Empirical Software Engineering
https://doi.org/10.1007/s10664-018-9674-x

The life-cycle of merge conflicts: processes, barriers, and strategies

Nicholas Nelson[1] · Caius Brindescu[1] · Shane McKee[1] · Anita Sarma[1] · Danny Dig[1]

Published online: 05 February 2019

Abstract
Merge conflicts occur when developers make concurrent changes to the same part of the code. They are an inevitable and disruptive aspect of collaborative software development. Thus tool builders and researchers have focused on the prevention and automatic resolution of merge conflicts. However, there is little empirical knowledge about how developers actually monitor for merge conflicts and plan, perform, and evaluate resolutions. Without such knowledge, tool builders might be building on the wrong assumptions and researchers might miss opportunities for improving the state of the art. We conducted semi-structured interviews with 10 software developers across 7 organizations, including both open-source and commercial projects. We identify key processes, techniques, and perceptions from developers, which we extend and validate via two surveys, a *Barriers Survey* and a *Processes Survey,* of 162 and 102 developers, respectively. Among others, we find that developers rely on reactive strategies of monitoring for merge conflicts. We find that developers defer responding to conflicts based on their perception of the complexity of the conflicting code and that deferring affects the workflow of the entire team. Developers also rely on this perception to visually evaluate their merge conflict resolutions for correctness. Finally, developers' perceptions alter the impact of tools and processes designed to preemptively and efficiently resolve merge conflicts. Understanding their processes and perceptions can help design human-oriented tools that better support their individual development processes.

Keywords Merge conflicts · Resolution processes · Developer perceptions · Barriers · Developer tools · Awareness · Planning · Evaluation

1 Introduction

Collaborative development is essential for the success of large projects (Hattori and Lanza 2010), and is enabled by version control systems. In Git, and other version control systems, developers work on their changes in isolation; periodically synchronizing them by merging

Communicated by: Lu Zhang, Thomas Zimmermann, Xin Peng and Hong Mei

✉ Nicholas Nelson
nelsonni@oregonstate.edu

Extended author information available on the last page of the article.

with the main line of development. This can be problematic, because developers can concurrently change the same code, without being aware of each others' changes. These overlapping changes become evident when they try to merge their work into the main line, and encounter a *merge conflict*. In the majority of cases, the merges succeed. However, research has shown (Kasi and Sarma 2013; Brun et al. 2011) that in open source projects, merge conflicts occur in approximately 19% of all merges.

Resolving merge conflicts is nontrivial, especially when the changes diverge significantly (Brun et al. 2011). The resolution process can be tedious and can cause delays as developers figure out how to approach and resolve conflicts (Kasi and Sarma 2013). Poorly-performed merge conflict resolutions have been known to cause integration errors (Bird and Zimmermann 2012), workflow disruptions, and jeopardize project efficiency and introduce delays (Estler et al. 2014).

Developers are aware of the problems posed by merge conflict resolutions. They follow different informal processes to avoid encountering, or having to resolve conflicts; e.g. sending out emails to the rest of the team (de Souza et al. 2003), performing partial commits, or racing to finish changes (Cataldo and Herbsleb 2008). Unfortunately, these practices come with their own problems, and can make the resolution of a merge conflict even harder (Brun et al. 2011).

Past work examined different mechanisms for proactive merge conflict detection, including Crystal for preemptive merging (Brun et al. 2011), Palantír for awareness of parallel changes (Sarma 2008) and WeCode for continuous merging (Guimarães and Silva 2012). Mens (2002) presented a survey of merge conflict resolution techniques examined up to 2002. Nishimura and Maruyama (2016) used fine-grained edit history to localize potential conflicts. Apel et al. (2011, 2012) presented an approach for merging code, by taking into consideration the syntactical structure of the code. Lippe and van Oosterom (1992) presented Operational Based Merging, and Dig et al. (2008) proposed a refactoring-aware implementation for Java, called MohaldoRef. Finally, Hunt and Tichy (2002) presented an extensible language-aware merging technique that uses both language structure and semantics for improved results.

However, several key questions remain unanswered: How do developers approach and manage merge conflicts? How do developers perceive the difficulty of a merge conflict resolution? Do the current tools support developers' merge conflict resolution needs? Without such knowledge, tool builders might be building on wrong assumptions and researchers might miss opportunities for improving the state of the art.

To answer these questions, we talked directly to developers. This step is crucial to understanding problems in the context in which they occur; which is highlighted by researchers as a pressing concern in software engineering (Fritz and Murphy 2010; Sillito et al. 2006; Ko et al. 2007). We interviewed 10 software developers from 7 organizations about their experiences and perceptions of merge conflicts. Our participants had a median of 5 years of software development experience, and work on a mix of both small-scale (less than 10 contributors) and large-scale projects (greater than 1000 contributors). These interviews helped us understand how developers approach and manage merge conflicts, and their unmet needs within their processes and tools.

To triangulate our findings and provide a broader understanding of developers' processes, techniques, tools, barriers, and perceptions of merge conflicts, we deployed two surveys to a larger population of software developers. The *Barriers Survey* and *Processes Survey* sampled 162 and 102 developers, respectively (264 developers in total). For both surveys, the majority of our participants had 6 or more years of software development experience, and reported facing merge conflicts a few times a week.

To understand how software developers manage merge conflicts, including the tools used and difficulties experienced, we answer the following research questions:

- **RQ1:** How do software developers become **aware** of merge conflicts?
- **RQ2:** How do software developers **plan** for merge conflict resolutions?
- **RQ3:** How do software developers **evaluate** merge conflict resolutions?
- **RQ4:** What difficulties do software developers experience when managing merge conflicts?
- **RQ5:** How well do tools support developer's needs for managing merge conflicts?

We found that developers, when initially assessing a merge conflict, rely on the *code complexity of the conflicting lines* and *their own knowledge in the area of the conflict* as the top two factors when estimating the difficulty of a merge conflict resolution. These concerns cause developers to alter their resolution strategy, and in some cases delay the resolution, which can have negative consequences.

After understanding the merge conflict, developers must resolve the conflict in order to return to normal development. We found that the key challenges that developers face when resolving conflicts are *understanding the conflicting code,* and having enough meta information about the conflicting code (who made the change, why, and when). Developers rely heavily on *their knowledge of the conflicting code* when implementing their merge resolutions.

Our findings show that developers perceive that an *increase in conflict complexity* has a greater impact on the resolution difficulty, than an increase in the size of the conflict. However, development tools lack features that address this dimension. This could partially be alleviated by focusing on the tool improvements most desired by developers: *better usability, better information filtering,* and *better history exploration.*

Overall, we make the following contributions:

1. We introduce a model of developers' processes for managing merge conflicts, from the point of *awareness* to the *resolution* of a conflict;
2. We discuss proactive and reactive strategies developers use when monitoring for merge conflicts;
3. We provide evidence for the prevalence of deferring a merge conflict resolution, and the knock-on effects of doing so;
4. We provide empirically-derived rankings of factors that developers perceive as increasing the difficulty of a merge conflict resolution;
5. We expose disparities between developers' needs when resolving merge conflicts, and the features provided by development toolsets.

This article extends the work presented at ICSME 2017 (McKee et al. 2017) through the addition of the first three contributions (see above), and in particular: (1) by providing a model of developer's processes for managing merge conflicts; (2) conducting an additional *Processes Survey* for validating our model and examining developers' strategies, and; (3) extending the results of the previously conducted *Barriers Survey* by further analysis based on our process model.

2 Related Work

Merge conflicts have been examined through different lenses by researchers. We present a brief summary of related works, grouped by relevant topics to this article.

2.1 Collaboration

Gousios et al. (2015) conduct a study in which they ask integrators to describe difficulties in maintaining their projects and code contributions. They showed that integrators have problems with their tools, have trouble with non-atomic changesets, and rank *git knowledge* in the top 30% of their list of biggest challenges. Gousios et al. (2016) additionally conducted a study into the challenges of the pull-based model from the perspective of contributors. They found that most challenges relate to code contribution, the tools and model used to contribute, and the social aspects of contributing (specifically highlighting merge conflicts). These works focus on the collaborative processes that go into contributing to open-source projects and operating as integrators within them, whereas we examine the processes and issues inherent to merge conflicts and the tools built to support their resolution.

Guzzi et al. (2015) conducted an exploratory investigation and tool evaluation for supporting collaboration in teamwork within the IDE. They found that developers working within a variety of companies were able to quickly and easily resolve merge conflicts, and did this using merge tools. However, they also note that although automatic merging was used, their participants also manually checked each conflict and suggest that this reveals some mistrust of tools. Guzzi et al. further explain that their interviewees avoid merge conflicts by using strict policies and software modularity. Their results complement our findings that toolset mistrust is a major concern, and that standards need to be implemented in order to avoid complex merge conflicts.

Begole et al. (2002) investigate the work rhythms of developers. They use minute-by-minute records of computer activity coupled with locality of the activity, calendar appointments, and e-mail activities to provide meaningful visualizations for group coordination. The passive nature of developers' interaction with these visualizations requires users to engage and coordinate with each other, which differs from version control systems that actively support the software development process.

These works highlight the importance of collaboration and coordination in the daily activities of developers, which provide impetus for our examination of developers adaptations in the presence of merge conflicts, which represents a breakdown in those activities.

2.2 History Understanding and Navigation

Codoban et al. (2015) seek to evaluate developer understanding and usage of code history. Our results show that tool support during history exploration factors a moderate amount into the difficulty of a merge conflict (N10: *Tool Support for History Exploration*, see Section 8.5). We independently verify their findings that developers experience tool limitations in usability (I1: *Usability*, see Section 9.1) and history visualization (I4: *Graphical information presentation*).

Ragavan et al. (2017) propose an Information Foraging Theory (IFT) model for how developers forage in the presences of history (in their paper they refer to this as "variations"). This model highlights the needs of developers attempting to understand variations in code, whereas we examine the methods and strategies that developers employ prior to encountering a merge conflict and the processes for evaluating their resolutions.

While these studies provide an insight into how developers use, explore, and understand history, they do not approach any of the problems that collaboration can bring to software development. We aim to examine the complete process from awareness of a merge conflict to it's eventual resolution.

2.3 Better Merge Conflict Resolution

Currently, all version control systems treat source code files as text. Therefore, merging is done at a textual level, ignoring all structure that the files might contain. Several researchers have looked at ways to improve this status quo.

Westfechtel (1991) propose a merging technique that uses the structural (i.e. lexical) information of a language when performing a merge. However, such tools are language dependent and the required algorithms are expensive to run. Apel et al. propose *JDime* which performs both structured (Apel et al. 2012) and semi-structured merges (Apel et al. 2011) merge techniques. Both approaches improve existing structured merging techniques by only using structural information when the unstructured (i.e. text only) merge has failed. Binkley et al. (1995) propose using call graph information to correctly merge different versions of the program.

Accioly et al. (2018a) used a semi-structured approach to understand the types of merge conflicts. Overall, they identified 9 conflict categories, depending on the syntactical elements that were conflicting. Their work identifies the types and frequencies of merge conflicts, however, it does not address the impact of human factors on the prevalence of merge conflicts.

Lippe and van Oosterom (1992) go a different way. They propose a new merging technique, *operation-based merging* that would replay the changes that were performed on the two branches, in the order in which they were performed. Dig et al. (2008) uses this technique and shows empirically that many more merge conflicts could be solved by a tool that understood the semantics of change operations.

These studies seek to improve the performance and reliability of merging tools, which complement our results which show that toolset mistrust is a major concern among developers. By addressing the quality and consistency of the algorithms and tools available for merge conflicts, tool builders can hopefully improve developer trust in the future.

2.4 Workspace Awareness

Biehl et al. (2007) propose *FastDASH*, a tool that fosters awareness between members of a team. FastDASH provides a dashboard that shows the files that are checked out, modified, and staged by other members of the team. da Silva et al. (2006) propose *Lighthouse* to show the changes being made at the design level. Their tools presents all changes from the perspective of changes to the model (in the form of UML diagrams) of all of the developers project. While all these approaches provide awareness of potential conflicts, they require the developer to actively monitor and discern if a conflict is likely or has occurred.

Sarma (2008) and Sarma et al. (2003) go a step further and propose *Palantír*. Palantír monitors other developer's workspaces, and, depending on the changes, will notify the developer, in a non-obtrusive manner, if a conflict has happened. Similarly, Hattori and Lanza (2010) propose *Syde* that monitors the changes at an Abstract Syntax Tree (AST) level. This allows the tool to give more precise information to the developer.

Brun et al. (2011) propose *Crystal*, which monitors selected branches in the repository. Crystal preemptively merges the branches in the background and will notify the developers of any conflicts that arise. It detects both *direct* conflicts (changes to the same line), and *indirect* conflicts (changes to a different line that cause build or test failures). Guimarães and Silva (2012) propose *WeCode*, which also merges in uncommitted code, in order to improve the time to detection of a merge conflict. Finally, Estler et al. (2013) presents a collaboration framework that integrates both fine grained changes and a real-time awareness system to prevent merge conflicts.

