Decoding the Issue Resolution Process In Practice via Issue Report Analysis: A Case Study of Firefox

Antu Saha
William & Mary
Williamsburg, Virginia, USA
asaha02@wm.edu

Oscar Chaparro
William & Mary
Williamsburg, Virginia, USA
oscarch@wm.edu

Abstract—**Effectively managing and resolving software issues is critical for maintaining and evolving software systems. Development teams often rely on issue trackers and issue reports to track and manage the work needed during issue resolution, ranging from issue reproduction and analysis to solution design, implementation, verification, and deployment. Despite the issue resolution process being generally known in the software engineering community as a sequential list of activities, it is unknown how developers implement this process in practice and how they discuss it in issue reports. This paper aims to enhance our understanding of the issue resolution process implemented in practice by analyzing the issue reports of Mozilla Firefox. We qualitatively and quantitatively analyzed the discussions found in 356 Firefox issue reports, to identify the sequences of stages that developers go through to address various software problems. We analyzed the sequences to identify the overall resolution process at Firefox and derived a catalog of 47 patterns that represent instances of the process. We analyzed the process and patterns across multiple dimensions, including pattern complexity, issue report types, problem categories, and issue resolution times, resulting in various insights about Mozilla's issue resolution process. We discuss these findings and their implications for different stakeholders on how to better assess and improve the issue resolution process.**

I. INTRODUCTION

Issue management is a fundamental process that aims to track and manage the code changes needed to address issues during the maintenance and evolution of a software project. Issue trackers are essential tools that provide the infrastructure to implement issue management [1]. Such systems provide a platform for documenting software issues, facilitating discussions among stakeholders, and tracking the work and progress of solving the issues [1, 2]. The issue management process assisted by issue trackers, typically involves steps such as issue understanding, triage, replication, and analysis, as well as issue fixing (*a.k.a.* issue resolution) [2, 3]. Issue resolution is a sub-process of issue management that aims to diagnose and resolve the reported problems.

According to existing literature [4–8], the typical issue resolution process includes steps such as issue reproduction, problem investigation, solution design, solution implementation, and validation/verification, which are sequentially applied to solve issues. However, while this process is meant to be generally applied to any software issue, it is unclear how developers implement it in practice for different problems and contexts and how developers discuss it in issue reports.

Understanding the issue resolution process implemented in practice is important for improving software maintenance and evolution processes. By gaining insights into how developers address software problems, we can identify bottlenecks and anomalous process implementations, align prescribed processes with actual practices, and provide developers with better guidance for issue resolution. Additionally, studying issue resolution can help identify common patterns and strategies that can be applied to similar problems in the future, and confirm the extent to which the implemented process deviates from the typical, linear resolution process from the literature.

This paper aims to enhance our understanding of the issue resolution process implemented in practice by identifying and analyzing the sequence of steps (*i.e.*, stages) that developers perform and discuss in issue reports when solving issues. To that end, we conducted a case study on Mozilla Firefox, a mature and widely-used open-source project. Combining qualitative and quantitative methods, we analyzed the discussions present in a sample of 356 Firefox issue reports to identify the stages of issue resolution that Firefox developers engage in, the sequences of stages that issue discussions form, recurrent transitions between stages present in the sequences, the overall issue resolution process implemented at Firefox, and the recurrent instances of this process to solve a variety of problem types reported in different issue reports.

Using a multi-coder iterative open-coding methodology, we identified six issue resolution stages (*e.g.*, issue reproduction, solution design, implementation, and code review). The stages appear in issue reports with varying frequencies across different issue types (defects, enhancements, and tasks) and problem categories (*e.g.*, Crashes, UI Issues, and Code Improvements), and form sequences that represent particular implementations of the resolution process at Firefox. The stage sequences reveal frequent relationships among stages, particularly between issue reproduction and analysis; among solution design, implementation, and code review; and among implementation, code review, and solution verification. Additionally, based on analysis of consecutive stages appearing in the sequences (*i.e.*, bi-grams), we identified the most common transitions between stages and derived the overall issue resolution process at Firefox from them. Such a process is primarily iterative, deviating from the theoretical linear process found in the literature and Firefox's documentation. In this process, developers go back and forth from one

1

stage to another as needed to solve the issues. Finally, utilizing qualitative analysis of the sequences, we identified 47 issue resolution patterns that represent recurrent instances of the overall process of solving different types of problems. Two Mozilla developers assessed the usefulness of the patterns, identifying potential use cases to enhance Firefox's resolution process.

Our study provides evidence of the iterative and diverse nature of the issue resolution process, which widely deviates from the theoretical linear process from the literature. Our methodology, stage sequences, and patterns serve to identify potential anomalies in the way Firefox developers implement the resolution process. Our pattern catalog and results can help educate future developers and train newcomers at Firefox in the intricate process of issue resolution. Finally, we advocate for developing advanced tooling to assist developers in recording issue resolution activities more easily, as this can have great benefits for traceability, process assessment, code change rationale management, and more.

In summary, this paper makes the following contributions:

- A model of Firefox's issue resolution process implemented in practice, derived from qualitative and quantitative analysis of issue report discussions.
- A novel catalog of 47 patterns of issue resolution, derived from qualitative analysis of issue discussions. The patterns represent instances of Firefox's resolution process, employed by Firefox developers to address different types of software problems.
- A comprehensive analysis of the derived process and patterns, across different types of issues and problem categories, showing that Firefox's issue resolution is a diverse and iterative process, which deviates from the prescribed linear process from the literature.
- A novel dataset with annotated issues and related artifacts that enables further research in this area. We publicly release this data and the derived catalog, scripts, and other artifacts useful to validate and replicate our study [9].

II. BACKGROUND, PROBLEM, AND MOTIVATION

A. Issue Management during Software Evolution

Issue management aims to track and manage all the change requests of a software system, including new feature developments, non-functional implementations, defect corrections, and enhancements to existing functionality [2, 6, 7]. Issue trackers, such as Bugzilla [10], Jira [11], and GitHub Issues [12], are communication and social platforms that allow coordination among different stakeholders around the process of issue management [13–15]. **Issue reports** are the main artifacts created and used during issue management [2], and include an issue title, a detailed issue description, metadata (*e.g.*, issue severity and priority, operating platform, and product versions affected by the issue), and attachments (*e.g.*, system logs and screenshots).

During the process of managing and solving the issues, stakeholders change the status of the issues (*e.g.*, from *Assigned* to *Resolved* and from *Verified* to *Closed* [8, 16]), and engage in discussions when needed. These discussions are recorded in **issue comments**, which document relevant information about the reported problems [6], including possible circumstances in which the problem occurs, potential causes, how the code should be changed to address the issue, and more.

B. Issue Resolution within the Issue Management Process

The literature has defined the different phases that compose the issue management process. Zhang *et al.* [4] reported three major phases: issue understanding, triaging, and fixing (*a.k.a.* **issue resolution**). In the first step, triagers read and understand the report to determine the issue type, priority, and severity. In the second step, the triagers assign the issue to an expert developer. Finally, in the issue resolution phase, the assigned developer locates the code that needs to be changed in response to the issue and implements the desired code change.

K. Saha *et al.* [5] define similar phases, including a fourth phase called issue verification, in which a developer verifies that the code change indeed addresses the issue and conforms with the quality standards. Zeller [6] includes issue reporting, duplicate report identification, and fix delivery as part of the issue management process. Zeller also decomposes the issue resolution phase for bug reports into multiple steps: bug reproduction, isolation and localization, and fix implementation.

Rajlich [7] defines software change as a general process to modify and evolve software systems, based on change requests. Rajlich defines seven phases of software change: initiation, concept location, impact analysis, actualization, refactoring, verification, and conclusion. Initiation includes prioritization of change requests. All these phases except for initiation and conclusion are part of issue resolution, which aims to analyze and implement the solution to an issue.

In this paper, we investigate how developers perform issue resolution, a critical sub-process of issue management that aims to diagnose and solve the reported problems [8]. This **issue resolution process** includes reproducing the reported bugs (for bug reports), understanding and analyzing the issues, designing a solution to the issue, implementing the solution, and validating/verifying the quality of the implementation.

C. The Issue Resolution Process Implemented in Practice

Although the issue resolution process includes different activities as discussed above, it is unclear how this process is implemented in practice to address different kinds of issues, under various operating circumstances. As we discuss in the related work section (Section VII), prior work has focused on studying different aspects of issue reports and their management, including the overall issue management process based on issue status transitions, but no prior work has studied in detail the process that developers implement in practice to analyze and solve issues. We fill in this knowledge gap by analyzing issue discussions and studying **patterns of issue resolution**.

We motivate our work by discussing two examples of issue resolution at Mozilla Firefox [17]. Issue report #1029919 [18] describes a buggy behavior in the way Firefox renders a web page: when the user hovers over HTML buttons in a page,

Firefox draws a border around the button. The issue report contains rich information in the issue comments that help us understand the process followed to solve this issue. At first, developers reported multiple bug reproduction attempts, asking for additional information from the reporter. After it was successfully reproduced, the developer assigned to solve the issue posted the result of his investigation, describing the potential problem location and cause. The developer then attached two fixing patches describing the root cause, the solution, and the potential impact of the solution on the system. The patches were reviewed by another developer and after the original developer corrected a few problems, the code reviewer inspected and approved the implementation. The code change was then integrated into Mozilla's code base, the reporter verified the fix, and a triager closed the issue.

