

[211] Tyler Wendland, Jingyang Sun, Junayed Mahmud, SM Hasan Mansur, Steven Huang, Kevin Moran, Julia Rubin, and Mattia Fazzini. Andror2: A dataset of manually-reproduced bug reports for android apps. In *MSR'21*.

[212] Chu-Pan Wong, Yingfei Xiong, Hongyu Zhang, Dan Hao, Lu Zhang, and Hong Mei. Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis. In *Proceedings of the Conference on Software Maintenance and Evolution (ICSME'14)*, pages 181–190, 2014.

[213] Xiaoxue Wu, Wei Zheng, Xin Xia, and David Lo. Data quality matters: A case study on data label correctness for security bug report prediction. *TSE'21*.

[214] X. Xia, D. Lo, Y. Ding, J. M. Al-Kofahi, T. N. Nguyen, and X. Wang. Improving automated bug triaging with specialized topic model. *IEEE Transactions on Software Engineering*, 43(3):272–297, 2017.

[215] Xin Xia, David Lo, Xinyu Wang, and Bo Zhou. Dual analysis for recommending developers to resolve bugs. *Journal of Software: Evolution and Process*, 27(3):195–220, 2015.

[216] Bowen Xu, David Lo, Xin Xia, Ashish Sureka, and Shanping Li. Efspredictor: Predicting configuration bugs with ensemble feature selection. In *Proceedings of the Asia-Pacific Software Engineering Conference (ASPEC'15)*, pages 206–213, 2015.

[217] Nancy Xu, Sam Masling, Michael Du, Giovanni Campagna, Larry Heck, James Landay, and Monica S Lam. Grounding open-domain instructions to automate web support tasks. *arXiv preprint arXiv:2103.16057*, 2021.

[218] Xin Ye, Razvan Bunescu, and Chang Liu. Learning to rank relevant files for bug reports using domain knowledge. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE'14)*, pages 689–699, 2014.

[219] Xin Ye, Hui Shen, Xiao Ma, Razvan Bunescu, and Chang Liu. From word embeddings to document similarities for improved information retrieval in software engineering. In *Proceedings of the International Conference on Software Engineering*, pages 404–415, 2016.

[220] Nor Shahida Mohamad Yusop, John Grundy, and Rajesh Vasa. Reporting usability defects: do reporters report what software developers need? In *EASE'16*.

[221] Marcelo Serrano Zanetti, Ingo Scholtes, Claudio Juan Tessone, and Frank Schweitzer. Categorizing bugs with social networks: A case study on four open source software communities. In *Proceedings of the International Conference on Software Engineering (ICSE'13)*, pages 1032–1041, 2013.

[222] Motahareh Bahrami Zanjani, Huzefa Kagdi, and Christian Bird. Using developer-interaction trails to triage change requests. In *Proceedings of the 12th International Working Conference on Mining Software Repositories (MSR'15)*, pages 88–98, 2015.

[223] Renrui Zhang, Ziyu Guo, Wei Zhang, Kunchang Li, Xupeng Miao, Bin Cui, Yu Qiao, Peng Gao, and Hongsheng Li. Pointclip: Point cloud understanding by clip. In *CVPR'22*.

[224] Tao Zhang, Jiachi Chen, He Jiang, Xiapu Luo, and Xin Xia. Bug report enrichment with application of automated fixer recommendation. In *Proceedings of the 25th International Conference on Program Comprehension (ICPC'17)*, pages 230–240, 2017.

[225] Wei Zhang. Efficient bug triage for industrial environments. In *Proceedings of the 36th IEEE International Conference on Software Maintenance and Evolution (ICSME'20)*, pages 727–735. IEEE, 2020.

[226] Zhaoxu Zhang, Robert Winn, Yu Zhao, Tingting Yu, and William GJ Halfond. Automatically reproducing android bug reports using natural language processing and reinforcement learning. *arXiv preprint arXiv:2301.07775*, 2023.

[227] Zhuosheng Zhang, Aston Zhang, Mu Li, Hai Zhao, George Karypis, and Alex Smola. Multimodal chain-of-thought reasoning in language models, 2023.

[228] Yu Zhao, Kye Miller, Tingting Yu, Wei Zheng, and Minchao Pu. Automatically extracting bug reproducing steps from android bug reports. In *International Conference on Software and Systems Reuse (ICSR'19)*, pages 100–111. Springer, 2019.

[229] Yu Zhao, Kye Miller, Tingting Yu, Wei Zheng, and Minchao Pu. Automatically extracting bug reproducing steps from android bug reports. In *Proceedings of the International Conference on Software and Systems Reuse (ICSR'19)*, pages 100–111, 2019.

[230] Yu Zhao, Ting Su, Yang Liu, Wei Zheng, Xiaoxue Wu, Ramakanth Kavuluru, William GJ Halfond, and Tingting Yu. Recdroid+: Automated end-

to-end crash reproduction from bug reports for android apps. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 31(3):1–33, 2022.

[231] Yu Zhao, Tingting Yu, Ting Su, Yang Liu, Wei Zheng, Jingzhi Zhang, and William G.J. Halfond. Recdroid: Automatically reproducing android application crashes from bug reports. In *Proceedings of the 41st IEEE/ACM International Conference on Software Engineering (ICSE'19)*, pages 128–139, 2019.

[232] Yu Zhao, Tingting Yu, Ting Su, Yang Liu, Wei Zheng, Jingzhi Zhang, and William GJ Halfond. Recdroid: automatically reproducing android application crashes from bug reports. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 128–139. IEEE, 2019.

[233] Jian Zhou and Hongyu Zhang. Learning to rank duplicate bug reports. In *Proceedings of the 21st International Conference on Information and Knowledge Management (CIKM'12)*, pages 852–861, 2012.

[234] Jian Zhou, Hongyu Zhang, and David Lo. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *2012 34th International conference on software engineering (ICSE)*, pages 14–24. IEEE, 2012.

[235] Yu Zhou, Yanxiang Tong, Ruihang Gu, and Harald Gall. Combining text mining and data mining for bug report classification. *Journal of Software: Evolution and Process*, 28(3):150–176, 2016.

[236] Thomas Zimmermann, Nachiappan Nagappan, Philip J. Guo, and Brendan Murphy. Characterizing and predicting which bugs get reopened. In *Proceedings of the International Conference on Software Engineering (ICSE'12)*, pages 1074–1083, 2012.

[237] Thomas Zimmermann, Rahul Premraj, Nicolas Bettenburg, Sascha

Just, Adrian Schröter, and Cathrin Weiss. What Makes a Good Bug Report? *IEEE Transactions on Software Engineering*, 36(5):618–643, 2010.

[238] Thomas Zimmermann, Rahul Premraj, Jonathan Sillito, and Silvia Breu. Improving bug tracking systems. In *Proceedings of the International Conference on Software Engineering (ICSE'09)*, pages 247–250, 2009.

[239] Weiqin Zou, David Lo, Zhenyu Chen, Xin Xia, Yang Feng, and Baowen Xu. How practitioners perceive automated bug report management techniques. *TSE'18*.