Servant et al. (2010) proposes *CASI*, that uses visualization to help developers detect conflict early. CASI shows all the program elements that are influenced by the changes made in the team, so that developers can coordinate more efficiently.

Kasi and Sarma (2013) take a more proactive approach and propose a novel task scheduling approach that aims to minimize the number of conflicts. *Cassandra* uses developer preferences, task and file dependencies to schedule tasks so that they are less likely to conflict. On a similar approach, Accioly et al. (2018b) present 2 predictors that can be used to identify potential merge conflict ahead of time. Their predictors achieve precision and recall percentages that range between 8.85%–57.99% and 13.15%–83.62%, respectively. In their current form, high numbers of erroneous predictions make these techniques unsuitable for industry adoption.

However, despite the extensive research, very few techniques are used by professional developers. This can be mostly blamed on a lack of awareness. This lack of awareness also means that tools are less mature, as maturity is reached once a tool has a stable user base. Our research aims to better understand the problems merge conflicts pose, and aid developers more directly. We hope that this will serve as a stepping stone to bringing more of the existing tools to the lime light.

2.5 Program Comprehension

Program comprehension is a major research area within Software Engineering. In this subsection, we present work that is related to merge conflicts, their understanding, and their resolution.

Borg et al. (), through interview, look at how tools support *Change Impact Analysis*. They find that developers have different information seeking behaviors, and tools should support these various seeking approaches.

Wang and Lo (2014) propose a technique that uses version history and previous bug reports to assist developers in localizing bugs. Panichella et al. (2014) explore how collaboration affects the structure of the source code under development. Robillard and Manggala (2008) propose a technique for reusing information obtained during previous code exploration tasks, to assist developers in more efficiently locating information they need now.

Tao et al. (2012) investigates how developers understand software changes, in an industrial setting. They find significant gaps between the developer's need and what the tools provide.

Existing work focuses on investigating how developers understand source code, and changes made to source code, when isolated as a single stream of changes. However, when resolving a merge conflict a developer has to understand two sets of changes that conflict. Moreover, they need to understand how the changes interact with the existing code base, and *between each other* in order to successfully resolve the conflict. It is also worth noting that the changes occurred at different points in time. Even if a developers were to consult the author of the changes, they might not have perfect recollection of the exact reasoning of how and why those changes were made. All these factors add to the cognitive load that developers face, which makes conflict resolution a difficult task.

3 Methodology

To understand the merge conflict processes, barriers, and resolution strategies of software developers, we used mixed methods consisting of interviews to gather qualitative insights, and surveys to provide quantitative triangulations into the broader context of merge conflicts. Mixed methods allow us to identify perspectives and themes from both individual and population-wide samples to strengthen the validity of both, as per guidelines from Easterbrook et al. (2008).

We conducted *Exploratory Interviews* with software developers to create a taxonomy of processes, barriers, strategies, and concerns experienced by developers when encountering and resolving merge conflicts. We then triangulate and extend the results of our interviews by conducting a *Barriers Survey* and a *Processes Survey* using concepts and vocabulary generated from the interviews.

To analyze the scale of barriers, constraints, and concerns of software developers when approaching merge conflicts, we conducted a *Barriers Survey* of software developers. In this survey, we sought to extend the results from our interviews and to include additional questions relating to tools, technology, and coordination within development teams.

Processes are developed to address common problems in teams and organizations. Identifying the problems developers face is an essential element for process improvements (Beecham et al. 2003). We conducted an additional *Processes Survey* of software developers to understand how they monitor for merge conflicts, how they plan their resolution strategies, and their processes for evaluating whether their resolutions are successful.

The full set of questions in the interview, as well as the questions and codebooks used for both surveys can be found on our companion site.[1]

3.1 Exploratory Interviews

Semi-structured interviews provide qualitative data collection through open-ended questions that elicit interviewee's thoughts and opinions about a particular topic. The resulting data includes themes and terminology from the perspective of the interviewee, as opposed to the interviewer, and provides a context for further quantitative inquiry (Easterbrook et al. 2008).

We conducted semi-structured interviews with software developers to understand their concerns when facing merge conflicts and the factors that impact merge conflict difficulty. We initially recruited developers from industry contacts or by soliciting contributors of various open-source projects. These initial developers referred additional participants. We accepted any referred participants with software development experience, and referred participants were not asked to refer additional participants. This limitation follows best practices for snowball sampling (Goodman 1961), and allowed us to reduce the impact of any particular cohort of developers on the generalizability of our results.

We interviewed ten software developers from seven different organizations spanning six different industries. Eight of the participants worked primarily on open-source projects. Participants worked on a variety of project sizes; from projects with less than five active contributors, to more than 1700 contributors (as evaluated by analyzing code repositories for a 12-month period). Table 1 provides additional demographics data, including software development experience, role, industry, project size, and whether the participant primarily focused on open- or closed-source software development. Each interview lasted between 30 to 60 minutes. Participants were offered US$50 in either cash, gift card, or a donation to a charity of their choice.

[1]Companion site: http://web.engr.oregonstate.edu/~nelsonni/emse18.html

Table 1 Interview participant demographics

Par.[i]	Exp.[ii]	Role	Industry	Source[iii]	Contrib.[iv]
P1	18 yrs.	Sr. Software Developer	Semiconductor Mfr.	Open	1700
P2	6 yrs.	Software Engineer	Semiconductor Mfr.	Open	1700
P3	3 yrs.	Software Engineer	Semiconductor Mfr.	Open	1700
P4	10 yrs.	Software Developer	Academia	Open	< 10
P5	3 yrs.	Infrastructure Engineer	Healthcare Software	Closed	< 10
P6	5 yrs.	Software Developer	Healthcare Software	Closed	< 10
P7	5 yrs.	Software Engineer	Business Software	Open	200
P8	25 yrs.	Director	Academia	Open	600
P9	8 yrs.	Software Developer	IT Services	Open	600
P10	2 yrs.	Software Developer	Sports Software	Open	< 5

[i]Par. = Interview participant

[ii]Exp. = Years of software development experience

[iii]Source = Source code licensing in primary project

[iv]Contrib. = Approximate number of individual contributors in primary project (between March 2016-March 2017)

At the beginning of the interview we gave participants a short explanation of the research goals, our definition of merge conflicts, and collected demographics data. We then asked participants about the roles that they play in their project, their experience working in team settings, questions about merge conflicts, the process of conflict resolution, and the difficulties that they faced in conflict resolution.

We formulated the interview questions about merge conflicts in order to understand how developers perceived and how they approached merge conflicts. The following is an example of some of the questions we asked in the interview; the full set is available on our companion site.

– Can you describe a merge conflict, or a set of conflicts, that you would consider to be the typical case?
– Do you have any particularly memorable merge conflict resolutions that you can recall?
– Have you had some code structures, design patterns, coding styles, etc., that you would consider a "usual suspect" in a conflict?
– What kind of measures would you take to minimize the amount of defects that you introduce?

The semi-structured interview format allowed participants to provide us with unanticipated information (Seaman 2008). Further, we allowed open-ended discussion about merge conflicts in general at the end of the interview, which allowed participants to share ideas and topics that they found particularly important. We continued interviewing participants until we reached saturation in the answers, which was measured using topic saturation as our benchmark (Fusch and Ness 2015).

3.2 Barriers Survey

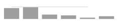
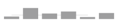
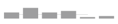
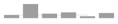
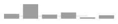
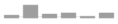
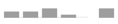
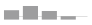
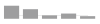
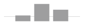
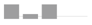
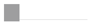
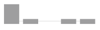
We conducted a 50-question *Barriers Survey* of software developers in order to examine the barriers, constraints, and concerns experienced when encountering merge conflicts. We developed questions to confirm, extend, and broaden the results from the interviews.

We recruited participants from contributor lists on popular open-source repositories on GitHub, advertised on social networking sites (Facebook, Twitter and Reddit), and by directly contacting software developers via email. Participants spanned a variety of organization structures and geographical locations, giving generalizability to results. The survey was conducted online and anonymity was guaranteed in order to elicit honest responses from participants. The *Barriers Survey* was available for 56 days and we received 162 survey responses, but individual parts of the survey have varying response rates and are reported where appropriate in Section 4.

Survey participants were given six different software roles to select, and in many cases, participants considered themselves to be fulfilling multiple roles (59.2% selected two or more). A majority of participants considered themselves to be *Software Developer* (95.1% overall). Participants indicated a median software developer experience of 6–10 years (36.4% overall), and worked on project sizing ranges from 2 to more than 51 developers (the median was 2–5 developers, constituting 48.8% of all responses). Table 2 provides additional demographics data delineated by role, including median and distribution of software development experience responses, and median team size.

We divided the *Barriers Survey* into four categories, each category containing 5-7 questions (see our companion site for a list of questions). First, we elicited background

Table 2 Survey participant demographics from barriers survey and processes survey

Barriers survey					
Role	Participants[i]		Soft. Dev. Experience[ii]		Team Size[iii]
Software Developer	154	(95.06%)		6–10 years	2–5 developers
System Architect	54	(33.33%)		11–15 years	6–10 developers
DevOps	53	(32.72%)		11–15 years	6–10 developers
Project Manager	44	(27.16%)		16–20 years	2–5 developers
Project Maintainer	40	(24.69%)		6–10 years	2–5 developers
System Administrator	23	(14.20%)		11–15 years	2–5 developers
Other	11	(6.79%)		11–15 years	6–10 developers
Processes Survey					
Roles	Participants[iv]		Soft. Dev. Experience[ii]		Team Size[iii]
Developer	41	(40.59%)		6–10 years	2–5 developers
Engineer	39	(38.61%)		6–10 years	6–10 developers
Architect	6	(5.94%)		11–15 years	6–10 developers
DevOps	7	(6.93%)		6–10 years	6–10 developers
Designer	1	(0.99%)		1–5 years	1 developer
Other	7	(6.93%)		1–5 years	2–5 developers

[i]Number and percentage of *Barriers Survey* participants that selected each role; multiple selections were allowed (162 total responses)

[ii]Percentile distribution of software development experience (on left) in ranges: 1–5 years, 6–10 years, 11–15 years, 16–20 years, 21–25 years, and 26+ years. Median selection (on right)

[iii]Median selection of primary team size

[iv]Number and percentage of *Processes Survey* participants that selected each role; only one selection per participant was allowed (102 total responses)

information about demographics, roles, and experience. Second, we asked questions related to difficulties that developers experience when encountering merge conflicts. Third, we asked questions related to conflict resolution and the factors that affect developers. Finally we asked questions about the tools and tool features that developers use when working with merge conflicts. Questions were presented either as 5-point Likert-type scales (with no pre-selected answers) or open-ended text forms to gather additional insights.

3.3 Processes Survey

Merge conflicts disrupt the collaborative development workflow, and developers have adapted and developed different processes for handling these complexities. To understand the common structure and prevalence of these processes we use surveys, which are used for mapping the state of practice, establishing baselines for investigating research topics, and gathering opinions regarding software engineering technologies and practices (de Mello and Travassos 2016). Therefore, we conducted a second 15-question *Processes Survey* of software developers that included both open-ended and predefined questions.

We recruited participants for the *Processes Survey* to be similar to the *Barriers Survey* participants so that we could compare and triangulate the results. We therefore recruited participants from contributor lists on popular open-source repositories on GitHub, advertised on social networking sites (Twitter and Reddit), and by directly contacting software developers via email. Due to the nature of social media and mailing lists, we cannot compute a response rate from these distribution methods. We observe that 35.29% of participants were located outside of the United States and several participants indicated that they sent the survey onward to other software developers.

The survey was conducted online and anonymity was guaranteed. The *Processes Survey* was available for 38 days and we received 113 survey responses, however, 11 responses were incomplete, resulting in 102 total responses. The results of this survey are presented in Section 4.

Survey participants had a mean of 9.1 years of software development experience, and primarily worked in teams of 2–5 members (45.1% overall). Based upon input we received during our *Barriers Survey*, we modified several names to more accurately reflect the terminology used within the software development industry. Participants were given seven different roles to select, as well as an *Other* field to provide additional roles not included in the pre-populated options. A majority of participants considered themselves to be a *Developer* (40.6% overall).

Participants indicated a median software developer experience of 6–10 years (29.7% overall), and worked on project sizing ranges from 1 to more than 51 developers (the median was 2–5 developers, constituting 44.6% of all responses). Table 2 provides additional demographics data delineated by role, including median and distribution of software development experience responses, and median team size.

We divided the survey into four categories, with each containing 3-5 questions. First, we elicited background information about demographics, roles, and experience. Second, we asked questions relating to how and when developers become aware of merge conflicts. Third, we asked questions related to planning and implementing merge conflict resolution strategies. Finally, we asked questions about evaluating the effectiveness of those merge conflict resolutions and the particular tools that are used throughout the processes of working with merge conflicts.

3.4 Data Analysis

The methods used to analyze and evaluate the results from the *Exploratory Interviews, Barriers Survey* and *Processes Survey* are described below. We present the results of this analysis in Sections 5–9.