Another example is issue #1718748 [19], which describes a failure in Firefox's cross-platform component that handles UI rendering. The issue states that some buttons in Firefox's toolbar customization UI become invisible when switching to a dark theme. The reporter is a QA member who identified a prior commit and issue that could have introduced the bug (via Mozilla's mozregression tool [20]). The developer responsible for that prior commit and issue, assigned to solve the issue, provided a patch with the description of the code change to correct the defect. Another developer reviewed the code change (via the Phabricator code review tool [21]) and a QA member then successfully verified the solution, marking the issue as *Verified* and *Fixed*.

Both examples illustrate different ways to resolve bugs related to Firefox's web page and UI rendering. In the first example, we observe all the expected major steps of issue resolution, however, in the second example, the process did not include any issue reproduction and analysis. In both cases, these issue resolution steps were performed by different stakeholders, recording the activities and the relevant information obtained during the issue resolution. While the nature of the problems might have been different, it is clear that the issue resolution process that we would expect from theory can be implemented and recorded in issue reports in different ways. Our goal is to investigate these different approaches and determine if there are recurrent patterns in the process of solving different kind of issues. We do so by qualitatively analyzing the discussions that developers document in issue reports.

D. The Issue Resolution Process at Mozilla Firefox

We selected Mozilla Firefox [17] as the subject of our study because (1) it is a mature and widely-used project with 19+ years of evolution, and (2) it has well-documented practices for issue management [3] and software development (*e.g.*, patching [22], code quality [23, 24], testing [25], and debugging [26]), which allow us to understand Firefox's issue resolution process in detail. Mozilla Firefox is a multi-language, multi-platform open source project that uses BMO, an adapted version of the Bugzilla issue tracker [10], to manage all the changes made to Firefox' source code [22].

All of Firefox's code changes are documented in issue reports by end-users, community members, QA members, and developers during system usage, testing, and analysis [27]. Firefox has three **issue report types** [28]: defects, enhancements (*i.e.*, user-facing improvements), and tasks (*i.e.*, back-end improvements). These issues are triaged differently by a rotating group of engineering managers who are owners of a Firefox component and by QA members [29–31]. These members assess the issues and assign a correct issue type, severity, priority, target release, and other metadata (*e.g.*, security flags [31]) to better prioritize and manage the problems. QA members, component owners, and developers are in charge of determining the resolution state of the issues (*e.g.*, *Resolved - Won't fix* or *Verified - Fixed* [30]).

The open nature of the project makes Firefox's software development, and in particular issue resolution, a worldwide and distributed process. Developers are assigned to issues and work on one or more patches to address the problems. For diagnosing and solving defects, Firefox provides guidelines to use several debugging tools across different platforms [26]. Once the patches are completed, they are attached to issue reports, requiring a code review via the Phabricator tool [21]. The tool posts comments on the issues whenever a code review is submitted. The code reviewer is mainly a component owner or peer, a newcomer mentor, and/or any other developer familiar with the modified code or module [22]. The patch is tested in Try [32], a system for running automated tests without integrating patches into Firefox's code base. Once the patches are approved, they are integrated (*a.k.a.* landed), by the code reviewer, into the 'autoland' repository, where regression tests are executed [33]. Once the tests pass and the code changes are further validated/verified by the QA team, they are merged by 'code sheriffs' into 'mozilla-central', Firefox's main development repository [33]. Merging into 'mozilla-central' occurs periodically or on-demand (*e.g.*, when critical security fixes are validated) [33, 34].

During the resolution process, the status of the reports is updated accordingly (*e.g.*, from *Assigned* to *Verified and Fixed*). Any information relevant to the issue (*e.g.*, failing regression test results), obtained at any moment during the process, is posted as an issue comment. For example, failing regression test results are posted in the issues. Code changes in 'mozilla-central' are integrated into 'mozilla-beta' for additional quality assurance during a four-week beta cycle. After this, a release candidate build is generated, tested thoroughly, and made available as the next version of Firefox [34].

III. STUDY METHODOLOGY

This study aims to investigate how the issue resolution process is implemented in practice at Mozilla Firefox to solve various software problems and tasks described in issue reports. We investigate the major stages of the issue resolution process, described in Section II-B, and how developers[1] follow them to solve a variety of problem categories (*e.g.*, crashes, UI issues,

---

[1]We hereon use *developers* to refer to all stakeholders involved in issue resolution: programmers, reporters, QA members, *etc.*

or refactoring changes) reported in various issue report types (defects, enhancements, or tasks). The study addresses the following research questions (RQs):

RQ$_1$: *What issue resolution stages are found in issue reports?*
RQ$_2$: *How do the resolution stages interact with each other?*
RQ$_3$: *What is the overall process of issue resolution?*
RQ$_4$: *What resolution patterns are found in issue reports?*
RQ$_5$: *What are the potential use cases of the patterns?*

RQ$_1$ investigates the major stages that Mozilla developers go through to address reported issues and how frequently these stages are discussed in issue reports. **RQ$_2$** investigates how these stages interact with one another, including how frequently these stages co-occur in issue reports. **RQ$_3$** investigates the overall issue resolution process at Mozilla Firefox. **RQ$_4$** investigates recurrent instances of the resolution process, expressed as sequences of stages. **RQ$_5$** examines the potential applications of the derived patterns for Mozilla developers.

A. Issue Collection

Mozilla's BMO is the centralized system for managing the issues of Firefox desktop and mobile [35]. In this study, we focused on the desktop version of Mozilla Firefox, studying the issues of its two main components: *Firefox* and *Core*. The *Firefox* component (*a.k.a. product* in BMO) implements the graphical user interface (GUI) of the web browser, while the *Core* component includes essential functionality such as web page rendering, web browsing, and networking services.

Our study focused on *FIXED* and *RESOLVED* issue reports for the selected components. To obtain recent issues with significant periods of system evolution, we downloaded all the issues created from January 1st, 2010 to April 30th, 2023 using Bugzilla's API [36], including their title/summary, comments (which contain the issue description), and relevant metadata: creation time, resolution time, and others. From 199,271 downloaded issues ($\approx$164.7k/34.5k for Core/Firefox), we randomly sampled 384 issues for study. This is a statistically significant sample, at 95% confidence level and 5% error margin, that captures the diversity and characteristics of the entire population of *Core* and *Firefox* issues. This is evidenced by comparing our sample and the entire issue population in terms of the proportion of issue types (defects: 71.1% vs 70.1%, enhancements: 16.9% vs 16.1%, and tasks: 12% vs 13.5%), the proportion of issues per product (Core: 81.5% vs 82.7% and Firefox: 18.5% vs 17.3%), average # of comments per issue (13.4 vs 14.6), and average resolution time (81 vs 88 days). The 384 issues contain 13.4 (9) comments, 30.27 (16) paragraphs, and 56.73 (25) sentences on average (median).

B. Issue Annotation

*1) **Goals and Overview**:* We qualitatively analyzed all the information provided in the issues, annotating content related to issue resolution by employing an iterative *open coding* methodology [37]. The annotation process was conducted by six Ph.D. students and one professor (*a.k.a. annotators*), including two authors of this paper. The annotators have 1-9 years of research experience (particularly in qualitative text analysis) and five of them have 1-4 years of industry experience.

The annotation targeted all the textual content written by different stakeholders in issue comments and aimed to identify: (1) *themes* or *codes* about different activities performed to resolve the issue (*e.g.*, reproduction attempts or a code review), and (2) the types of problems described in the issues (*e.g.*, crashes, UI issues, *etc.*), which we call **problem categories**.

*2) **Annotation Tool and Unit**:* We used the Hypothesis annotation tool [38] to directly annotate the web pages of the issue reports. The tool allowed us to collaboratively assign *codes* to text snippets in the issue threads, modify the assigned codes, and discuss the annotations.

We coded *text snippets* in the issue comments. The minimal annotation unit was a complete sentence. Since one or more sentences may convey the same type of information (*i.e.*, a given resolution activity), the annotation included individual sentences, multiple sentences, paragraphs, or even entire comments. A single textual snippet was allowed to be coded with one or more codes.

*3) **Code Catalog and Coding guidelines**:* We maintained a *code catalog* via a Google spreadsheet shared among the annotators. The catalog included a list of codes, code descriptions, rules to apply the codes, and text snippets from annotated issues used as examples. The code catalog also included a list of problem categories, with detailed definitions and examples of annotated issues. We also maintained a shared Google document with detailed guidelines of the annotation procedure, coding rules, and necessary resources for annotating the issues (*e.g.*, official Mozilla documentation to get familiar with Firefox's resolution process and a glossary of annotation terminology). Both the catalog and guidelines were built from scratch and developed by all the annotators incrementally and collaboratively.

*4) **Annotation Procedure**:* We adopted an iterative multi-coder open-coding methodology wherein each issue report was annotated and validated by at least two annotators. The 384 issues were distributed evenly among the seven annotators, who iteratively examined, annotated, and validated the issue comments in batches of 30-50 issues. The first annotator assigned codes to text snippets in the comments, and a second annotator reviewed these annotations for accuracy and completeness. Discrepancies were resolved in reconciliation sessions. Annotator roles alternated across batches, with each person either annotating from scratch or reviewing the annotations by the first annotator. To avoid fatigue and reduce possible human mistakes, annotators annotated small sets of issues with breaks in between.

The overall process for a single issue involved the first annotator thoroughly reviewing the issue, including attached patches, linked commits, and metadata (*e.g.*, issue commentators, tags, and status), to identify/annotate relevant content and the problem category. Codes were assigned based on the content's meaning and the code catalog. The second annotator then reviewed these annotations, verifying their accuracy, suggesting additional codes, or flagging mistakes. After processing a batch, both annotators discussed disagreements to reach a consensus.