3.4.1 Interview Analysis

Interviews were audio-taped and transcribed. The first and third authors unitized (Campbell et al. 2013) the interview transcripts into cards that each contained a single logically consistent statement. To organize these cards we employed card sorting, a collaborative technique of exploring how people think about a certain topic (Spencer 2009; Hudson 2013), which allows key concepts and associations to be identified through an open sorting method that iteratively develops categories during the process.

We performed two iterations of the open card sorting process. In the first iteration, we developed a standardized coding scheme and improved it to an acceptable point through *negotiated agreement,* which was reached when no further thematic categories could be created and agreed upon by both coders (Garrison et al. 2006; Ritchie et al. 2013). The coding scheme dictated that sentences must be consecutive and topically related to be grouped into a single card. Logically connected statements that were separated by other lines were considered to be separate cards, as a conservative measure to preserve context within each card.

In the second iteration, the first and third authors sorted cards according to our coding scheme and discussed the resulting taxonomies until consensus was reached. Based upon our research questions, we grouped the resulting categories as follows: the processes that developers use for merge conflicts (Section 5–7), the difficulties that developers face with merge conflicts (Section 8), and the impact of development tools on the resolution process (Section 9).

3.4.2 Survey Analysis

We evaluated the results of the *Barriers Survey* by performing open card sorting on the all open-ended questions. The resulting categories were standardized to an acceptable point through *negotiated agreement* (Ritchie et al. 2013).

The *Barriers Survey* was primarily composed of Likert-type questions, which were used to measure the extent to which participants agreed with a particular statement. This means that lower mean and median values indicate less agreement with the statement in a particular question. We use this design to validate both the degree of agreement to the interview results, as well as the existence of individual factors.

For the *Processes Survey*, we evaluated the distribution of survey answers for each of the four Likert-type question by analyzing across demographic categories. We used Likert-type questions to measure the extent to which participants agreed with a particular statement, or the degree to which a factor has impacted the participant. Where answers differed across a demographic category, we note the difference and provide further discussion of these results.

The *Processes Survey* contained five open-ended questions. We performed open thematic coding (Fereday and Muir-Cochrane 2006) to analyze the responses to these questions. The resulting codebook, including descriptions and examples, are available on our companion site.

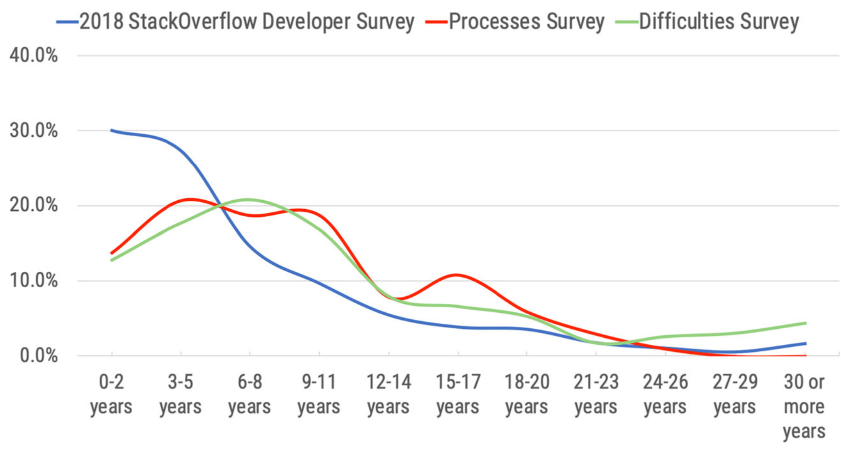


Fig. 1 Percentile distributions of professional programming experience compared between *Processes Survey*, *Barriers Survey*, and 2018 StackOverflow Developer Survey participants. Responses are inclusively binned into 3-year buckets for comparison across surveys

After establishing a codebook, the first two authors independently coded the responses to each open-ended question. For question 7 (*"How do you monitor for merge conflicts?"*), we achieve an inter-rater reliability (IRR) agreement of 0.95 (Jaccard similarity coefficient). For questions 8 (*"How do you determine the urgency of a merge conflict?"*), 11 (*"What is your first step in trying to understand code involved in a merge conflict?"*), 14 (*"What effect did deferring your response to a merge conflict have on the resolution of the conflict?"*), and 19 (*"If your first attempt at resolving a merge conflict fails, what backup strategies do you use?"*), we achieve high IRR agreements of 0.81, 0.88, 0.74, and 0.92, respectively.

To ensure that our survey samples are representative of the larger developer population, we compare the demographics from the *Processes Survey* and the *Barriers Survey* with the results of the 2018 StackOverflow Developer Survey.[2]

The 2018 StackOverflow Developer Survey was conducted on 101,592 software developers from 183 countries. This survey includes the number of years spent coding professionally by 77,903 participants.

Figure 1 provides a graphical representation of the percentile distribution of professional programming experience among participants in the *Processes Survey*, *Barriers Survey*, and the 2018 StackOverflow Developer Survey. We see that our sample population has more experienced developers, and the trends match between all three samples.

To further compare the distribution of programming experience across these population samples, we conduct nonparametric tests of the equality of the probability distributions between two samples. Comparing *Processes Survey* responses with 2018 StackOverflow Developer Survey responses, we conclude that we cannot reject the null hypothesis that these samples were drawn from the same population distribution (Two-sample Kolmogorov-Smirnov test, $D = 0.33636$, $p = 0.5939$; where D represents the combined statistic for both directional hypotheses and varies from 0 to 1). We similarly conclude that *Barriers Survey* responses and 2018 StackOverflow Developer Survey responses could plausibly be drawn from the same population distribution (Two-sample Kolmogorov-Smirnov test, $D =$

[2]https://insights.stackoverflow.com/survey/2018

0.27273, $p = 0.8326$). In both cases, we cannot reject the null hypothesis that the survey sample population is the same as the 2018 StackOverflow Developer Survey population, therefore, we conclude that our samples are representative of the development community.

4 Results

To understand how developers manage merge conflicts, we asked interview participants to describe their current processes for handling merge conflicts.

Participants talked about different steps that they follow, including using tools that alert them to potential or current merge conflicts, processes for analyzing and understanding conflicting code prior to implementing a resolution, and the use of tools for validating that their resolution worked. As an example, P3 said:

> *"Part of my job on the integration team requires that I check for bad regressions. I use scripts to track patches as they're being backported, so I know when and where to look if [a patch] introduces a conflict. [. . .] And once I've fixed [the conflict], I try to compare with the previous version to make sure [the code] works in a similar way."*

Our interview and survey results suggest that developers follow a series of phases through which they manage the life-cycle of individual merge conflicts. We construct a model of the developer processes for managing merge conflicts and examine each phase in detail. Figure 2 provides an illustration of this model. It consists of four phases: *awareness, planning, resolution,* and *evaluation.*

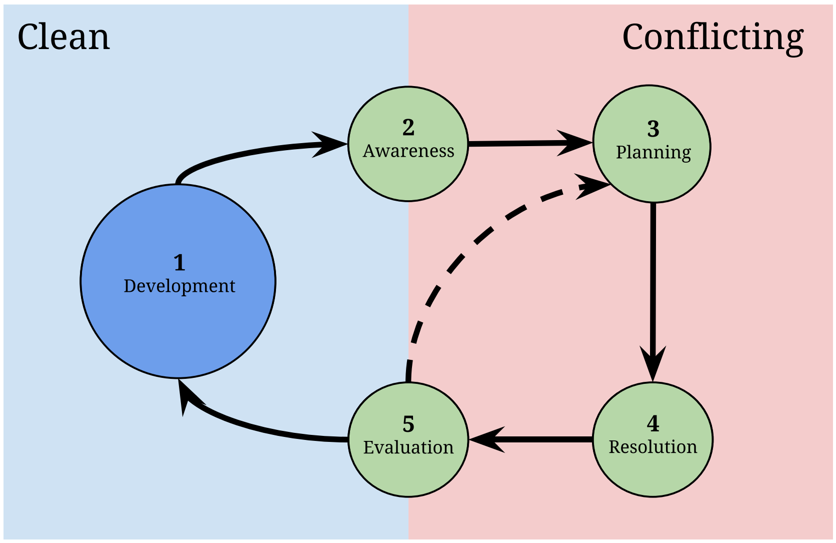


Fig. 2 Model of Developer Processes for Managing Merge Conflicts. Developers alternate between *clean* and *conflicting* states of code. Beginning from (1) *development*, developers maintain (2) *awareness* of conflicts within the codebase in different ways. Once aware, developers begin (3) *planning* for a (4) *resolution* to fix the conflict. And finally, developers (5) *evaluate* the effectiveness of their deployed resolutions (returning to *planning* if the resolution failed)

First, the *awareness* phase consists of the actions developers take to become aware of merge conflicts. This could be passive, as the developer will become aware of a merge conflict when attempting to merge changes or perform a pull. At the other end of the spectrum are developers who *proactively* monitor for merge conflicts as they write code. They are actively looking for changes that might be problematic, either manually or through the use of specialized tools.

Second, the *planning* phase occurs after the developer has become aware that a conflict has occurred, and they are about to tackle the conflict. This includes the decision of when they will try and resolve the conflict. Some developers might try and resolve it immediately, while others might postpone the resolution. Some might change their strategy depending on the conflict, incoming deadlines, or availability of resources. This also includes other actions, such as if they are going to tackle the conflict alone, or collaborate with other developers knowledgeable in the area of conflict (Costa et al. 2016).

Third, the *resolution* phase represents the implementation of the planned resolution. Several tools exist that help in this phase (Nishimura and Maruyama 2016; Mens 2002; Brun et al. 2011). Here we focus on the difficulties that developers face during these resolution implementations (see Section 6).

Finally, after the conflict has been resolved, developers enter in the *evaluation* phase. In this phase, the developer has to evaluate their resolution before considering the conflict as resolved. This is to ensure the correctness of the resulting code. Possible actions during this stage includes compiling the source code. Developers wanting more guarantees can go a step further and run the tests. Finally, some groups have policies such as code reviews that need to be performed on the merge conflict resolution.

In order to explore and validate this model, and our assumptions, we conducted the *Processes Survey*. Our aim in this survey was to understand how developers become aware of merge conflicts (what steps they take, what tools they use, etc.). Also, we wanted to investigate their strategies for dealing with merge conflicts and how they decide whether the resolution has addressed all of their concerns.

Results from our study are categorized according to the life-cycle of merge conflicts; with specific results for the *awareness* (Section 5), *planning* (Section 6), and *evaluation* phases (Section 7). We then present the difficulties that developers experience when managing merge conflicts (Section 8). And finally, we examine the gaps in tool support for managing merge conflicts according to developer's needs (Section 9).

5 How do Software Developers Become Aware of Merge Conflicts? (RQ1)

From the *Processes Survey* we found that 29.41% of participants do not actively monitor for merge conflicts during their development activities. For the rest of the developers who answered with *yes* or *sometimes* (61.77%), we identified 61 different tools mentioned in 126 instances.

5.1 Reactive and Proactive Monitoring for Merge Conflicts

Monitoring for potential merge conflicts can occur at different points in time; before or after a conflict is introduced into a version control system. Reactive monitoring for merge conflicts involves notifying the developer that a conflict has already occurred. Although delays in incorporating changes can increase the costs of resolving any subsequent conflicts (de

Souza et al. 2003; Grinter 1995), developers still use reactive processes to manage conflicts. For the developers who answered that they monitor for merge conflicts (replied either *yes* or *sometimes*), we found that 73.68% (42 out of 57 responses; 6 participants left this field blank) described reactive processes. For example, one *Processes Survey* participant said they use PagerDuty (an IT incident management and notification system) to detect merge conflicts on important development branches:

> *"[...] using PagerDuty, we are all notified if a conflict is pushed to the next or future release branches, so that we can respond quickly. We don't want broken code accidentally going out on a release."*

And another participant mentioned that they try to solve merge conflicts early in order to minimize disruptions to the team:

> *"We try to catch conflicts early so that fewer developers have to be involved in looking at broken code."*

Proactive monitoring allows developers to preemptively catch merge conflicts before they happen. 15 participants (14.71%) mentioned they achieved this by manually tracking incoming changes, such as one participant who indicated:

> *"I monitor commit logs before I begin merging branches so that I see any potentially overlapping code that will break the merge."*

Other teams rely more on communication. This can happen during regular team meetings, to make sure that everybody is aware of each other's tasks, for example another participant said:

> *"[...] standups allow us to know where everyone is working that week."*

While 10 participants (9.80%) indicated that they broadcast their changes in order to notify team members if they will make breaking changes. One participant indicated that team members:

> *"[...] send emails before making breaking changes to the API or related sub-modules."*

When examining the data (see Table 3), we observe that developers across all experience groups monitor for merge conflicts at similar rates (67.7% overall) and that rates of proactive processes also remain consistent (26.3% overall). We find no statistically significant

Table 3 Rates of monitoring for merge conflicts from *processes survey*

Exp.	No[ii]		Yes/Sometimes[ii]		Proactive[iii]		Reactive[iii]	
1–5 years	12	(36.4%)	21	(63.6%)	6	(27.3%)	16	(72.7%)
6–10 years	8	(29.6%)	19	(70.4%)	5	(31.3%)	11	(68.7%)
11+ years	10	(30.3%)	23	(69.7%)	4	(21.1%)	15	(78.9%)
Totals	30	(32.3%)	63	(67.7%)	15	(26.3%)	42	(73.7%)

[i]Ranges of software development experience

[ii]Participant responses to the survey question *"do you monitor for merge conflicts?"* (percentage of experience range responses)

[iii]Participants that monitor for merge conflicts with the proactive or reactive strategy (percentage of 57 proactive/reactive responses)

correlations between participant experience groups when examining across the combinations of monitoring and either proactive or reactive strategies, and conclude that experience is not a driving factor in the adoption of either proactive or reactive strategies for merge conflict monitoring.

To conclude, only a third of developers actively monitor for merge conflicts. When developers are caught unaware of the conflict, additional developers and resources are necessary to fix the conflicting code. This can lead to more frustration, as they do not have any warning of when the conflict will occur and whether they have the time to deal with it immediately.

5.2 Tools for Monitoring for Merge Conflicts

Examining the tools used by participants with reactive processes, we find that 87.72% of these participants rely on version control systems (e.g. Git, SVN, TFS, CVS), and 21.05% use continuous integration systems (e.g. Jenkins, Travis CI, TeamCity). Table 4 presents the top 10 tools developers use when monitoring for merge conflicts, including the totals for both reactive and proactive strategies.

Additionally, we examine the tools used by participants with proactive processes. We find that all participants with a proactive strategy rely on version control systems, and 33.33% use continuous integration systems. Additionally, 26.66% of proactive participants use code analysis tools (e.g. SonarQube, Code Climate).

We find that the majority of tools used by developers for merge conflict monitoring are built to only support reactive strategies, and that multiple tools must be used in conjunction for a proactive approach.

To summarize, we find that developers employ *reactive* processes, even if they are proactive in monitoring for merge conflicts once they have occurred. This can be seen as a consequence of the tools that developers have at their disposal. All the tools mentioned

Table 4 Merge awareness toolsets (top 10) from *processes survey*

Tool[i]	Description	Proactive[ii]		Reactive[ii]		Total[iii]	
Git	Version Control System	10	(9.8%)	30	(29.4%)	40	(39.2%)
GitHub	Project Hosting Site	2	(2.0%)	5	(4.9%)	7	(6.9%)
Email (generic)	Email Client/System	2	(2.0%)	4	(3.9%)	6	(5.9%)
SVN[iv]	Version Control System	0	(0.0%)	4	(3.9%)	4	(3.9%)
VCS (generic)	Version Control System	2	(2.0%)	2	(2.0%)	4	(3.9%)
Visual Studio	IDE	1	(1.0%)	2	(2.0%)	3	(2.9%)
PagerDuty	IT Incident Mgmt.	0	(0.0%)	3	(2.9%)	3	(2.9%)
GitLab	Project Hosting Site	2	(2.0%)	1	(1.0%)	3	(2.9%)
Jenkins	Continuous Integration	0	(0.0%)	3	(2.9%)	3	(2.9%)
TFS[v]	Version Control System	1	(1.0%)	1	(1.0%)	2	(2.0%)

[i] *Processes Survey* participants were allowed to provide multiple tools. 57 out of 102 participants (56%) indicated the use of at least one merge awareness tool

[ii] Participants using this tool with the proactive or reactive strategy (percentages)

[iii] Total number of survey participants using each particular tool

[iv] Subversion

[v] Team Foundation Server

focus support exclusively towards a *reactive* approach, which biases developers towards one particular solution. If developers want a more *proactive* approach, then based on the tools they use, they need to come up with their own solution. The most often cited techniques involve increasing communication among developers. While this technique might be effective in small teams, it scales very poorly and cannot be effectively used in larger organizations (Brooks 1974).

Finally, our results point to the conclusion that developers are not implementing proactive concepts shown in research prototypes (e.g. Palantír Sarma et al. 2003 and Crystal Sarma et al. 2011), and are therefore not leveraging those tools to actively monitor for merge conflicts. These research prototypes are unlikely to have the maturity required for adoption in professional environments, but we find that either tool builders are unaware of the need for such tools or developers have not been educated about proactive monitoring. However, developers are trying to mitigate the severity of merge conflicts by attempting to resolve them as soon as they become aware.

6 How Do Software Developers Plan for Merge Conflict Resolutions? (RQ2)

When encountering a merge conflict, developers follow different strategies. They can either: (a) defer the merge conflict to a later date, or; (b) solve the conflict. In the *Processes Survey* we sought an understanding of these strategies and when developers use them. The tools that developers use when implementing merge conflict resolutions are discussed in Section 9.

6.1 Deferring Responses to Merge Conflicts

One quarter of our participants consider all merge conflicts to be equally urgent. The rest indicated that further information related to the complexity of the conflicting code, the location of the conflict, or the current stage of a software release dictates the urgency assigned to a conflict. Therefore, we can assume that most developers will interrupt their work regardless of the type of merge conflict. They will give the same level of attention, for example, to a conflict generated by whitespace or formatting changes, as a conflict that is generated by overlapping logical changes.

The easiest option when encountering a merge conflict is to simply not deal with it. Indeed, we found that 56.18% of participants have deferred at least once when responding to a merge conflict. The reasons for deferring are varied and listed in Table 5.

The location and complexity of conflicting code (D1, D2) were the most selected factors, and match the top difficulty factors of merge conflicts (F1, F2) as described in Section 8.1.

As the third most selected factor, *ownership of the conflicting code* (D3) indicates that the deferral is not always temporal, but can also be logistical when developers defer to other team members. A *Barriers Survey* participant succinctly defines the role ownership impacts his workflow as:

"Code is mine? I fix it. Code is others? I submit PR or bug reports."

We additionally asked participants to rate the degree to which code ownership factors into their overall merge conflict strategy, and participants indicate that code ownership factors *about half the time* in their strategy of code ownership (mean: 3.21 on a 5-point Likert-type

Table 5 Factors in deferring responses to merge conflicts from *Processes Survey*

Factor	Description	Selections[i]	Percentage[i]
D1	Complexity of the conflicting code	36	(25.00%)
D2	Number of conflicting code locations	32	(22.22%)
D3	Ownership of the conflicting code	25	(17.36%)
D4	Size of the conflicting code	20	(13.89%)
D5	Approaching deadlines	13	(9.03%)
D6	Work schedule constraints	2	(1.39%)
D7	Other	7	(4.86%)

[i]*Processes Survey* participants were allowed to select multiple factors. 44 out of 102 participants (43%) selected more than one factor

scale). Only 10.11% of participants indicated that code ownership *never* factors into their resolution strategy.

While developers have listed multiple reasons for deferral, two stand out: complexity and the number of conflicting locations. Both of these reasons indicate that a developer is more likely to defer if the conflict resolution appears to be lengthy, either because the potential changes are non-trivial or because there are many smaller conflicts requiring the developers' attention.

Deferring the merge conflict resolution comes with a price. Table 6 shows the top effects of deferring a response to a merge conflict. The most common effect was that developers have had to stop the development (*Stop the Presses,* 15 responses) in order to resolve the conflicts. This halt in development includes asking team members to also refrain from adding any additional code into the codebase. The second most common effect is the *increased complexity* of the conflicts (E2), reported by nine participants. A *Barriers Survey* participant noted that:

> *"Deferring a merge conflict simply kicks the can down the road (or off a cliff). Typically resolving the conflict only gets more difficult as time passes."*

Another participant even hinted that the increased complexity can be quite severe, on an order of magnitude greater than if the conflict were addressed immediately:

> *"Untangling takes days instead of minutes when it gets too out of hand."*

Table 6 Effects of deferring response to a merge conflict from *Processes Survey*

Effect	Description	Participants[i]	Percentage[i]
E1	Stop the Presses	15	(32.61%)
E2	Increased complexity	9	(19.57%)
E3	Non-operation effects	5	(10.87%)
E4	Policy/cultural changes	3	(6.52%)
E5	The Nuclear Option	2	(4.35%)
E6	Physical manifestations	1	(2.17%)
E7	Impact beyond the organization	2	(2.17%)

[i]46 out of 102 participants (45.1%) provided a description of the effects of deferring

In some cases, features had to be removed from releases, in order for integration problems to be mitigated and the conflict to be successfully resolved. One participant said:

"We have had several releases come up short in new features because they got delayed by integration problems."

In order to prevent similar problems arising, some organizations have instituted *policy changes* (E4) to prevent this from happening in the future. However, the survey participants that selected *policy changes* (E4) had a mean of 10.2 years of software development experience, which is higher than the overall mean of 9.1 years. The awareness of policy changes that were introduced specifically to address merge conflicts might be higher among senior developers. A *Barriers Survey* participant said:

"We've had devs push a bunch of code up before going on holiday and mucking up a release, so we've instituted an all hands on deck policy for the 2 weeks leading up to a major release"

In one extreme case, a participant reported that an unresolved merge conflict affected production software (E7), which resulted in downtime of the product, as it broke functionality:

"Broke the app for customers until we could get a patch pushed [. . .]."

Finally, the merge conflicts can get too severe and intractable for developers to cope with the complexities. In these types of situations, developers have to resort to the *Nuclear Option* (E5), where they scrap their changes and manually reimplement them. Such as in the case of one participant, who said:

"Uh.... KABOOM! More changes came in and everything piled up. Nothing to do but wipe it all back to clean and start trying to piece things back together."

The results of deferring can be disastrous.

However, it is difficult to assess a deferral to determine if it will turn a single merge conflict into a larger problem. Prototypes such as continuous merging (Guimarães and Silva 2012) and workspace awareness for indirect conflicts (Sarma et al. 2007) might aid developers in assessing the severity of a merge conflict, which in turn would allow them to make informed decisions for deferring a merge conflict resolution.

6.2 Resolution Attempts & Strategies

When developers don't defer their response, they have to resolve the conflicts now. They primarily approach merge conflicts by *examining the merge* (U1), *analyzing or manipulating the code* (U2), or *examining the code* (U3); where the main difference between examining the merge and examining the code is the number of changes that must be understood. Examining the state of code involves understanding one set of changes and code that surrounds them, whereas examining a merge conflict requires understanding two sets of change *and their interactions.* As discussed in Section 2.5, this imposes a considerably higher cognitive load on the developer. Table 7 lists all six strategies described by the *Processes Survey* participants.

A *Processes Survey* participant described their strategy of *examining the merge* (U1) as:

"Reviewing the most recent commits (comments and code) to see whether it's a shallow conflict or not."

Table 7 Initial strategies for understanding conflicting code from *Processes Survey*

Strategy	Description	Participants[i]	Percentage[i]
U1	Examining the merge	26	(32.91%)
U2	Analysis/manipulation of the code	19	(24.05%)
U3	Examining the code	18	(22.79%)
U4	Focus on design concerns	8	(10.13%)
U5	Examine project organization	6	(7.60%)
U6	No strategy	2	(2.53%)

[i]79 out of 102 participants (77%) provided a description of their initial strategy

And another participant indicated their strategy of analyzing the code (U2) involves:

> *"[...] determining if the merge conflict involves important functionality; stepping through with a debugger helps."*

Overall, we find that developers initially focus on the code involved in the merge conflict or information related to the merge itself.

Surprisingly, we found that two of our participants (2.53% of participants) indicated that they *"don't have a strategy"* or *"mostly try to fix it as soon as possible."*

To conclude, developers reported that expertise in the area of the conflicting code is one of the top factors in determining the difficulty of a merge conflict. Additionally, developers also indicate that increases in perceived complexity of merge conflicts is strongly linked with the degree of difficulty in resolving them. Therefore, developers' perceptions and intuition are relied on throughout the implementation of their resolution.

7 How Do Software Developers Evaluate Merge Conflict Resolutions? (RQ3)

After implementing a merge conflict resolution, software developers must evaluate whether their resolution has returned the codebase to a clean state. We asked developers to select the conditions (from interviews) that they use to determine whether their resolution has successfully addressed the merge conflict.

7.1 Success Conditions for Merge Conflict Resolutions

In the *Exploratory Interviews*, developers described six common conditions they considered important in their evaluation. We asked *Processes Survey* participants to select from this list of conditions, including an *Other* option to elicit additional conditions. Only two developers selected that condition, indicating "performance tests showing similar performance" and "client approval." We received 324 selections from 89 participants and present the aggregated results in Table 8.

All tests pass (C1), *code successfully compiles* (C2), and *code looks correct (i.e. visual test passes)* (C3) were the most commonly selected conditions required for a successful merge resolution. These results are in line with existing literature showing that testing (C1) can be used for validating program functionality and correctness (Beizer 1984; Tian 2005). Similarly, the use of compilers to validate code (C2) as being executable and in

Table 8 Conditions of successful merge conflict resolutions from *Processes Survey*

Condition	Description	Selections[i]	Percentage[i]
C1	All tests pass	67	(75.28%)
C2	Code successfully compiles	67	(75.28%)
C3	Code looks correct (i.e. visual test passes)	66	(74.16%)
C4	VCS warnings are gone	51	(57.30%)
C5	Merged code is approved during code review	38	(42.70%)
C6	Merged code accepted into production codebase	33	(37.08%)
C7	Other	2	(2.25%)

[i] *Processes Survey* participants were allowed to select multiple conditions. 79 out of 89 participants (89%) selected multiple conditions

good-working order will be familiar to any developer using a compiled programming language.