To establish the initial coding framework, two researchers annotated the first batch of 30 issues, creating an initial set of codes and problem categories. These were refined through discussion sessions, resulting in complete definitions, examples, and rules for applying the codes. This initial annotation informed the creation of the coding guidelines, which included resources for understanding issues and general annotation rules.

Before annotating the remaining issues, training sessions were conducted with the other annotators to review the coding guidelines, discuss examples from the initial batch, and solve misunderstandings. Throughout the entire annotation process, the code catalog was continuously updated, with changes such as new codes, code merges, or renames collectively agreed upon and promptly communicated. When the catalog was updated, previously coded issues were revisited to ensure consistency. Regular communication via Zoom meetings and Slack discussions was essential to maintain the accuracy and uniformity of the catalog and annotated content.

*5) **Annotation Results and Inter-coder Agreement**:* During the annotation process, 28 issues, that were pull requests (PR) automatically created by the issue tracker, were discarded. In summary, we annotated **3,707 textual snippets** in 2,574 issue comments across the 356 issue reports. The annotation process resulted in **22 issue resolution codes**, and **17 problem categories** which we further grouped into **3 problem classes**. Tables I and II show examples of these elements; our replication package contains the full catalog of codes and problems [9].

The annotators agreed on 3,438 annotations with an agreement rate of $\approx 93\%$ and a Cohen's kappa of 0.92, which indicates high overall agreement [39]. Common sources of disagreement (269/3,707 text snippets) included misunderstandings due to ambiguous comments or unclear code definitions. If both annotators were unable to reach an agreement, a third annotator reviewed the issue to resolve the conflict.

C. Inferring and Analyzing Issue Resolution Stages

The 22 codes obtained from the issue report annotation represent the information about activities performed by developers during issue resolution. We implemented two steps for inferring the issue resolution stages from the annotation codes. In the first step, we qualitatively analyzed the code catalog and annotated issues and identified 13 codes (*i.e.*, **actionable codes**) that signified specific actions performed to directly address the problems (*e.g.*, reproducing the problem or implementing a solution as a code change). In the second step, we engaged in an analysis of issues/codes and a discussion to categorize the 13 codes for inferring the issue resolution stages.

The first step was necessary because 9 of the 22 codes were either: (1) requests to perform an action, not an action in itself; or (2) cross-cutting actions, which can be performed at any stage of the resolution process. SOLUTION_REVIEW_REQUEST is an example of a request, which represents a petition, made by a developer to another one, to review a proposed solution to the problem. SOLVED_BY_OTHER_ISSUE is an example of a cross-cutting code that represents an issue resolved in another issue.

TABLE I: Issue Resolution Stages

Stage	Description	Annotation Codes	# of Issues
REPRODUCTION (**R**)	Developers attempt to reproduce the issue.	REP_ATT	47 (13.2%)
ANALYSIS (**A**)	Developers analyze the issue by reviewing the problem, identifying the problem cause, or locating the relevant code.	PROB_LOC, PROB_REV, CAUS_IDENT	134 (37.6%)
SOLUTION DESIGN (**SD**)	Developers discuss how to solve the issue, *i.e.*, propose a potential solution or review a proposed solution.	POT_SOL_DES, SOL_REV	150 (42.1%)
IMPLEMENTATION (**I**)	Developers make the necessary code changes to resolve the issue.	CODE_IMPL	328 (92.1%)
CODE REVIEW (**CR**)	Developers review the implemented code changes.	CODE_REV	264 (74.2%)
VERIFICATION (**V**)	Developers verify the solution by testing the implemented code changes.	SOL_VER, UPLIFT_APRV, IMPL_REV, COL_PROB_ANA, COL_POT_SOL	146 (41%)

TABLE II: Problem Categories and Classes

Problem Class	Problem Categories (Examples)	# of Categ.	# of Issues
Implementation	UI Issue, Feature Development, Crash	12	261
Refactoring	Code Improvement, Unnecessary Code Removal	2	51
Testing	Test Failure, Test Update, Flaky Tests	3	44

Based on the qualitative analysis, we identified six different issue resolution stages, namely: REPRODUCTION (**R**), ANALYSIS (**A**), SOLUTION DESIGN (**SD**), IMPLEMENTATION (**I**), CODE REVIEW (**CR**), and VERIFICATION (**V**). Each stage is represented by one to five actionable codes, each code belonging to a single stage. Examples of codes for the ANALYSIS stage (**A**) are PROBLEM_LOCALIZATION and CAUSE_IDENTIFICATION. Table I shows all the stages with their description and codes.

To answer **RQ$_1$**, we analyze the frequency in which the six stages appear in the issue reports, across different report types and problem classes and categories.

D. Analysis of Stages Sequences and Process Inference

When the identified stages are aggregated in the order in which codes appear in the issue report (*i.e.*, chronologically), they create a sequence of codes, which we can then examine to understand the process adopted to resolve the issue. For example, issue #1363344's [40] annotation code sequence is: CODE_IMPLEMENTATION, PROBLEM_REVIEW_REQUEST, CODE_REVIEW, CODE_REVIEW, CODE_REVIEW. We created a stage sequence by utilizing the code sequence and the code-stage mapping for each issue. For example, for the above code sequence of issue #1363344 [40], the derived stage sequence is: **I,CR,CR,CR** which we simplified as **I,CR** by merging the repeating consecutive stages.

To answer **RQ$_2$**, we counted the bi-grams and tri-grams appearing in the stage sequences as well as the number of issues where these n-grams appear. Bi-grams are pairs of consecutive stages, while tri-grams are triplets of consecutive stages in the sequences. We also analyzed how frequently the stages appear at the beginning or end of the sequences.

To answer **RQ$_3$**, we constructed a graph representing the overall issue resolution process, where the nodes correspond to the stages and the edges represent transitions between stages. This graph was constructed based on the most frequent bi-grams found in the sequences and serves to validate the patterns of issue resolution we derive as part of **RQ$_4$** (see Section III-E).

E. Inferring Issue Resolution Patterns

To answer RQ_4, we engaged in qualitative analysis of the stage sequences, and derived issue resolution patterns by grouping similar stage sequences into more coarse-grained sequences. The derived patterns correspond to instances of the derived issue resolution process in RQ_3.

1) Pattern Notations: To communicate the issue resolution patterns clearly and analyze them in different dimensions, we represent the patterns as a string based on three notations:

- **A?** indicates that stage A is optional;
- **(A|B)** indicates that either A or B or both stages appear;
- **(A,B,...,Z)+** indicates that stages A, B, ..., and Z appear more than once, and at least one subsequence of two or more stages (A,B or B,Z or A,B,Z, *etc.*) appears more than once.

2) Deriving Issue Resolution Patterns: At first, we identified the stage sequences where the 3rd notation, (A,B,...,Z)+, is applicable and created issue resolution patterns for those stage sequences applying the notation. For example, issue #991812 [41] with the stage sequence `I,CR,I,CR,I,CR,V,I,V` has `I`, `CR`, and `V` appearing more than once and the sub-sequence `I,CR` appears more than once. Hence, the sequence can be collapsed to create the issue resolution pattern `(I,CR,V)+`. With this notation, the order of the stages does not matter.

Second, we created groups of stage sequences that differ only by one or two stages in order and qualitatively analyzed each sequence to understand the differences among the sequences. We aim to represent the sequences using the first two notations (*i.e.*, A? and (A|B)) to form a coarse-grained sequence. For example, issues #698552 [42], #676248 [43], and #730907 [44] have the stage sequences "`SD,I,CR`", "`SD,I,CR,I`", and "`SD,I,CR,V`", respectively. Here, all three stages, `SD`, `I`, and `CR`, are included in the three issues. However, the sequences only differ by the last stage: `R` or `V` is present for the last two issues while it is not present in the first one. Hence, we can create a common pattern for these three issues, *i.e.*, `SD,I,CR,(I|V)?` which will represent all three sequences.

We meticulously created this grouping by considering several factors (*e.g.*, the # of issues per sequence, the presence of unique stages per sequence, and the issue resolution process of each issue in the group) so that we would not lose information or create any misleading sequence that does not represent the actual resolution process. For example, we could create a group of these two sequences `I` and `I,CR` by making `CR` as an optional stage in this way `I,CR?`. However, the first sequence is found for 21 issues and the second sequence is found for 50 issues which implies these two sequences are already widely used and can represent two distinct ways of issue resolution. In the first sequence, no `CR` is performed, whereas in the second, it is performed to resolve the issue. Hence, we did not create a group with these two sequences.

In all the qualitative steps, one researcher qualitatively analyzed the issue and made necessary changes by documenting the rationale behind each change which was reviewed and validated by the second researcher. Both researchers continuously discussed the patterns and solved any disagreements.

3) Pattern Derivation Results and Pattern Categorization: Our analysis resulted in 47 distinct issue resolution patterns – the 10 most frequent patterns are shown in Table III. The patterns contain 1-6 stages and appear in 1-64 issues (7.6 issues on average). The more unique stages and the more interacting stages a pattern has, the more complicated a pattern is. We argue that the complexity of a pattern reflects the effort developers invest in resolving an issue, which can be quantified by the number of stages in the sequences associated with the pattern. Therefore, we categorized the patterns as simple or complex based on the average number of stages in their sequences. Since the distribution of these averages is not skewed (see our replication package for the distribution [9]), the mean serves as a threshold for classification. Specifically, the process involves calculating the average number of stages (P_a) for each pattern, determining the overall mean across patterns ($T = 6.2$ stages), and classifying a pattern as complex if $P_a > T$ or simple if $P_a \leq T$. In Section IV-D, we discuss the pattern catalog and compare it against the derived process from RQ_3 to answer RQ_4.

F. Investigating Potential Use Cases of the Derived Patterns

To answer RQ_5, we conducted semi-structured interviews with two Mozilla developers, aimed to gather detailed feedback from them on the usefulness of the resolution patterns. The interviews were conducted over Zoom for 60 minutes and were structured into four sections:

1) Participant's Background: Participants were asked to share their background and experience in software development and issue resolution at Mozilla and other companies.
2) Mozilla's Issue Resolution Process: Participants were asked to describe Mozilla's issue resolution process (both prescribed by Mozilla and implemented by developers) as well as specific approaches they follow.
3) Research Presentation: The research team presented the study's goals, methodology, and findings, including the identified patterns. Participants were encouraged to ask questions about the patterns and findings.
4) Question-Answering: Participants were asked 11 questions to provide feedback on the identified patterns, with a focus on understanding their potential benefits for Mozilla.

Follow-up questions were asked when additional information was needed. The interview questionnaire, protocol, and anonymized responses are found in our replication package [9].

1) Finding Participants: Our target population consisted of Mozilla stakeholders with experience in issue resolution. To identify potential participants, we explored the developers' profiles from Mozilla Research [45] website, LinkedIn, Mozilla's issue tracker, Mozilla's Forum [46], and Matrix [47]. We created a shortlist of 42 potential participants, all of whom were invited to participate via email.

2) Participants Background: Two developers responded to our call and participated in the interview (*i.e.*, referred to as D1 and D2). Both are current Mozilla developers with 7 to 11 years of experience at the company. Both have extensive issue resolution experience, having resolved around 1.4K issues and contributed to approximately 19K issues in total.

3) Response Analysis: We recorded and transcribed the interviews using Zoom to facilitate response analysis. We corrected inaccuracies in the transcripts, *e.g.*, misspellings, incorrect phrases, and punctuation. Using the revised transcripts, one author analyzed and grouped the participants' answers to each question into themes representing use cases of the patterns. A second author reviewed the answers and themes for accuracy. Misinterpretations were resolved through discussion.

IV. RESULTS

A. RQ₁: Issue Resolution Stages

Table I lists the identified six issue resolution stages and reveals that not all issue reports include all stages, which indicates that Firefox developers either do not go through these stages, do not need to discuss them in the reports or discuss them in other systems or artifacts (*e.g.*, instant messaging tools). We discuss the stages starting with the most frequent ones.

1) IMPLEMENTATION: This stage is frequently performed and discussed (in 92.1% of the issues), which is expected as Firefox's issue tracker is integrated with the version control system (Mercurial). Among the 28 issues not including any IMPLEMENTATION, 25 issues were resolved in other issues, two issues were resolved by updating libraries in the host operating system, and the remaining issue was closed after more than four years of being open because the issue was no longer valid.

2) CODE REVIEW: This stage is also frequently performed and discussed (in 74.2% of the issues), which is expected as Firefox's issue tracker is integrated with Firefox's code review tools (*e.g.*, Phabricator [21]). While CODE REVIEW is frequently discussed, it is not found in 25.8% of the reports, especially in defect reports. We found that defects are the least discussed with CODE REVIEW (31.9%), compared to enhancement and task reports (6.7% and 7.7%). Phabricator [21], adopted in 2019, replaced MozReview [48] and Splinter [49] and became Firefox's only code review tool. Analysis shows that 35% of defects resolved on or before 2019 lacked a CODE REVIEW, compared to only 17% after 2019. We found that post-2018 issues without CODE REVIEW do not include code changes, as the issues were resolved in other issues.

3) VERIFICATION: This stage, covering manual and automatic testing, appears in only 41% of issues. It is less common in task reports (26.9%) than in defect and enhancement reports (42.2% and 41.7%). Refactoring and testing issues include VERIFICATION less often (19.6% and 31.8%) compared to implementation-related issues (46.7%). Categories like Code Improvement, Test Failure, Code Design, and Performance Optimization have few VERIFICATION discussions, while Crashes (65%), Feature Dev. (61.5%), and UI Issues (63.6%) show higher inclusion. This indicates still low VERIFICATION discussions in the issues and this is consistently found every year.

4) ANALYSIS and SOLUTION DESIGN: The ANALYSIS and SOLUTION DESIGN stages are infrequently discussed, appearing in only 37.6% and 42.1% of issues, respectively. Defects are analyzed more frequently (*i.e.*, ANALYSIS in 44.8% of defects) compared to enhancements (20%) and tasks (3.9%), with Crashes, Flaky Tests, Incorrect Page Renderings,

and Test Updates being the most analyzed defects. SOLUTION DESIGN is more common in enhancements (42.2%) and defects (43.3%) than tasks (38.5%). Refactoring issues are the least analyzed (7.8%) compared to implementation (42.2%) and testing-related issues (45.5%).

5) REPRODUCTION: REPRODUCTION is the least frequent stage, appearing in only 13.2% of issues, with just 17% of defects including it. This suggests that Firefox stakeholders rarely discuss bug reproduction in issue reports. We observed that REPRODUCTION is included when bugs are hard to reproduce or when reproduction is necessary to identify the root cause or localize the bug. The first scenario leads typically to more effort in solving the issues: compared to defects with no REPRODUCTION, defects with REPRODUCTION take longer to resolve (avg/med: 82.9/5.5 vs. 79.3/18.5 days) and involve more commentators (avg/med: 4.9/4 vs. 7.7/7). This is validated by a Mann-Whitney U test [50] with $\alpha = 0.05$, with p-values nearly 0. The 2nd scenario is supported by the data: 73% of the issues with REPRODUCTION include an ANALYSIS.

> **RQ₁ Findings**: The six stages of issue resolution found in Firefox issue reports appear with varying frequency across different issue reports and problem categories. IMPLEMENTATION and CODE REVIEW are the most frequent stages while REPRODUCTION is the least frequent.

B. RQ₂: Interactions between Issue Resolution Stages

We examined the 356 stage sequences obtained in Section III-D by analyzing the frequency of stage bi-grams and tri-grams in the sequences. Bi-grams are pairs of consecutive stages (**S,T**) in a sequence, which represent possible stage transitions (**S**→**T**) in the resolution process. In our data, we found all possible bi-grams between stages, except for **CR**→**R**, and five extremely rare transitions (appearing only once or twice): **I**→**R**, **V**→**R**, **R**→**CR**, and **R**→**V**. All these transitions include REPRODUCTION (**R**) as the source or target stage.

Of the 1,430 bi-grams found in the sequences, nine are the most recurrent, covering 80.6% of the bi-gram occurrences: **I**→**CR** (403 cases), **CR**→**I** (187), **SD**→**I** (133), **CR**→**V** (94), **I**→**V** (86), **V**→**I** (73), **A**→**I** (70), **A**→**SD** (65), and **CR**→**SD** (41). From these transitions, we observe the interplay among IMPLEMENTATION, CODE REVIEW, and VERIFICATION, in which IMPLEMENTATION undergoes quality assurance activities and these also lead to additional code changes (or to SOLUTION DESIGN – see **CR**→**SD** above). SOLUTION DESIGN (**SD**) can lead to code changes, and ANALYSIS (**A**) can result in code changes or solution design activities. Notably, while **A**→**R** and **R**→**A** are not among the most frequent bi-grams, they appear in 73.5% and 70.6% of the issues containing both stages (34). This further supports our finding that ANALYSIS and REPRODUCTION typically occur together.

The analysis of tri-grams, sets of three consecutive stages in a sequence (**S**→**T**→**U**), not only provides extra evidence of the interplay among IMPLEMENTATION, CODE REVIEW, and VERIFICATION, but also the relationship among ANALYSIS, SOLUTION DESIGN, IMPLEMENTATION, and CODE REVIEW.
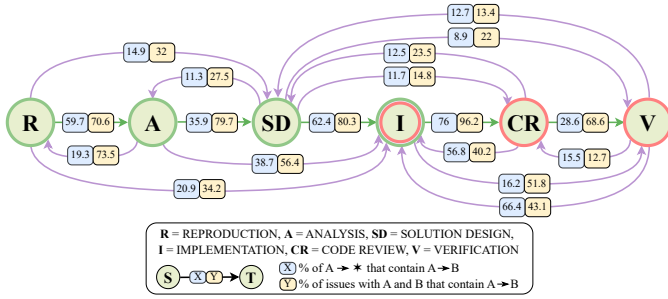
Fig. 1: Overall Issue Resolution Process of Firefox

Of 1,109 tri-grams found in the sequences, 25 are the most frequent, covering 80.6% of the tri-grams, with the following three being the most frequent: **I**→**CR**→**I** (168 occurrences in 97 issues), **CR**→**I**→**CR** (114 occurrences in 61 issues), and **SD**→**I**→**CR** (85 occurrences in 76 issues).

> **RQ₂ Findings**: Firefox's developers switch among different resolution stages to solve issues. Stage bi-gram and tri-gram analysis reveal that developers frequently engage in three scenarios: 1) reproducing the issues (**R**) along with issue analysis (**A**) to confirm the issues and reason about them; 2) analyzing the issues (**A**) along with solution design (**SD**), and then engaging in implementation (**I**) and code reviews (**CR**); and 3) inspecting (**CR**) and verifying the implemented solution (**V**), adapting the implementation when needed (**I**).

C. *RQ₃: Issue Resolution Process*

Figure 1 shows the overall issue resolution process at Firefox, derived from the bi-gram analysis we performed on the 356 stage sequences. The process is a directed graph in which nodes represent the six resolution stages (*e.g.*, IMPLEMENTATION or **I**) and the edges represent transitions between stages (*e.g.*, IMPLEMENTATION→CODE REVIEW or **I**→**CR**). The nodes with green and red border are initial and end nodes, selected from the most frequent initial and end stages in the sequences. All nodes imply a loop to itself, indicating the stage can be performed multiple times in a row.