The use of visual inspection as a measure of successful merge conflict resolutions is surprising to us, given that *complexity of conflicting lines of code* (F1) is the highest rated factor for impact on merge conflict difficulty (McKee et al. 2017). Inspecting code requires time and expertise in the area of conflicting code. However, the survey participants that selected *code looks correct (i.e. visual test passes)* (C3) had a mean of 9.2 years of programming experience, which is only slightly higher than the overall mean of 9.1 years of programming experience.

Looking at the combination of *code looks correct (i.e. visual test passes)* (C3) with the other conditions, we find that 54 participants also selected *all tests pass* (C1) (52.9%). As the most common co-occuring selections, we conclude that although developers rely upon their expertise to visually inspect a merge conflict resolution, they also run the test suite to validate their evaluation. Experience can play a big factor, as this visual method (C3) is highly subjective.

The two most common evaluation criteria that developers mentioned are that the *code compiles,* and that *all tests pass.* However, less then half selected both options. While tests passing can be considered a good criteria of a successful resolution, the fact that the code compiles is not. Even if the code compiles, there can be logical errors that are introduced during the merge resolution process, especially if the resolution was difficult.

Interestingly, only a minority of developers (42.70%) mentioned code reviews as part of their success criteria. A reason for this might be that developers consider code reviews to be preventative measures, since code reviews can be conducted both pre– and post–merging changes into the codebase.

7.2 Merge Resolution Evaluation Toolsets

From the *Exploratory Interviews*, we identified five categories of software development tools that developers mention in relation to merge conflicts. In the *Processes Survey*, we asked the developers to identify the tools they use when evaluating a merge conflict resolution. We received 204 selections from 89 participants. The aggregated results are presented in Table 9, ranked according to the percentage of participants that selected each toolset.

By far, the most selected tools were *version control systems* (VCS) and *continuous integration* (CI) platforms, with 82 (92.14%) and 62 (69.66%), respectively. The mean for

Table 9 Merge resolution evaluation toolsets from *Processes Survey*

Description	Selections[i]	Percentage[i]
Version control Systems (e.g. Git, Subversion, CVS)	82	(92.14%)
Continuous integration (e.g. TravisCI, Jenkins, TFS)	62	(69.66%)
Program analysis Tools (e.g. Coverity, CodeSonar)	26	(29.21%)
DevOps tools (e.g. Nagios, Monit, Kabana)	17	(19.10%)
Release management tools (e.g. Chef, Puppet, Salt)	9	(10.11%)
Other tools	8	(8.99%)

[i]*Processes Survey* participants were allowed to select multiple toolsets. 64 out of 89 participants (71.91%) selected multiple toolsets

all other tool categories was 15 selections (16.85%), and represents a combined 29.4% of response selections.

The use of version control systems to determine whether a resolution was successful aligns with the *VCS warnings are gone* (C3) condition. Also, continuous integration is dependent on code being compilable (C2), and tests being written and maintained (C1). However, the availability of tools for evaluating merge conflict resolutions might constrain the conditions that developers are willing to consider for their merge conflict resolutions to be successful. Further research is needed to determine whether there is a causal relationship between these dimensions, and whether more effective conditions could be supported by merge conflict toolsets.

Not all of the tools developers use for evaluating the result of a merge conflict resolution can detect all types of merge conflicts. For example, Version Control Systems will detect only direct conflicts. Even if the conflict is solved, from the version control systems' perspective, there still might be build or test issues. Indirect conflicts might slip through if the developer does not run the test suite after resolving the conflict. While almost 70% of our participants mentioned that they used Continuous Integration as part of the evaluation process, those that don't might be inadvertently introducing bugs when they resolve the merge conflict.

Finally, developers have to manually check if their merge resolution is correct. This is done, either by checking that the version control warnings are gone, inspecting the code for any mistakes, or by manually running the tests. We notice that there is a lack of an automated process. Without it the developer might, willingly or unwillingly, skip steps. Also, this lack of a comprehensive toolset might hamper new developers in their efforts to successfully resolve merge conflicts.

7.3 Backup Strategies

Merge conflict resolutions are not always successful. When they fail, developers must alter their patch and potentially switch strategies in order to successfully resolve the conflict.

To understand the prevalence of failed conflict resolutions, we asked *Processes Survey* participants to indicate the frequency in which their first attempt at resolving a merge conflict fails (see Table 10). The most common response was *somewhat infrequently* (mean: 3.49 on a 5-point Likert-type scale). This suggests that first attempts typically succeed. However, this also shows that 78.7% of participants (70 out of 89) occasionally fail at their first attempt and must make additional attempts to resolve a merge conflict. We also observe

Table 10 Frequency of failure on first attempts at merge conflict resolution from *Processes Survey*

	Frequency	Selections[i]	Percentage	Dev. Experience[ii]
1	Very frequently	4	(4.49%)	6.00 years
2	Somewhat frequently	13	(14.61%)	7.31 years
3	Occasionally	26	(29.21%)	9.27 years
4	Somewhat infrequently	27	(30.34%)	10.15 years
5	Very infrequently	19	(21.35%)	9.42 years

[i]89 out of 102 participants (87.26%) indicated a frequency

[ii]Mean software development experience for participants that indicated a specific frequency

that the frequency of failed first attempts follows software development experience; where the least experienced developers experience failed first attempts most often.

Furthermore, we asked survey participants to describe their backup strategies when their first attempt at resolving a merge conflict fails. We received 75 responses and the aggregate results are presented in Table 11, ranked according to the percentage of participants that described using each backup strategy.

Developers' backup strategies include *take it offline* (B1), *collaborating* (B2), *try again* (B3), *redoing changes* (B4), and *no backup strategy* (B5). Since *no backup strategy* (B5) is not a strategy in and of itself, we focus on strategies B1–B4 instead.

The *take it offline* (B1) strategy involves moving conflicting code away from shared branches or code repositories, and working locally to resolve the conflict without disrupting other developers. The antithesis of this strategy is *collaborating* (B2), where developers seek out other developers that are more knowledgeable about the area of conflicting code. The B1 and B2 strategies contrast each other, and show that developers reserve more costly strategies (in terms of time, effort, and coordination) as backups to their primary resolution strategies. The most common backup strategies are reactionary in nature, which is different from the action-oriented nature of the primary strategies (Table 7).

Additionally, we find that developers also simply *try again* (B3) to merge the same code together and hope that their tools are able to succeed with a second attempt. Developers also resort to *redoing changes* (B4), by way of reverting and manually recreating the changes found in conflicting commits when their initial attempt failed. The B3 and B4 strategies appear to cement the extremes of the cost spectrum of backup strategies for resolving merge conflicts. Simply retrying the same merge (B3) requires very little additional work. It implies that developers think that they might have missed something, and that by

Table 11 Backup strategies for resolving merge conflicts from *Processes Survey*

Strategy	Description	Participants[i]	Percentage[i]
B1	Take it offline	19	(25.33%)
B2	Collaborating	17	(22.67%)
B3	Try again	15	(20.00%)
B4	Redoing changes	14	(18.67%)
B5	No backup strategy	10	(13.33%)

[i]75 out of 102 participants (73.53%) provided a description of their backup strategy

going through the changes again, they might catch or have a better understanding of the two changes that are conflicting. However, the process of redoing changes (B4) is a duplication of previous efforts. This *Nuclear Option* is clearly a time-consuming strategy for developers (both in planning and implementing a resolution), and yet the perceived costs of trying to unravel the conflicting code appear to be higher than the costs of reimplementing features.

Finally, an interesting result is that some developers do not have a strategy for approaching a merge conflict resolution. The existence of this *no strategy* approach is anecdotal, but curious, since we assume that developers are rational actors seeking to organize themselves in ways that increase the likelihood of successful outcomes. Yet this strategy appears to go counter to that notion. One explanation for the lack of a strategy is the lack of experience. With a mean of 3.5 years of programming experience (5.6 years less than the overall mean), these participants might not have encountered enough situations to form a coherent strategy.

Interestingly, when developers perceive a merge conflict to be too difficult to resolve they occasionally resort to removing all conflicting code and reimplementing the underlying functionality in order to fix it.

8 What Difficulties Do Software Developers Experience When Managing Merge Conflicts? (RQ4)

To understand the difficulties software developers face when managing merge conflicts, we asked interview participants to reflect on situations when they faced a merge conflict. Based on responses in the *Exploratory Interviews*, we asked *Barriers Survey* participants to rate the resulting factors and needs.

8.1 Difficulty Factors

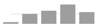
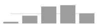
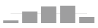
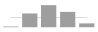
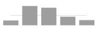
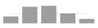
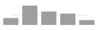
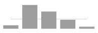
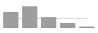
We identified nine factors that developers consider when approaching a conflict and attempting to determine its difficulty (see Table 12). We asked *Barrier Survey* participants to rate how each of these nine factors affected their perceptions of difficulty when approaching a merge conflict.

We received 162 responses and present the aggregated results in Table 12; ranked according to the mean score for each factor. Here, we discuss in detail the top 4 factors with a mean score greater than 3.00. These factors can be grouped into *technical aspects* and *expertise,* and our results are presented according to these groups.

8.2 Technical Aspects

Two of the top four factors refer to the perceptions about the complexity of merge conflicts (F1, F3), with the fourth factor being *number of conflicting lines of code* (F4), which can be construed as a specific metric for the complexity of the conflict. While developers mentioned complexity of the lines of code and the file, none mentioned using any metrics, such as cyclomatic complexity (Fenton and Ohlsson 2000; McCabe 1976) or Function Point Analysis (Garmus and Herron D 2001; Symons 1988). Instead, developers made educated guesses on the complexity of the code based on their own experience of either writing the code, or having worked with it. Some of the simple-to-compute metrics, such as the *number of conflicting lines of code* (F4), the *number of files in the conflict* (F8), the *atomicity of changesets in conflict* (F6), and the *time to resolve a conflict* (F5) were mentioned.

Table 12 Difficulty factors of merge conflicts from *Barriers Survey*

Factor	Description	1 2 3 4 5	Median[i]	Mean[i]
F1	Complexity of conflicting lines of code		4	3.52
F2	Expertise in area of conflicting code		4	3.50
F3	Complexity of files with conflicts		3	3.23
F4	Number of conflicting lines of code		3	3.14
F5	Time to resolve a conflict		3	2.82
F6	Atomicity of changesets in conflict		3	2.80
F7	Dependencies of conflicting code		3	2.78
F8	Number of files in the conflict		3	2.68
F9	Non-functional changes in codebase		2	2.16

[i]Responses on 5-point Likert-type scale indicating the degree of effect on resolution difficulty (1 indicates *no effect*, 5 indicates *great effect*)

Research has shown (Gil and Lalouche 2017) that the size of the code is the most important predictive feature for external factors (e.g. bugs) of all proposed complexity measures. This suggests that it might be enough for developers to rely on it when assessing the complexity of the conflicting code. However, we find that the impacts of size differ from those of complexity measures when developers provide their own definitions for these measures (see Section 9.6).

The only factor where static analysis tools can help was in identifying the *dependencies of conflicting code* (F7). This indicates that understanding the complexity of the conflicting code is important, but developers do not use the metrics that have been proposed by research. While some of the simple proxies for complexity are used, developers primarily rely on their own judgement of the complexity of a conflict. This perception of the conflict complexity can affect whether a developer resolves the conflict immediately, or whether they should wait to examine the conflict when further resources are available. In the interviews, P8 commented:

> *"Small is always easy. A 1-line merge conflict is always easier to resolve than a 400-line merge conflict."*

If a merge conflict is perceived to be large or complex, a developer may decide to forgo attempting to resolve it through code manipulation and choose to revert the changes instead (Guzzi et al. 2015). This *"nuclear option"* requires developers to disrupt the development flow, set aside their current development work, and potentially remove good, working code that was not part of the conflict in order to return to a non-conflicting state. In the interview, P1 describes this process as:

> *"If you have many conflicts involved, many commits in the conflict... throw one of the branches away. You cannot combine tens of commits conflicting... it's not sane!"*

Further, when integrators are preparing code for production environments they prioritize merge conflicts for code review based upon the perceived difficulty of resolving the affected code. We find that these decisions rely on human judgement factors as much as they rely on data-driven metrics. Developers may not have the time to compute project-wide complexity metrics, such as those proposed in literature. Instead, they use educated guesses and intuition based on familiarity with the codebase; either from writing the code, or having worked with it. Therefore, we need metrics that can be easily calculated by unexperienced developers as they face a conflict.