The process includes only the most frequent bi-grams **S** → **T**, selected based on the *proportion of bi-grams* starting with **S** (**S**→**∗**) that contain **S**→**T**. This proportion is shown in the blue boxes of the edges in Figure 1. For example, **I**→**CR** has a frequency of 403/530 = 76% since from all 530 bigrams that start with **I** (**I**→**∗**) there are 403 occurrences of **I**→**CR**. The frequencies of all the transitions starting from a given node add up to at least 90%. For example, all transitions coming out of **I** add up to 92.3%. The yellow boxes of the edges represent the *proportion of issues* containing **S** and **T** that contain the bigram **S**→**T**. For example, 254 of 264 (96.2%) issues with both **I** and **CR** contain **I**→**CR**.

We make two observations from the process in Figure 1:
- The process deviates from the (theoretical) linear process outlined by the existing literature and Firefox's documentation (see Sections II-B and II-D). Instead of a linear sequence of stages (starting with **R** and going through every stage from left to right until VERIFICATION and/or CODE REVIEW are completed — see the path with green transitions in Figure 1), the process is more complicated than expected as it includes iterative interactions between stages. This means developers go back and forth from one stage to another, forming different workflows of issue resolution.
- Some nodes have a high number of incoming and outgoing transitions, indicating the level of importance of such stages in the process. Specifically, SOLUTION DESIGN has four incoming and four outgoing transitions, and IMPLEMENTATION has five incoming and two outgoing transitions. These stages are pivotal because they allow for stage switches from and to many of the other stages. CODE REVIEW and VERIFICATION are the second most important stages, both having three incoming and three outgoing transitions, while ANALYSIS and REPRODUCTION are less important with fewer transitions.

> **RQ₃ Findings**: Firefox's issue resolution follows an iterative process that deviates from the (theoretical) linear process. In this process, developers go back and forth from one state to another as needed to solve the issues. SOLUTION DESIGN and IMPLEMENTATION, followed by CODE REVIEW and VERIFICATION, play a key role as they are the source and target for most of the other stages.

D. *RQ₄: Issue Resolution Patterns*

Figure 1 shows that Firefox's issue resolution process differs from the expected linear process from prior work. However, the figure does not show how much the process differs and the different instances of the process that developers follow.

Employing the qualitative approach described in Section III-E, we identified 47 distinct instances of the process, which we call patterns of issue resolution. These patterns appear in the 356 issue reports with varying frequency and complexity: a pattern appears in 1 to 64 reports (7.6 on avg, 4 median), 20 patterns are categorized as complex (they imply high issue resolution effort), and 27 as simple (they imply low resolution effort). Of all the patterns, 18 patterns are the most recurrent: they are found in 5 to 64 reports (16 on avg., 9.5 median), covering 287 reports (80.6%); 12 are simple, and 6 are complex. For space reasons, Table III shows the 10 most recurrent patterns, but the entire pattern catalog can be found in our replication package [9].

1) Pattern Examples: We describe two patterns of different kinds to illustrate different workflows of issue resolution.

The pattern **A,SD,I,(I|CR|V)?** represents the process in which developers first analyze the reported problem (**A**). They then design the solution (**SD**) (*e.g.*, propose a potential solution or review a proposed solution), and then implement the solution (**I**). Developers may then review the code (**CR**) and/or test the code changes to verify if they solved the issue (**V**). Based on the QA feedback, more code changes may occur (**I**). This pattern is *simple* because it includes only three mandatory stages (**A,SD,I**) followed by three optional stages (**(I|CR|V)?**).

The pattern **(SD,I,(CR|V))+** suggests a process in which SOLUTION DESIGN, IMPLEMENTATION, CODE REVIEW, and/or

TABLE III: Top 10 Frequent Issue Resolution Patterns

Pattern	Description	Com-plexity	# of Issues
I,CR,I?	Implement the solution and review the code; followed by another optional implementation.	Simple	64
A,I,(I|CR|V)?	Analyze the problem and implement the solution; followed by another optional I or CR or V or any combination.	Simple	32
(I,(CR|V))+	Implement the solution; review the code and/or verify the implementation; I, CR and/or V repeat more than once.	Complex	28
SD,I,CR,(I|V)?	Design and implement the solution and review the code; followed by another optional I or V.	Simple	24
A,SD,I,(I|CR|V)+	Analyze the problem, design, and implement the solution; followed by another optional I or CR or V or any combination.	Simple	22
I	Implement the solution.	Simple	21
I,CR,V,I?	Implement the solution, review the code, and verify the implementation; followed by another optional I.	Simple	16
SD,(I,(CR|V))+	Design the solution; implement the solution, review code and/or verify the implementation; I, CR and/or V repeat more than once.	Complex	13
(SD,I,(CR|V))+	Design and implement the solution; review the code, and/or verify the implementation; SD,I, CR and/or V repeat more than once.	Complex	12
A,(I,(CR|V))+	Analyze the problem; implement the solution, review code, and/or verify the implementation; I, CR and/or V repeat more than once.	Complex	7

R=REPRODUCTION, **A**=ANALYSIS, **SD**=SOLUTION DESIGN,
I=IMPLEMENTATION, **CR**=CODE REVIEW, **V**=VERIFICATION

TABLE IV: Number of Issues Across Issue Types

Issue Type	Pattern Complexity		Total
	Complex	Simple	
Defect	79	191	**270**
Enhancement	19	41	**60**
Task	6	20	**26**
Total	**104**	**252**	**356**

VERIFICATION are performed repetitively to solve the issue. In the repetitive series **(CR|V)** means either one or both can appear after a **SD** and a **I**. To resolve issues with this pattern, developers need to perform four distinct stages where all stages are repetitive, making this pattern *complex*.

2) Process and Pattern Diversity: All the identified patterns represent instances of the issue resolution process. The 47 instances indicate a wide variety of ways to solve issues at Firefox. While more generalized patterns can be formed from the 47 patterns, our qualitative approach carefully identified the patterns to accurately reflect the observed process from the issues. We did not forcefully merge patterns into more general ones but did validate the patterns against the process from Figure 1 (which was derived quantitatively).

The diversity of the patterns/process is observed across problem categories. Six of the 17 problem categories have more unique patterns (14 to 23) than the remaining 11 categories (1 to 11 patterns). These six categories are: Defective Functionality (23 unique patterns found in 43 issues), Code Design (21 patterns in 75 issues), UI Issue (22 patterns in 33 issues), Test Failure (17 patterns in 17 issues), Crash (17 patterns in 17 issues), and Feature Development (14 patterns in 39 issues).

Issue resolution for some categories is more diverse than for other categories, despite having a similar number of issues. For example, UI Issues are solved with 22 patterns, and Code Improvement issues are solved with 11 patterns, despite both categories covering 32-33 issues. We also found that the six most frequent patterns shown in Table III were used to resolve issues of more than half of the categories (9-11 of 17 categories). This illustrates that the same issue resolution pattern can solve problems of different kinds.

The diversity of the patterns/process is also observed throughout the lifespan of Firefox, from 2010 to 2023. During the 14-year time span, the five most frequent patterns (found in 48% of the issues) were observed in 11 to 14 different years. The 10 most frequent patterns (found in 67% of the issues) are found in 7 to 14 different years. All the 39 patterns appearing

in two or more issues were utilized in 2 to 14 years.

3) Pattern Complexity and Resolution Effort: Table IV shows that 70.8% of the issues (252 of 356) are solved with the 27 simple patterns. These, compared to issues solved with a complex pattern, are solved faster (avg/med: 58/5 vs. 119.8/19.5 days), require fewer stages in the process (avg/med: 2.9/3 vs. 9.9/9), and include fewer commentators (avg/med: 4.4/4 vs. 7.4/7). A Mann–Whitney U test [50] (at $\alpha = 0.05$) confirmed these differences across all these factors with *p-value* = 0.0.

While all the problem kinds are solved with simple patterns in most of the cases (55.8% - 85.1.7% of the issues), 5 of 17 categories tend to have more issues solved with complex patterns (104 of 356 = 29.2%), compared to the other 12 categories. These five categories are: Code Design (22 of 75 issues are solved with a complex pattern), Defective Functionality (19 of 43), Feature Development (13 of 39), UI Issue (14 of 33), and Crash (8 of 23). This indicates that these categories contain issues that require more effort to be solved. Despite the issues in these categories being solved with similar resolution time (avg/med: 72.4/9 vs 81.6/6 days), they include more stages in their process (avg/med: 5.2/4 vs. 4/3) and more commentators (avg/med: 5.8/5 vs. 4.4/4) with statistical significance (Mann–Whitney U test, *p-value* = 0.0), suggesting potentially more resolution effort.

Table IV also reveals that 76.9% of the tasks are solved using a simple pattern whereas 70.7% of the defects and 68.3% of the enhancements are solved with a simple pattern. As pattern complexity suggests, compared to defects and enhancements, tasks require less effort: they are solved significantly faster (avg/med: 9/4 vs. 82.3/7 and 76.9/11.5), require fewer process stages (avg/med: 3.6/2 vs. 5.2/4 and 5/4), and include fewer commentators (avg/med: 4/3 vs. 5.4/4 and 5.3/5). These results are statistically significant, according to the Mann–Whitney U test (p-values = 0.02, 0.03, and 0.002, respectively.)

> **RQ4 Findings**: The 47 identified issue resolution patterns indicate that solving issues at Firefox is done in a wide variety of ways. Of these, 18 patterns are recurrently found in 80.6% of Firefox issue reports. The process of issue resolution in Firefox is diverse and far from linear. The diverse and iterative nature of the process is consistently observed throughout Firefox's 14 years of evolution.

E. RQ5: Use Cases for the Issue Resolution Patterns

The two interviewed Mozilla developers (*i.e.*, D1 and D2) highlighted the following use cases for the derived patterns:

1) Identifying Issues with a Complex Resolution: D1 and D2 suggested that the patterns could help detect issues with

complex resolutions, especially those involving repetitive stages, which may signal excessive time and effort spent by developers. D1 noted that such patterns could highlight when a bug takes an unexpectedly complex path, explaining, "If you went through three different implementations and three different verifications and it still didn't work, something went wrong here." D2 also emphasized the value of a tool that identifies these complex issues, saying, "It might be interesting to have some sort of tool that watches the bugs and when it sees this snowball effect... it could alert a product person that this bug is chewing up a lot of time." Both developers agreed on the importance of early detection of complex resolutions. Both emphasized that detecting these issues would help understand why a process is taking longer than expected, allowing for timely corrections.

2) Identifying Issues not Following the Expected Process: D1 suggested that a tool could be useful for detecting issues that deviate from expected workflows, especially those requiring human verification. He explained, "Some bugs require human verification... it requires installing third-party software on a machine. [...] And it's up to developers to highlight when this is the case." D1 emphasized that a tool could help by alerting developers when an issue seems to need third-party support for verification, stating, "If you had a tool that said, hey, this bug that you open looks like it might need some third-party support for verification, that might actually be a helpful thing."

3) Identifying Potentially Complex Code Components: D2 suggested that tracking the complexity of issue resolution in specific Mozilla Firefox code components or modules could reveal underlying quality issues, such as technical debt or accumulated code complexity. As D2 put it, "You could track the complexity of the issue resolution process in a given module... and get insights like, hey, it looks like most of the time when you touch this area of code it ends up being a slog." D2 emphasized that such insights could signal the need for refactoring, stating, "Maybe it's time to refactor this? Maybe it's time to clean this up... this part of the code base is a tar pit and we probably want to spend some resources making it less ornery."

4) Improving Bots to Detect Unsolvable Issues: D2 suggested that the resolution patterns could help improve the heuristics of existing bots (*e.g.*, bugbug [51]) used to process Mozilla issues, enabling them to better identify issues that are unsolvable or particularly challenging to solve. As D2 explained, "If you could identify signs that this bug is not going to be solved or is about to fall through the cracks, that would be pretty cool." He noted that current bots already attempt to detect when issues have "fallen through the cracks" and need attention, saying, "We have some bots that do that kind of work."

5) Suggesting and Decomposing Meta-Issues: Both D1 and D2 suggested that issues with complex resolutions, as indicated by the complex patterns, might represent meta-issues: large issues that could be broken down into smaller, more manageable issues. They proposed that a tool capable of flagging these cases and suggesting possible decompositions would be highly beneficial. As D1 explained, "We have this idea of a meta bug, which is a bug which hosts a whole bunch of related bugs," and suggested the tool could flag such cases and offer ideas

like, "could this be split? Here are some topics that it sounds like you could split this down into."

6) Training Junior Developers: D1 and D2 both highlighted that the patterns could serve as valuable training tools for junior Mozilla developers, offering insights into the practical aspects of issue resolution. D1 noted that junior developers often have high expectations and approach issues linearly, seeking a perfect solution. However, D1 emphasized that the patterns show the issue-resolution process is typically incremental and iterative, involving multiple cycles of code review and verification. As D1 explained, "You will probably have to iterate... you will probably have to go through this solution more than once. And that's okay. That's expected. It's part of the job."

Both developers suggested that automation is needed to realize such use cases, particularly tools that identify patterns in issue discussions and classify them as simple or complex. Our future work will develop such tools to automate the identification of textual content in issue comments (*e.g.*, issue resolution activities) and use algorithms to derive sequences of stages and patterns. This process will likely combine machine learning with heuristic-based approaches.

> **RQ$_5$ Findings**: The interviewed Mozilla developers suggested that the resolution patterns could help identify complex issues, workflow deviations, and low-quality code components, improve bots for detecting unsolvable issues, decompose large issues, and train junior developers.

V. DISCUSSION AND IMPLICATIONS

Firefox's Issue Resolution in Practice. Our study highlights the iterative and diverse nature of Firefox's issue resolution process, which widely deviates from the theoretical linear models often assumed in the literature (*e.g.*, Rajlich's incremental change process [7]). Instead of following a straightforward path, developers address various types of issues by moving back and forth through multiple stages as needed. This reflects the iterative and incremental approach characteristic of modern software development, aligning more closely with agile methodologies than with rigid frameworks like Waterfall [7].

Patterns Generalizability. While the results only apply to Firefox, we conducted a small case study that annotated 20 issue reports from two open-source projects: Chromium [52] and GnuCash [53]. The goal was to validate if these projects follow resolution patterns similar to Firefox's. Details of the study methodology are found in our replication package [9].

We identified seven distinct resolution patterns for the 10 Chromium issue reports, all of which correspond to Firefox's patterns. Three patterns, `I,CR`, `I,CR,V`, and `SD,I,CR,V` appeared in two issues each, aligning with Firefox patterns `I,CR,I?`, `I,CR,V,I?`, and `SD,I,CR,(I|V)` which appeared in 64, 16, and 24 issues respectively. Notably, these Firefox patterns are among the top seven most recurrent patterns, which strengthens pattern generalizability (they are found in Chromium issues). As for GnuCash, we identified 10 resolution patterns across the 10 issues, with nine of these patterns aligning with nine of the 47

Firefox patterns. The nine Firefox patterns are fairly common as they were observed in 4 to 22 issues. The GnuCash pattern `'(A,I)+'` does not have any corresponding Firefox pattern.

The 7 and 10 patterns identified for Chromium and GnuCash indicate that developers in these projects also employ diverse approaches to issue resolution. This suggests that some Firefox patterns may generalize across projects of varying scales and governance. However, a large-scale study with a statistically significant sample is needed to confirm these observations.

VI. Threats to Validity

Construct and Internal Validity. Relying solely on issue reports poses a validity threat. Issue discussions may not capture all of Firefox's resolution activities, either because certain actions do not require documentation or were discussed/recorded in other artifacts or channels. This limitation may explain why some stages (*e.g.*, issue reproduction) are absent in certain issues. Consequently, the derived patterns should be interpreted with caution, as they reflect the resolution process *documented in issue reports*, which may differ from the practical process. However, according to Firefox's documentation [22], issue reports are the primary artifacts for tracking Firefox changes, and developers are strongly encouraged to document relevant problem information within them. Moreover, we are confident in the accuracy of traces for implementation, code review, and verification stages, due to the tool integration with the issue tracker, as well as the requirement for verification to mark an issue as "VERIFIED." This provides confidence that issue report discussions capture the implemented resolution process.

Researcher subjectivity and potential confirmation bias introduced during issue coding, resolution pattern inference, and results interpretation represent key validity threats. To address these, we implemented a rigorous open-coding methodology involving multiple coding phases. Each issue report was reviewed multiple times, accompanied by discussion sessions between annotators. Both annotators critically annotated and verified the data at each phase, resolving disagreements through consensus. The results interpretation was thoroughly discussed and supported by data-driven evidence.

External Validity. Our pattern catalog and results may not generalize to all issues from Firefox and to other systems, as is typical in case studies. To strengthen generalization, our study analyzed a statistically significant sample, in which the distribution of coded issues resembles that of all Firefox issues. We also annotated 20 issue reports of Chromium and GnuCash, and found that some of the most frequent Firefox patterns cover the resolution workflows found in the 20 issues, which implies that at least some of the derived patterns can be generalized to these projects. While the results are indicative, in-depth studies are needed to confirm these results and establish generalizability.

VII. Related Work

Researchers have proposed a variety of techniques to address issue management challenges and automate several tasks in the process [54]. For example, researchers have proposed automated techniques to better report issues [55, 56], predict the priority and severity of the issues [57–59], categorize issue types [60, 61], assign developers to issues [15, 62], suggest potential duplicate issues [63–65], reproduce buggy behavior [66, 66–69], localize buggy code files [70–80], and predict re-opened issues [81, 82].

Researchers have studied issue reports for a variety of purposes: to understand decision-making [83] and the discourse used to describe issues [84]; extract decision information [85]; understand stakeholders' information needs [86]; characterize/predict different kinds of issues such as *won't fix* issues [87], fixed/resolved issues [88], non-reproducible bugs [89], and bug/issue types [61, 90, 91]; predict issue severity [92]; understand workarounds [93] and visual content in issues [94]; questions [95], and information types in issues [96].

Researchers have used automated mining of issue data (*e.g.*, status changes) and version control/code review data to identify development processes and assess delays and inconsistencies [97, 98]. They have utilized process mining techniques to integrate data from different sources (*e.g.*, VCS, issue trackers, and mail archives) [99–101] and proposed process mining techniques [102–104] to gain insights about the development processes. Other work has studied the life cycle of issues by mining and analyzing issue state transitions [8, 105–109]. These works focused more broadly on issue management and identified transitions of issue states (*e.g.*, from "Assigned" to "In progress" to "Closed"). However, issue states are often too broad to provide detailed insights into how stakeholders resolve issues in practice.