8.3 Expertise

Our findings show that the *expertise in the area of conflicting code* (F2) is one of the top factors in determining the difficulty of a merge conflict. This reiterates the fact that developers rely on their own knowledge about the conflicting codebase when approaching a conflict. And as seen in Section 7, this expertise has a direct impact on the ability of developers to use *code looks correct (i.e. visual test passes)* (C3) as a strategy for evaluating merge conflict resolutions.

Our results indicate that when developers feel they don't have the expertise in the conflicting codebase, they consider the conflict difficult to merge and seek out more information or assistance from others. P5 illustrated this need for expertise when describing his workflow:

> *"A lot of what I work on is in my own little area . . . I know what to do [. . .]. But in [an unfamiliar part of the code,] then I'll get someone else to resolve the merge conflict for me. It's someone else's code, and I don't want to screw it up."*

Our findings confirm the need for tools that identify appropriate experts (Costa et al. 2016) and encourage further research into selection of knowledgeable developers for merge conflict resolution.

8.4 Unmet Needs for Merge Conflict Resolutions

There can often be gaps in how developers perceive the difficulty of merge conflicts and the actual hurdles that they face when resolving these conflicts. These gaps can then in turn affect how effective developers are at resolving the conflict.

We, therefore, asked our interview participants open-ended questions about their experiences in resolving the most recent conflicts, especially their recollection of what made the conflict resolution difficult. Their responses indicated that there are several unmet needs. We identified ten needs (see Table 13), which range from needs about the ability to understand the code, their expertise, and existing tool support.

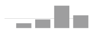
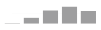
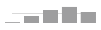
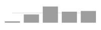
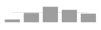
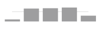
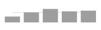
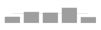
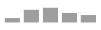
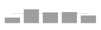
Using the results from the interview, we asked *Barriers Survey* participants to rate how much each of the ten needs affected their ability to resolve the merge conflicts. We received 141 responses using a 5-point Likert-type scale indicating the the effect on resolution difficulty, with 1 being *Not at all*, 3 being *A moderate amount*, and 5 being *A great deal.* Results of the survey are presented in Table 13.

All the unmet needs have a mean score of at least 3.03 on the 5-point Likert-type scale, implying that all of them mattered at least a moderate amount. We present and discuss in detail the top four unmet needs, plus additional observations regarding the other six unmet needs. As with the factors in the previous section, all these needs also relate to *technical aspects* (e.g., understanding the conflicting code) and their *expertise* in resolving conflicts.

8.5 Technical Aspects

Three needs among the top four relate to technical aspects of merge conflict resolution. The *understandability of conflicting code* (N1) is ranked as the most important need, with both

Table 13 Developer needs for merge conflict resolutions from *Barriers Survey*

Need	Description	1 2 3 4 5	Median[i]	Mean[i]
N1	Ease of understanding conflicting code		4	3.89
N2	Expertise in area of conflicting code		4	3.72
N3	Amount of info about conflicting code		4	3.62
N4	Tools presenting understandable info		3	3.48
N5	Changing assumptions within code		3	3.30
N6	Complexity of project structure		3	3.18
N7	Trustworthiness of tools		3	3.12
N8	Informativeness of commit messages		3	3.07
N9	Project culture		3	3.04
N10	Tool support for history exploration		3	3.03

[i]Responses on 5-point Likert-type scale indicating the degree of importance to merge resolutions (1 indicates *no importance*, 5 indicates *great importance*)

contextual information about the conflict (N3) and *the way in which tools present relevant information* (N4) ranking in the top four.

Data from version control systems is used by developers to identify the evolution of the code (Codoban et al. 2015). However, it is not easily available and requires a context switch from the code editor to the version control system (Guzzi et al. 2015). Moreover, these changes are often processed in isolation, especially when there are many changes (conflicts) to process. Such decomposition of overall conflicting changes into smaller "chunks" is needed to be able to manage the complexity of the resolution process. However, this occludes viewing the changes in a larger context. Often developers deal with the decomposed (smaller) changes, hoping that they will work well together. For example, P1 compared the resolution hurdles between two conflicts, where one was simple, and the other spanned multiple files and complex blocks of code.

> *"You focus on understanding the small change, not the big one. It's easier to understand... get the small change to go with the flow of the bigger change."*

Another challenge when viewing changes in isolation is the fact that developers may miss the impact of the changes made as part of the resolution to the rest of the code base. Identifying the impact of changes on the rest of the code base has been repeatedly found to be a problem in collaborative development (de Souza and Redmiles 2008; Guzzi et al. 2015).

The top unmet needs in our study also revolved around the challenges that developers face in how much information they had about the conflicting code (N3), and the difficulty in finding the needed information from current tools and practices (N3, N4, N8, N10). This indicates that despite advances in supporting parallel development practices, the right information needed to resolve conflicts is still not easily available to developers.

Conflict resolution can sometimes lead to defects in the code base. This can arise for several reasons. For example the rationale of the two conflicting changes might be unclear and the merge might cause unintentional problems down the line. Or the resolved changes might not follow rigorous code review and testing to which the original changes were subjected. Therefore, even when the developer understands the particular conflicting code, they may still need additional meta-information about the rationale of changes and idea of future feature implementation. This is especially true in situations where the code base is old, and such information not readily available. During our interview, P7 commented:

> *"It's harder to merge code when you're merging in some legacy code... But if you're a young team, and everybody who wrote the code is still a part of the team, it's easier."*

8.6 Expertise

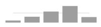
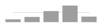
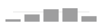
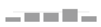
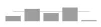
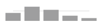
Knowledge is a key component of developers' needs when resolving merge conflicts. Along with general knowledge there is a need for expertise in the specific areas of code involved in a conflict. Developers recognize this need as having a sizable effect on their ability to resolve a merge conflict, and selected *expertise in the area of conflicting code* (N2) as the second most important need.

Examining code artifacts, reviewing change history, and reading documentation helps with understanding the code when they are present and well-maintained. However, locating and maintaining these supporting documents is not always possible. In fact, Forward and Lethbridge (2002), in a survey of 48 software developers, found that 68% either agreed or strongly agreed that documentation is always outdated. When these gaps arise, developers compensate by consulting experts in the area of conflicting code instead.

This result aligns with the goals of the TIPMerge tool (Costa et al. 2016), which seeks to locate experts that are best suited to resolve conflicts in a particular area of code. However, TIPMerge, as well as other recommendation tools are not being used by real-world developers, as evidenced by the lack of such tools in the list of top 10 merge awareness tools (Table 4) and merge resolution tools (Table 15). The reason for this lack of research tools adoption requires further investigation.

Another surprising fact was that while the informative nature of commit messages (N8) and project culture (N9) were mentioned, they were not as highly ranked. We had expected them to be higher based on prior work (Yamauchi et al. 2014; Hindle et al. 2009; Cortés-Coy et al. 2014; Hattori and Lanza 2008). We found no statistical differences between commercial or open source projects, including when accounting for experience levels. Our results indicate that team practices, including writing commit messages may have matured enough, such that these factors are no longer considered critical in our sample set.

Table 14 Desired improvements to merge toolsets from *Barriers Survey*

Imp.[i]	Description	1 2 3 4 5	Median[ii]	Mean[ii]
I1	Usability		4	3.43
I2	Filtering of relevant information		4	3.41
I3	Support for exploring project history		3	3.30
I4	Graphical information presentation		3	3.14
I5	Transparent tool functionality/operations		3	2.82
I6	Terminology consistent with other tools		2	2.53

[i]Imp. = Improvement

[ii]Responses on 5-point Likert-type scale indicating the degree of potential impact on merge conflict processes (1 indicates *no impact*, 5 indicates *great impact*)

Table 15 Merge resolution toolsets (top 10) from *Barriers Survey*

Tool	Description	Participants[i]	Percentage[i]
Git	Version Control System	37	(15.68%)
Vim/vi	Text Editor	17	(7.20%)
Text Editor (unspecified)	Text Editor	14	(5.93%)
Git Diff	Diffing Tool	11	(4.66%)
GitHub	Website	11	(4.66%)
Eclipse	IDE	10	(4.24%)
KDiff3	Diff & Merge	9	(3.81%)
Meld	Diff & Merge	8	(3.39%)
SourceTree	Git/Hg Desktop Client	8	(3.39%)
Sublime Text	Text Editor	7	(2.97%)

[i]Survey participants were allowed to provide multiple tools. Each entry represents the number (and percentage) of participants that responded with that particular tool. 115 out of 162 participants (70.99%) indicated the use of at least one merge resolution tool

9 How Well Do Tools Support Developer'S Needs for Managing Merge Conflicts? (RQ5)

Development tools need to be easy to use and provide contextualized, pertinent information in a manner that is easy to understand. To investigate how well current tools satisfy the needs of developers, we asked interview participants open-ended questions about how they resolve merge conflicts. We also ask about improvements that would be most valuable to them.

We framed the *Barriers Survey* questions to validate the improvement needs expressed in our interviews, and ranked those six needs according to mean score. Table 14 presents the needs from the survey responses ordered by their mean scores. We received 119 responses using a 5-point Likert-type scale to indicate the usefulness of each type of tool improvement (1 being *Not Useful*, 3 being *Moderately Useful*, and 5 being *Essential*).

Our results indicate that developers use a wide range of tools, with many directly using the Git command line interface. Our interview participants mentioned six different dimensions along which they would like improvements to tool support.

In addition, we also asked *Barriers Survey* participants which tools they use during conflict resolution. We identified 105 different tools from the 115 responses. Some mentioned generic responses such as *"text editor,"* for which we create a separate category.

Table 15 lists the top 10 most common tools used by participants to resolve merge conflicts.

In examining the list of these tools, we note that developers most often use basic tools (e.g. Git, Vim/vi, or a Text Editor) to handle merge conflicts instead of employing specialized tools or plugins to modern IDEs. In this list, there is only one IDE (Eclipse), and three diff/merge toolsets (Git Diff, KDiff3, and Meld). Along with the list of toolsets used for evaluating whether a merge resolution was successful (see Table 9), we find that developers lack proactive conflict detection tools and code analysis tools that could address many of the developer needs (see Table 13) and desired improvements (see Table 14).

We next discuss the top four improvements rated by survey participants. These are the responses that have a mean value higher than 3.00.

9.1 Better Usability

Usability is an important factor that determines whether a toolset supports or hinders the developer's workflow. Our *Barriers Survey* results indicate that *better usability* (I1) is the most desired improvement of toolsets used for conflict resolution.

While usability of a particular tool is important, the usability concerns become even more pertinent when they span multiple tools that are similar and must operate in sync with each other. Survey results indicate that participants use an average of 2.5 tools, and as many as 7 tools, to resolve merge conflicts.

For instance, in our interview, P1 demonstrated how he typically resolved a merge conflict by using four different tools and said:

> *"I have to jump around between tools and copy and paste version numbers...this is why integration matters."*

Switching across multiple tools while resolving a conflict is disruptive and comes at a cost. Psychology studies (Meiran 2000; Gopher et al. 2000) have shown that task switching reduces performance and causes mental fatigue. Gerald Weinberg highlighted that context switching arising from toolset fragmentation is a big problem in engineering teams (Weinberg 1992).

9.2 Better Exploration of Project History

Developers have been known to use historical data to understand code evolution and development processes (Codoban et al. 2015). Version control and bug tracking systems contain a huge amount of meta-information about the evolution of code and development processes. However, it is not easy to find the right bit of information in these large systems. Currently, there is insufficient support for performing detailed analysis of how a code snippet evolved over time and why. Better ways of exploring the project history (I3) was one of the top requested improvements in our survey. As P1 mentioned in the interview:

> *"Give me a way to explore the history. To drill down, to go back up, you know? To resurface and understand what happened."*

Currently, when performing any complex analysis it is easier to write stand alone scripts to extract the information. During the interview, P1 mentioned that he has written several scripts to locate particular historical commits that relate to a current merge conflict. Similarly, P9 described a tool, `git-diff`, that was developed by their team to add additional difference analysis functionality across branches:

> *"git-diff will just do the diff based on the SHAs... we're adding metadata... It also hooks into GitHub labels to do some more advanced heuristics."*

While writing these scripts allows extraction of relevant data contextualized to the need, it also leads to a proliferation of multiple scripts that are written by individual developers and need to be maintained or integrated. This further adds to the problem of context-switching when developers must switch between multiple tools, and execute multiple scripts.

We are not the first to recognize the gap in tool support provided for analyzing development history among developers (Codoban et al. 2015; Sun et al. 2015; Guo et al. 2016; Yan et al. 2014). It appears that practical applications of history exploration are still beyond the reach of developers. One of the reasons for this might be the simple set of text editors, and toolsets, that our study participants seem to prefer.

9.3 Better Filtering of Less-Relevant Information

Tools that routinely handle large or complex datasets require filtering in order to efficiently locate desired pieces of information. For example, when there are several commits in a pull request and multiple code reviews documented across the new code. It is difficult to extract the key issue in the pull request, which can get lost in the sea of low level details. Similarly, if there are multiple commits in a pull request or branch, it is hard to extract the right information. Therefore, tools that provide filtering can better assist developers in working with large amounts of metadata associated with the changes. *Better ways of filtering out less relevant information* (I2) was selected as the second most important need; P1 explained:

> *"You want to extract the relevant commits. The ones that actually clash...you want to zoom in on them and understand just enough and don't waste time."*

While improvements in history exploration (I3) will make project metadata more accessible, improvements in filtering for relevant metadata will allow developers to focus on the relevant parts of the code impacted by the merge conflict.