Unlike prior work, our research qualitatively analyzed issue reports to identify resolution stages, develop a process model, and uncover detailed patterns of issue resolution at Firefox. This in-depth analysis led to new insights into the issue resolution process. To our knowledge, we are the first to examine how the issue resolution process is actually implemented and discussed in practice, and how it differs from the theoretical models found in the literature.

VIII. Conclusions

We conducted a case study to understand the process employed by Mozilla Firefox developers to solve software issues. By implementing a multi-coder open-coding methodology, we qualitatively analyzed the issue report comments, identified six issue resolution stages, and derived an overall process model. We found 47 issue resolution patterns which are instances of the process and represent how Firefox developers resolve issues in practice. This process is iterative and widely deviates from the theoretical linear process from the literature.

Acknowledgements

REFERENCES

[1] T. Zimmermann, R. Premraj, J. Sillito, and S. Breu, "Improving bug tracking systems," in *ICSE'09*, 2009, pp. 247–250.

[2] T. Zimmermann, R. Premraj, N. Bettenburg, S. Just, A. Schröter, and C. Weiss, "What Makes a Good Bug Report?" *TSE*, vol. 36, no. 5, pp. 618–643, 2010.

[3] "Firefox's bug handling documentation," https://firefox-source-docs.mozilla.org/bug-mgmt/index.html, 2024.

[4] T. Zhang, H. Jiang, X. Luo, and A. T. Chan, "A literature review of research in bug resolution: Tasks, challenges and future directions," *The Computer Journal*, vol. 59, no. 5, pp. 741–773, 2016.

[5] R. K. Saha, S. Khurshid, and D. E. Perry, "Understanding the triaging and fixing processes of long lived bugs," *Information and software technology*, vol. 65, pp. 114–128, 2015.

[6] A. Zeller, *Why programs fail: a guide to systematic debugging*. Elsevier, 2009.

[7] V. Rajlich, *Software engineering: The current practice*. Crc Press, 2011.

[8] Ç. Eren, K. Şahin, and E. Tüzün, "Analyzing bug life cycles to derive practical insights," in *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering*, 2023, pp. 162–171.

[9] "Online replication package," https://tinyurl.com/repl-pack, 2024.

[10] "Bugzilla," https://www.bugzilla.org/, 2024.

[11] "Jira," https://www.atlassian.com/software/jira, 2024.

[12] "Github," https://github.com/features/issues, 2024.

[13] D. Bertram, "The social nature of issue tracking in software engineering," *University of Calgary*, 2009.

[14] D. Bertram, A. Voida, S. Greenberg, and R. Walker, "Communication, collaboration, and bugs: the social nature of issue tracking in small, collocated teams," in *Proceedings of the 2010 ACM conference on Computer supported cooperative work*, 2010, pp. 291–300.

[15] X. Xia, D. Lo, X. Wang, and B. Zhou, "Accurate developer recommendation for bug resolution," in *2013 20th Working Conference on Reverse Engineering (WCRE)*. IEEE, 2013, pp. 72–81.

[16] "The life cycle of a bug in bugzilla," https://www.bugzilla.org/docs/2.18/html/lifecycle.html, 2024.

[17] "Firefox browsers," https://www.mozilla.org/en-US/firefox/, 2024.

[18] "Firefox bug #1029919," https://bugzilla.mozilla.org/show_bug.cgi?id=1029919, 2024.

[19] "Firefox bug #1718748," https://bugzilla.mozilla.org/show_bug.cgi?id=1718748, 2024.

[20] "mozregression," https://mozilla.github.io/mozregression/, 2024.

[21] "Phabricator," https://phabricator.services.mozilla.com/, 2024.

[22] "Firefox: how to submit a patch," https://firefox-source-docs.mozilla.org/contributing/how_to_submit_a_patch.htm, 2024.

[23] "Firefox: Code quality," https://firefox-source-docs.mozilla.org/code-quality/index.html, 2024.

[24] "Firefox: Reviewer checklist," https://firefox-source-docs.mozilla.org/contributing/reviewer_checklist.html, 2024.

[25] "Firefox: Fixing security bugs," https://firefox-source-docs.mozilla.org/bug-mgmt/processes/fixing-security-bugs.html, 2024.

[26] "Working on firefox," https://firefox-source-docs.mozilla.org/contributing/index.html, 2024.

[27] "Firefox's bug pipeline documentation," https://tinyurl.com/2up6wjp3, 2024.

[28] "Mozilla's bug types," https://firefox-source-docs.mozilla.org/bug-mgmt/guides/bug-types.html, 2024.

[29] "Firefox's bug triage," https://firefox-source-docs.mozilla.org/bug-mgmt/policies/triage-bugzilla.html, 2024.

[30] "Firefox's new feature triage," https://firefox-source-docs.mozilla.org/bug-mgmt/policies/new-feature-triage.html, 2024.

[31] "Firefox: Approving security bugs," https://firefox-source-docs.mozilla.org/bug-mgmt/processes/security-approval.html, 2024.

[32] "Pushing to try," https://firefox-source-docs.mozilla.org/tools/try/index.html, 2024.

[33] A. Halberstadt and M. Castelluccio, "Testing firefox more efficiently with machine learning," https://hacks.mozilla.org/2020/07/testing-firefox-more-efficiently-with-machine-learning/, 2020.

[34] "Pocket guide: Shipping firefox," https://firefox-source-docs.mozilla.org/contributing/pocket-guide-shipping-firefox.html, 2024.

[35] "Mozilla products in bmo," https://bugzilla.mozilla.org/describecomponents.cgi, 2024.

[36] "Bugzilla's rest api," https://wiki.mozilla.org/Bugzilla:REST_API, 2024.

[37] D. Spencer, *Card sorting: Designing usable categories*. Rosenfeld Media, 2009.

[38] "The hypothesis web annotation tool," https://web.hypothes.is, 2024.

[39] J. Cohen, "A coefficient of agreement for nominal scales," *Educational and psychological measurement*, vol. 20, no. 1, pp. 37–46, 1960.

[40] "Firefox bug #1363344," https://tinyurl.com/mr3d7h6e, 2023.

[41] "Firefox bug #991812," https://bugzilla.mozilla.org/show_bug.cgi?id=991812, 2024.

[42] "Firefox bug #698552," https://bugzilla.mozilla.org/show_bug.cgi?id=698552, 2024.

[43] "Firefox bug #676248," https://bugzilla.mozilla.org/show_bug.cgi?id=676248, 2024.

[44] "Firefox bug #730907," https://bugzilla.mozilla.org/show_bug.cgi?id=730907, 2024.

[45] "https://research.mozilla.org/," 2024.

[46] "https://www.mozilla.org/en-US/about/forums/," 2024.

[47] "https://wiki.mozilla.org/Matrix," 2024.

[48] "Mozreview," https://wiki.mozilla.org/EngineeringProductivity/Projects/MozReview, 2024.

[49] "Splinter," https://wiki.mozilla.org/BMO/Splinter, 2024.

[50] P. E. McKnight and J. Najab, "Mann-whitney u test," *The Corsini encyclopedia of psychology*, pp. 1–1, 2010.

[51] "https://github.com/mozilla/bugbug," 2024.

[52] "Chromium," https://www.chromium.org/Home/, 2024.

[53] "Gnucash," https://www.gnucash.org/, 2024.

[54] W. Zou, D. Lo, Z. Chen, X. Xia, Y. Feng, and B. Xu, "How practitioners perceive automated bug report management techniques," *IEEE Transactions on Software Engineering*, vol. 46, no. 8, pp. 836–862, 2018.

[55] Y. Song, J. Mahmud, Y. Zhou, O. Chaparro, K. Moran, A. Marcus, and D. Poshyvanyk, "Toward interactive bug reporting for (Android app) end-users," in *FSE'22*, 2022.

[56] M. Fazzini, K. P. Moran, C. Bernal-Cardenas, T. Wendland, A. Orso, and D. Poshyvanyk, "Enhancing mobile app bug reporting via real-time understanding of reproduction steps," *TSE22*, 2022.

[57] Q. Umer, H. Liu, and I. Illahi, "Cnn-based automatic prioritization of bug reports," *IEEE Transactions on Reliability*, vol. 69, no. 4, pp. 1341–1354, 2019.

[58] Y. Tian, D. Lo, X. Xia, and C. Sun, "Automated prediction of bug report priority using multi-factor analysis," *Empirical Software Engineering*, vol. 20, pp. 1354–1383, 2015.

[59] Z. Huang, Z. Shao, G. Fan, H. Yu, K. Yang, and Z. Zhou, "Bug report priority prediction using developer-oriented socio-technical features," in *Proceedings of the 13th Asia-Pacific symposium on internetware*, 2022, pp. 202–211.

[60] K. Somasundaram and G. C. Murphy, "Automatic categorization of bug reports using latent dirichlet allocation," in *Proceedings of the 5th India software engineering conference*, 2012, pp. 125–130.

[61] G. Catolino, F. Palomba, A. Zaidman, and F. Ferrucci, "Not all bugs are the same: Understanding, characterizing, and classifying bug types," *JSS*, vol. 152, pp. 165–181, 2019.

[62] B. Chaitra and K. Swarnalatha, "Bug triaging: right developer recommendation for bug resolution using data mining technique," in *Emerging Research in Computing, Information, Communication and Applications: ERCICA 2020, Volume 2*. Springer, 2022, pp. 609–618.

[63] J. Zhou and H. Zhang, "Learning to rank duplicate bug reports," in *CIKM'12*, 2012, pp. 852–861.

[64] J. He, L. Xu, M. Yan, X. Xia, and Y. Lei, "Duplicate bug report detection using dual-channel convolutional neural networks," in *Proceedings of the 28th International Conference on Program Comprehension*, 2020, pp. 117–127.