9.4 Better Graphical Presentation of Information

The usefulness of information is helped or hindered by the way in which it is presented to users. In our survey results, we found that *better graphical presentation of information* (I4) was ranked the fourth highest improvement needed (mean: 3.14).

In our interviews, several developers reported experiencing issues with inconsistent terminology, inconsistent visual metaphors (e.g. colors, notifications, etc.), and the organizational layout of different development tools. The cost of context switching in software development is well-known to researchers (Czerwinski et al. 2004; Li et al. 2007; Blackwell and Burnett 2002; Convertino et al. 2003), and our results indicate that switching between different terminology and information presentation styles can also be a problem. There is a need for tools that share commonality in both terminology and presentation.

9.5 Tool Mistrust/Transparency

Most merge tools attempt to resolve conflicts using a variety of algorithms, but revert to manual resolution when these algorithms fail. Several interview participants indicated that they mistrust merge tools when they obscure the steps and rationale for particular results when resolving merge conflicts. The opaque nature of history exploration tools was also found to be a source of developers' overall mistrust of their toolsets. P4 commented:

> *"I've never trusted the merge tools or diff tools... Sometimes I'll even manually go and do the merge myself rather than use a tool. Just because I've had several times where it's a bad merge, and I broke some code."*

Based upon this theme of mistrust, we asked *Barriers Survey* participants to rate the degree to which they trust their merging, history exploration, and conflict resolution tools. We received 121 responses to this question, with a mean score of 3.66, placing the most common responses between *a moderate amount* and *a lot* of trust (Table 16). Assuming that responses of *a moderate amount*, *a little*, or *not at all* indicate some degree of mistrust, we find that 42.15% of developers experience some gap in toolset trust.

However, the severity of toolset mistrust is not as significant as our interview results suggested. Only 8.26% of developers indicated that they trust their toolset *a little* or *not*

Table 16 Degree of trust for merging, history exploration, and conflict resolution tools from *Barriers Survey*

	Trust	Selections[i]	Percentage	Dev. Experience[ii]
1	Completely	20	(16.53%)	6-10 years
2	A lot	50	(41.32%)	11-15 years
3	A moderate amount	41	(33.88%)	11-15 years
4	A little	10	(8.27%)	6-10 years
5	Not at all	0	(0.00%)	0 years

[i] 121 out of 162 participants (74.69%) indicated a degree of trust

[ii] Mean software development experience for participants that indicated a specific degree of trust

at all (10 out of 121 responses). As the results of the *Barriers Survey* were counter to our interview results, we looked further. We found that: (1) participants reported on the trust levels of the tools that they regularly use, (2) some participants reported that they had discontinued using toolsets when they ran into errors, and (3) that trust in tools is fluid and changes based on the situations in which they are employed. Although we were unable to capture participant trust levels in these discontinued tools, we can observe that participants use (and therefore trust) simple tools more often than complex tools.

9.6 Perceptions of Tool Effectiveness

The perceived size and complexity of merge conflicts affect the way in which developers plan, allocate, and enact resolutions. To understand the degree to which these two factors impact developers' perceptions about the effectiveness of their toolsets, we asked *Barriers Survey* participants to rate their toolset across four different merge conflict archetypes: (A1) *simple, small merge conflicts*, (A2) *simple, large merge conflicts*, (A3) *complex, small merge conflicts*, and (A4) *complex, large merge conflicts*.

Individual participants have different toolsets, and consider different factors when determining the perceived size and complexity of a merge conflict. We therefore instructed participants to rate their own toolset using their notion of what constitutes *simple* vs. *complex* and *small* vs. *large* merge conflicts.

Figure 3 provides a visual illustration of the results of this survey question. The four plots display the results for each of the archetypes, with archetype (A1) in the top-left plot, (A2) in bottom-left plot, (A3) in the top-right plot, and (A4) in the bottom-right plot. Individual plots are composed of a horizontal axis containing participants' software development experience, which we collect since experience can determine the range of conflicts that they have faced and their perceptions. The vertical axis shows the range of possible responses for the effectiveness of merge toolsets. The size and number within each bubble represent the number of respondents with a particular amount of software development experience that rated their toolset at that specific effectiveness level.

For example, a practitioner with 6-10 years of experience who indicates that her merge toolset is *extremely effective* for *small, simple merge conflicts* would be represented in the largest bubble (containing 19) in A1. She would also be represented in the largest bubble (containing 13) in the bottom-right plot (A4) if she indicated that her merge toolset was *moderately effective* for *large, complex merge conflicts*.

Observing the overall trends when moving between plots, we find that developers perceive complexity of the conflict to have a greater impact on the effectiveness of their merge

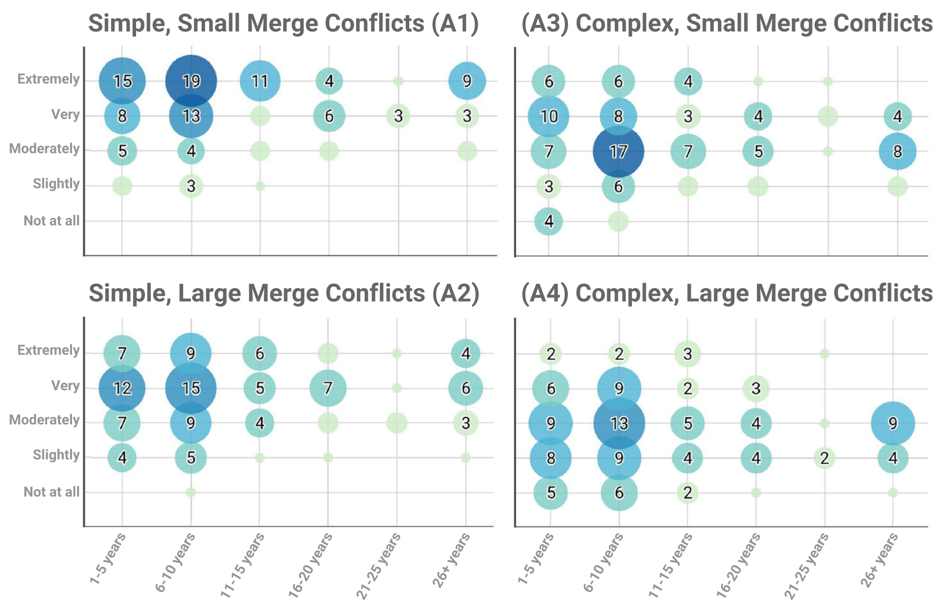


Fig. 3 Effectiveness of developers' toolsets in supporting perceived size and complexity of merge conflicts, split on software development experience. Bubble values indicate number of *Barriers Survey* responses for effectiveness of a particular merge conflict size and complexity, and bubble sizes indicate the number of responses for comparison

toolsets than the size of merge conflicts. Numerical analysis confirms this when finding that the mean response for archetype (A1) is 4.278 (where 5 is *extremely effective* and 1 is *not effective at all*), (A2) is 3.782, (A3) is 3.347, and (A4) is 2.783. The shift from *small* to *large* merge conflict size (A1 to A2) results in a difference in mean responses of 0.496, whereas the shift from *simple* to *complex* merge conflict complexity (A1 to A3) results in a difference in mean responses of 0.930.

Examining this trend through the lens of software development experience, we find that more experienced developers perceive a greater disparity in the effects of increasing merge conflict complexity (as opposed to merge conflict size) on the effectiveness of their merge toolsets. Table 17 illustrates the numerical analysis between each of the software development experience groups defined in the *Barriers Survey*.

When comparing the aggregate differences between shifting from *small* to *large* merge conflict size (A1 to A2) and shifting from *simple* and *complex* merge conflict complexity (A1 to A3), we see that participants with 1-5 years of software development experience have a 0.367 gap in dimensional differences (complexity vs. size mean differences). This gap widens as the number of year of software development experience increases, until participants with 26+ years of software development experience have a 0.786 gap in dimensional differences. The exception to this pattern is the small group of participants with 21-25 years of software development experience (10 participants, 4.42% overall), who indicate that the size of a merge conflict has a greater impact on the effectiveness of their merge conflict toolsets. We find that this group uses the same merge toolsets as the larger participant population (see Table 15), which suggests that concerns vary among experienced developers.

Table 17 Mean effectiveness of developers' toolsets across software developer experience and merge conflict archetypes from *Barriers Survey*

Experience[i]	A1[ii]	A2[ii]	A3[ii]	A4[ii]	MC Size[iii]	MC Complexity[iv]	Diff.[v]
1-5 years	4.200	3.733	3.367	2.733	0.467	0.833	0.367
6-10 years	4.231	3.667	3.256	2.795	0.564	0.974	0.410
11-15 years	4.438	4.000	3.563	3.000	0.438	0.875	0.438
16-20 years	4.167	3.833	3.333	2.750	0.333	0.833	0.500
21-25 years	4.250	3.750	4.000	3.000	0.500	0.250	−0.250
26+ years	4.500	3.929	3.143	2.571	0.571	1.357	0.786

[i] Software development experience of participants

[ii] Mean response for the archetype; where 5 is *extremely effective* and 1 is *not effective at all*

[iii] Difference in mean responses when shifting from *small* to *large* merge conflict size (A1 to A2)

[iv] Difference in mean responses when shifting from *simple* to *complex* merge conflict complexity (A1 to A3)

[v] Aggregate difference between MC Complexity and MC Size (negative indicates a reverse relationship)

These results suggest that merge tools are currently equipped to handle increases in the size of merge conflicts, but not as well equipped for increases in complexity. Additionally, more experienced developers perceive this gap to be greater than less experienced developers, suggesting that experienced developers have greater unmet needs for tool support when merge conflict complexity is unavoidable. The increasing amount of code being developed in distributed environments means that scaling support in both dimensions is necessary to accommodate all developers' needs.

10 Implications

10.1 For Tool Builders

Version control systems provide an easy method for storing and retrieving recent development history, but examining older development history at scale and in a usable manner has not completely met developers' expectations. Tool builders should work to address this unmet need by leveraging research in search systems for developer-assistance (Nabi et al. 2016) and machine learning-based code assistance (Bradley and Murphy 2011) to provide intuitive and expressive tools for history exploration.

Even if developers use a *reactive* monitoring approach for detecting merge conflicts, better tool support can make their lives easier. For example, instead of notifying a developer that a merge conflict has occurred, adding an annotation within tools indicating the type of conflict might assist developers. This type of contextualized information would allow developers to more precisely know how urgent the merge conflict is, without having to interrupt their workflow.

Developers indicate that current merge toolsets do not scale to handle large, complex merge conflicts (see Section 9.6). To address this concern, tool builders should look at consolidating feature sets that currently span multiple tools in order to provide better usability (I1 from Table 14). Tool builders should also add more expressive search and filtering features for both project history and meta-information related to merge conflicts (I2, I3), to ease the frustration of developers that must understand the context and evolution of code involved in the conflict.

Before starting a merge conflict resolution, we found developers having to "guesstimate" the difficulty of the conflict resolution to decide whether to work on it now or deffer it, whether to integrate the changes or simply start over. Prediction tools that identify the complexity of conflicts and difficulty of resolution can help alleviate this. This will provide to developers with enough information to make accurate and informed decisions in order to prevent further issues down the line.

When it comes to evaluating the result of a merge conflict resolution, we found that developers employ various strategies (Table 8). Developers mentioned that *passing tests* (C1) and *successful compilation* (C2) are some of the criteria for success. However, for large projects, running the test suite can be time intensive. Tools can help developers by providing information or running only tests that are impacted by the conflict resolution. Test selection techniques, such as regression testing (Gligoric et al. 2015), can benefit from using this conflict information to more quickly inform developers about possible bugs. This would help developers as they would have to rely less on their own intuition (C5), when it comes to evaluating the result.

The fact that developers mention code complexity as one of the main factors in deferring a merge conflict resolution, and that developers "eyeball" the resolutions, seems to be an indication that evaluation might be a problem in the resolution. Merge conflicts are perceived as difficult because *the evaluation of the results are difficult.* In this case, tools should provide better support for developers when they evaluate their resolution.

Finally, most developers have failed at least once in resolving a merge conflict resolution (Table 10). Tools can provide better insight into why the resolution has failed, so developers have information to formulate their next steps. Currently, when a merge conflict resolution fails, developers have to interrupt their workflow by taking the resolution offline (B1 from Table 11), or they have to ask for help, which has the potential of interrupting other developers (B2). If developers had more information about *why* the merge conflict resolution failed, they might be able to recover more efficiently.

10.2 For Developers

Understanding the constraints of the different phases involved in merge conflict resolution can allow more effective prevention and management of merge conflicts as they occur.