[65] T. Zhang, D. Han, V. Vinayakarao, I. C. Irsan, B. Xu, F. Thung, D. Lo, and L. Jiang, "Duplicate bug report detection: How far are we?" *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 4, pp. 1–32, 2023.

[66] Y. Zhao, T. Yu, T. Su, Y. Liu, W. Zheng, J. Zhang, and W. G. Halfond, "Recdroid: Automatically reproducing android application crashes from bug reports," in *ICSE'19*, 2019, pp. 128–139.

[67] S. Feng and C. Chen, "Gifdroid: an automated light-weight tool for replaying visual bug reports," in *ICSE'22*, 2022.

[68] Z. Zhang, R. Winn, Y. Zhao, T. Yu, and W. G. Halfond, "Automatically reproducing android bug reports using natural language processing and reinforcement learning," *arXiv preprint arXiv:2301.07775*, 2023.

[69] Y. Zhao, T. Su, Y. Liu, W. Zheng, X. Wu, R. Kavuluru, W. G. Halfond, and T. Yu, "Recdroid+: Automated end-to-end crash reproduction from bug reports for android apps," *TOSEM'22*, vol. 31, no. 3, pp. 1–33, 2022.

[70] S. A. Akbar and A. C. Kak, "A large-scale comparative evaluation of ir-based tools for bug localization," in *MSR'20*, 2020, pp. 21–31.

[71] J. Lee, D. Kim, T. F. Bissyandé, W. Jung, and Y. Le Traon, "Bench4bl: reproducibility study on the performance of ir-based bug localization," in *ISSTA'18*, 2018, pp. 61–72.

[72] X. Ye, H. Shen, X. Ma, R. Bunescu, and C. Liu, "From word embeddings to document similarities for improved information retrieval in software engineering," in *ICSE'16*, 2016, pp. 404–415.

[73] A. Ciborowska and K. Damevski, "Fast changeset-based bug localization with bert," in *ICSE'22*, 2022, pp. 946–957.

[74] C.-P. Wong, Y. Xiong, H. Zhang, D. Hao, L. Zhang, and H. Mei, "Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis," in *ICSME'14*, 2014, pp. 181–190.

[75] P. S. Kochhar, Y. Tian, and D. Lo, "Potential biases in bug localization: Do they matter?" in *ASE'14*, 2014, pp. 803–814.

[76] J. M. Florez, O. Chaparro, C. Treude, and A. Marcus, "Combining query reduction and expansion for text-retrieval-based bug localization," in *SANER'21*. IEEE, 2021, pp. 166–176.

[77] O. Chaparro, J. M. Florez, and A. Marcus, "Using bug descriptions to reformulate queries during text-retrieval-based bug localization," *ESE'19*, vol. 24, pp. 2947–3007, 2019.

[78] ——, "Using observed behavior to reformulate queries during text retrieval-based bug localization," in *ICSME'17*, 2017.

[79] O. Chaparro, J. M. Florez, U. Singh, and A. Marcus, "Reformulating queries for duplicate bug report detection," in *SANER'19*. IEEE, 2019, pp. 218–229.

[80] A. Saha, Y. Song, J. Mahmud, Y. Zhou, K. Moran, and O. Chaparro, "Toward the automated localization of buggy mobile app uis from bug descriptions," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 1249–1261.

[81] T. Zimmermann, N. Nagappan, P. J. Guo, and B. Murphy, "Characterizing and predicting which bugs get reopened," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 1074–1083.

[82] E. Shihab, A. Ihara, Y. Kamei, W. M. Ibrahim, M. Ohira, B. Adams, A. E. Hassan, and K.-i. Matsumoto, "Predicting re-opened bugs: A case study on the eclipse project," in *2010 17th Working Conference on Reverse Engineering*. IEEE, 2010, pp. 249–258.

[83] T.-M. Hesse, V. Lerche, M. Seiler, K. Knoess, and B. Paech, "Documented decision-making strategies and decision knowledge in open source projects: An empirical study on firefox issue reports," *IST*, vol. 79, pp. 36–51, 2016.

[84] O. Chaparro, J. Lu, F. Zampetti, L. Moreno, M. Di Penta, A. Marcus, G. Bavota, and V. Ng, "Detecting missing information in bug descriptions," in *FSE'17*, 2017.

[85] A. Mahadi, K. Tongay, and N. A. Ernst, "Cross-dataset design discussion mining," in *SANER'20*, 2020, pp. 149–160.

[86] S. Breu, R. Premraj, J. Sillito, and T. Zimmermann, "Information Needs in Bug Reports: Improving Cooperation Between Developers and Users," in *CSCW'10*, 2010, pp. 301–310.

[87] S. Panichella, G. Canfora, and A. Di Sorbo, ""won't we fix this issue?" qualitative characterization and automated identification of wontfix issues on github," *IST*, vol. 139, p. 106665, 2021.

[88] P. J. Guo, T. Zimmermann, N. Nagappan, and B. Murphy, "Characterizing and predicting which bugs get fixed: An empirical study of Microsoft Windows," in *ICSE'10*, 2010, pp. 495–504.

[89] M. M. Rahman, F. Khomh, and M. Castelluccio, "Works for me! cannot reproduce–a large scale empirical study of non-reproducible bugs," *EMSE*, vol. 27, no. 5, p. 111, 2022.

[90] L. Tan, C. Liu, Z. Li, X. Wang, Y. Zhou, and C. Zhai, "Bug characteristics in open source software," *EMSE*, vol. 19, no. 6, pp. 1665–1705, 2014.

[91] N. Limsettho, H. Hata, A. Monden, and K. Matsumoto, "Unsupervised bug report categorization using clustering and labeling algorithm," *JSEKE'16*, vol. 26, no. 07, pp. 1027–1053, 2016.

[92] A. Sureka and P. Jalote, "Detecting Duplicate Bug Report Using Character N-Gram-Based Features," in *ASPEC'10*, 2010, pp. 366–374.

[93] A. Yan, H. Zhong, D. Song, and L. Jia, "How do programmers fix bugs as workarounds? an empirical study on apache projects," *EMSE*, vol. 28, no. 4, p. 96, 2023.

[94] V. Agrawal, Y.-H. Lin, and J. Cheng, "Understanding the characteristics of visual contents in open source issue discussions: a case study of jupyter notebook," in *ESEM'22*, 2022, pp. 249–254.

[95] Y. Huang, D. A. da Costa, F. Zhang, and Y. Zou, "An empirical study on the issue reports with questions raised during the issue resolving process," *EMSE*, vol. 24, pp. 718–750, 2019.

[96] D. Arya, W. Wang, J. L. Guo, and J. Cheng, "Analysis and detection of information types of open source software issue discussions," in *ICSE'19*, 2019, pp. 454–464.

[97] R. Marques, M. M. da Silva, and D. R. Ferreira, "Assessing agile software development processes with process mining: A case study," in *CBI'18*, vol. 1, 2018, pp. 109–118.

[98] T. Krismayer, C. Mayr-Dorn, J. Tuder, R. Rabiser, and P. Grünbacher, "Using constraint mining to analyze software development processes," in *ICSSP'19*, 2019, pp. 94–103.

[99] W. Poncin, A. Serebrenik, and M. Van Den Brand, "Process mining software repositories," in *CSMR'11*, 2011, pp. 5–14.

[100] M. Gupta, A. Sureka, and S. Padmanabhuni, "Process mining multiple repositories for software defect resolution from control and organizational perspective," in *MSR'14*, 2014, pp. 122–131.

[101] M. Mittal and A. Sureka, "Process mining software repositories from student projects in an undergraduate software engineering course," in *Companion proceedings of the 36th international conference on software engineering*, 2014, pp. 344–353.

[102] V. Rubin, C. W. Günther, W. M. Van Der Aalst, E. Kindler, B. F. Van Dongen, and W. Schäfer, "Process mining framework for software processes," in *Software Process Dynamics and Agility: International Conference on Software Process, ICSP 2007, Minneapolis, MN, USA, May 19-20, 2007. Proceedings*. Springer, 2007, pp. 169–181.

[103] M. Gupta and A. Sureka, "Nirikshan: Mining bug report history for discovering process maps, inefficiencies and inconsistencies," in *Proceedings of the 7th India Software Engineering Conference*, 2014, pp. 1–10.

[104] V. Saini, P. Singh, and A. Sureka, "Control-flow based anomaly detection in the bug-fixing process of open-source projects," in *Proceedings of the 13th Innovations in Software Engineering Conference (formerly known as India Software Engineering Conference)*, 2020, pp. 1–11.

[105] B. Dobrzyński and J. Sosnowski, "Tracing life cycle of software bugs," in *Dependability Engineering and Complex Systems: Proceedings of the Eleventh International Conference on Dependability and Complex Systems DepCoS-RELCOMEX. June 27–July 1, 2016, Brunów, Poland*. Springer, 2016, pp. 109–120.

[106] T. Koponen, "Life cycle of defects in open source software projects," in *IFIP International Conference on Open Source Systems*. Springer, 2006, pp. 195–200.

[107] J. Wang and H. Zhang, "Predicting defect numbers based on defect state transition models," in *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement*, 2012, pp. 191–200.

[108] B. Coremans, A. L. Klomp, S. A. Rukmono, J. Krüger, D. Fahland, and M. R. Chaudron, "Process mining from jira issues at a large company," in *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2023, pp. 425–435.

[109] Y. Wang, J. Cao, and D. Lo, "Mining and predicting micro-process patterns of issue resolution for open source software projects," 2020.