Developers indicate that understanding code, having appropriate information, and dealing with complex codesets are key themes of difficulty when working with merge conflicts (Sections 5, 6). Existing tool support can help with some of these issues, but developers also need to educate themselves on development processes that prevent and alleviate the severity of merge conflicts. For example, the number of conflicting files and the size of changes are considered important factors (Section 8.1). Researchers (Brindescu et al. 2014) have previously found that developers using distributed version control systems commit more often, and that committing more often makes debugging easier when something breaks (Meyer 2014). Therefore, developers should strive to make smaller commits, and commit often.

Other agile development processes such as continuous integration, iterative development, and branch merging policies are known to facilitate development in large, distributed teams. However, not all developers are actively using such techniques (Phillips et al. 2011), and large organizations will require additional diligence in order to address the increased possibility of conflicts occurring. Developers should integrate proactive merge conflict monitoring into their own processes, and seek to add tools that enable proactive monitoring within their organizations.

The most experienced developers indicate that their toolsets are least equipped to handle increases in merge conflict complexity (Table 17). If experienced developers struggle to use merge toolsets for complex merge conflicts, then less experienced developers are also likely to encounter increasing difficulties as they gain experience dealing with complex merge conflicts. Therefore, developers should take steps to reduce the complexity of merge conflicts by using proactive merge conflict detection tools, create well-defined code ownership boundaries, and reduce the time between commits.

10.3 For Researchers

The life-cycle of merge conflicts is a preliminary model for understanding the phases involved in a merge conflict. Additional research is needed to expand this model, such as to understand which strategies are most (and least) effective at addressing the needs of developers in each phase of the cycle.

Version control systems (VCS) feature heavily in the top merge awareness tools (Table 4), the top merge resolution tools (Table 15), and the top merge resolution evaluation toolsets (Table 9). Therefore, researchers should also examine the usage patterns for different features within VCS tools across the life-cycle phases to understand the evolution of developer needs with their tools.

The size of a merge conflict is often considered an approximation for complexity. However, our findings show that developers perceive size and complexity to be different factors in merge conflict resolution and tool effectiveness (Table 17). Which suggests that researchers need to further investigate measures that can be used to better approximate merge conflict complexity.

The top factors that impact the assessment of merge conflict difficulty are primarily focused on program comprehension (F1, F3, F4 from Table 12). Program comprehension has been an important research focus, with entire conferences dedicated to it. Previous research has explored tool support and visualizations to help comprehend programs, both small and large. Although similar, merge comprehension involves understanding the flow of changes across time in two parallel streams, whereas program comprehension primarily focuses on understanding the current state of code. Therefore, tools and visualizations for merge conflicts must embrace these unique constraints in order to provide relevant information for developers. Our results indicate that developers still have unmet needs along the following dimensions: (1) comprehending code snippets in isolation, (2) understanding the code context underlying multiple code snippets that are split across multiple files, and commits, and; (3) the ability to quickly comprehend the complexity of these code snippets.

Developers indicate that their needs during merge conflict resolutions center around the retrieval, organization, and presentation of relevant information (N1, N3, N4 from Table 13). With the variety of meta-information available across different toolsets, and the inconsistent use of terminology, there is a need for standardization and best practices to be developed. Standardization efforts would likely help to alleviate some of the mistrust of merging tools that developers have expressed. However, researchers should investigate the margin of errors that are tolerated by developers to determine the context in which developers discontinue use of tools.

Expertise is seen as both a significant factor that affects the assessment of merge conflict difficulty (F2), and an important need for developers to effectively resolve the conflict (N2).

We have also seen that experience can play a part in developer's assessment of the success of a merge conflict resolution (C3).

Previous work has focused on recommending developers best suited to perform a collaborative merge based on the previous edits to conflicting files (da Silva et al. 2015) or developers' experience across branches and project history (Costa et al. 2016). However, these efforts have resulted in tools that require standalone installation and execution. Our results indicate that developers are concerned about toolset fragmentation, and therefore adding an additional tool might be counterproductive to the workflow of most developers.

Our results show that developers mention that they *redo the changes* when a merge conflict resolution fails. This *Nuclear option* is very expensive. Tools should provide better merging support, so that developers can resolve a conflict and not have to scrap good code just because it happens to intersect with other changes.

Finally, we find that developers need to quickly estimate whether they can fix the conflict, and whether to resolve it now or delay the resolution. This indicates that developers need mechanisms to identify the skillsets required to complete the conflict resolution task, by viewing the code fragments (D1, D2). Research should investigate mechanisms to identify required skillsets by using information retrieval or machine learning techniques on the code fragment and past edits.

11 Threats to Validity

As in any empirical study, there are threats to validity with our work. We attempt to remove these threats where possible, and mitigate the effect when removal is not possible.

Construct Validity Interview questions were open-ended and designed to elicit developer opinions about the experiences, difficulties, and perceptions of merge conflicts. We determined particular factors and needs after concluding all interviews, and thus did not bias interview participants to only factors previously mentioned. We created survey questions using factors found through card-based unitization. This methodology allowed us to capture the common themes that developers experience when working with merge conflicts, but might have allowed themes specific to particular sub-groups to be unrepresented in our results.

Internal Validity Confounding and extraneous factors can affect conclusions relating to cause and effect. We lessen this effect by using multiple methods to triangulate our results, and compare against other datasets where appropriate. Because we use these methods to highlight stronger answers, this also means that we may have missed subtle trends across our data that could have been visible otherwise.

External Validity Interview results may not generalize to all developers due to a small sample size, but we reduce this effect by selecting interview participants from open- and closed-source projects, varying industries, and varying project sizes (see Table 1). To expand and confirm our interview results, we survey 264 developers on varying aspects to ensure our results match with trends in the larger software development community. We do not report a response rate for our surveys, since social media and mailing lists do not allow accurate measurement of the number of individuals that read our recruitment message and did not choose to participate.

12 Conclusion

In this paper we present insights into developers' processes and perceptions as they resolve merge conflicts. Through semi-structured interviews and two surveys, we investigate the problems that merge conflicts pose, from the developers perspective. We find that the majority of developers use a reactive process when monitoring for merge conflicts. They rely mostly on their own knowledge of the code and on the complexity of the conflict when deciding on how to approach it. While some developers defer the merge conflict resolution, it can lead to increased complexity, to the point that they might need to throw away changes and reimplement. Finally, we find that when the merge conflict resolution fails, developers will either take the work offline, or collaborate to successfully resolve it.

Providing insights into developers' processes and perceptions is critical, so that we can design tools that help resolve the issues facing developers. We provide actionable implications for researchers, tool builders, and developers to harness the results of our study. In future work, we plan to explore whether these factors, needs, and desired toolset improvements can be seamlessly merged into tools or techniques that assist developers' workflows.

Acknowledgments We thank Iftekhar Ahmed, Amin Alipour, Souti Chattopadhyay, Alex Hoffer, Michael Hilton, Sarah Nadi, Sruti Ragavan, and the anonymous reviewers for their valuable comments on earlier versions of this paper. We also thank all of our interview and survey participants, especially those who helped distribute survey links. This research was partially supported by NSF grants CCF-1439957, CCF-1553741, CCF-1560526, and IIS-1559657.

References

Accioly P, Borba P, Cavalcanti G (2018a) Understanding semi-structured merge conflict characteristics in open-source Java projects. Empir Softw Eng 23(4):2051–2085

Accioly P, Borba P, Silva L, Cavalcanti G (2018b) Analyzing conflict predictors in open-source Java projects. In: Proceedings of the 15th international conference on mining software repositories (MSR). ACM, pp 576–586

Apel S, Liebig J, Brandl B, Lengauer C, Kästner C (2011) Semistructured merge: rethinking merge in revision control systems. In: Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on foundations of software engineering (ESEC/FSE). ACM, pp 190–200

Apel S, Lessenich O, Lengauer C (2012) Structured merge with auto-tuning: balancing precision and performance. In: Proceedings of the 27th IEEE/ACM international conference on automated software engineering (ASE). ACM, pp 120–129

Beecham S, Hall T, Rainer A (2003) Software process improvement problems in twelve software companies: an empirical analysis. Empir Softw Eng 8(1):7–42

Begole JB, Tang JC, Smith RB, Yankelovich N (2002) Work rhythms: analyzing visualizations of awareness histories of distributed groups. In: Proceedings of the 2002 ACM conference on computer supported cooperative work (CSCW). ACM, pp 334–343

Beizer B (1984) Software system testing and quality assurance. Van Nostrand Reinhold Co.

Biehl JT, Czerwinski M, Smith G, Robertson GG (2007) FASTDash: a visual dashboard for fostering awareness in software teams. In: Proceedings of the SIGCHI conference on human factors in computing systems (CHI). ACM, pp 1313–1322

Binkley D, Horwitz S, Reps T (1995) Program integration for languages with procedure calls. ACM Trans Softw Eng Methodol (TOSEM) 4(1):3–35

Bird C, Zimmermann T (2012) Assessing the value of branches with what-if analysis. In: International symposium on the foundations of software engineering (FSE), p 45

Blackwell A, Burnett M (2002) Applying attention investment to end-user programming. In: Symposia on human-centric computing languages and environments (HCC), pp 28–30

Borg M, Alègroth E, Runeson P, 2017 Software engineers' information seeking behavior in change impact analysis: an interview study. In: The 25th IEEE international conference on program comprehension (ICPC). IEEE, pp 12–22

Bradley AW, Murphy GC (2011) Supporting software history exploration. In: Working conference on mining software repositories (MSR), pp 193–202

Brindescu C, Codoban M, Shmarkatiuk S, Dig D (2014) How do centralized and distributed version control systems impact software changes? In: Proceedings of the 36th international conference on software engineering (ICSE). ACM, pp 322–333

Brooks FP (1974) Mythical man-month. Datamation 20(12):44–52

Brun Y, Holmes R, Ernst MD, Notkin D (2011) Proactive detection of collaboration conflicts. In: International symposium and European conference on foundations of software engineering (ESEC/FSE), pp 168–178

Campbell JL, Quincy C, Osserman J, Pedersen OK (2013) Coding in-depth semistructured interviews: Problems of unitization and intercoder reliability and agreement. Sociol Methods Res 42(3):294–320

Cataldo M, Herbsleb JD (2008) Communication networks in geographically distributed software development. In: ACM Conference on computer supported cooperative work & social computing (CSCW), pp 579–588

Codoban M, Ragavan SS, Dig D, Bailey B (2015) Software history under the lens: a study on why and how developers examine it. In: International conference on software maintenance and evolution (ICSME), pp 1–10

Convertino G, Chen J, Yost B, Ryu YS, North C (2003) Exploring context switching and cognition in dual-view coordinated visualizations. In: International conference on coordinated and multiple views in exploratory visualization (CMV), pp 55–62

Cortés-Coy LF, Vásquez ML, Aponte J, Poshyvanyk D (2014) On automatically generating commit messages via summarization of source code changes. In: International working conference on source code analysis and manipulation (SCAM), pp 275–284

Costa C, Figueiredo J, Murta L, Sarma A (2016) TIPMerge: recommending experts for integrating changes across branches. In: International symposium on foundations of software engineering (FSE), pp 523–534

Czerwinski M, Horvitz E, Wilhite S (2004) A diary study of task switching and interruptions. In: SIGCHI conference on human factors in computing systems (CHI), pp 175–182

da Silva IA, Chen PH, Van der Westhuizen C, Ripley RM, van der Hoek A (2006) Lighthouse: coordination through emerging design. In: Proceedings of the 2006 OOPSLA workshop on eclipse technology eXchange. ACM, pp 11–15

da Silva JR, Clua E, Murta L, Sarma A (2015) Niche vs. breadth: calculating expertise over time through a fine-grained analysis. In: International conference on software analysis, evolution and reengineering (SANER), pp 409–418

de Mello RM, Travassos GH (2016) Surveys in software engineering: identifying representative samples. In: Proceedings of the 10th ACM/IEEE international symposium on empirical software engineering and measurement, ESEM '16. ACM, pp 55:1–55:6

de Souza CR, Redmiles D, Dourish P (2003) Breaking the code, moving between private and public work in collaborative software development. In: International conference on supporting group work (GROUP), pp 105–114

de Souza CRB, Redmiles DF (2008) An empirical study of software developers' management of dependencies and changes. In: International conference on software engineering (ICSE), pp 241–250

Dig D, Manzoor K, Johnson RE, Nguyen TN (2008) Effective software merging in the presence of object-oriented refactorings. IEEE Trans Softw Eng (TSE) 34(3):321–335

Easterbrook S, Singer J, Storey MA, Damian D (2008) Selecting empirical methods for software engineering research. In: Guide to advanced empirical software engineering. Springer, pp 285–311

Estler HC, Nordio M, Furia CA, Meyer B (2013) Unifying configuration management with merge conflict detection and awareness systems. In: Proceedings of the 22nd Australian software engineering conference (ASWEC), pp 201–210

Estler HC, Nordio M, Furia CA, Meyer B (2014) Awareness and merge conflicts in distributed software development. In: International conference on global software engineering (ICGSE), pp 26–35

Fenton NE, Ohlsson N (2000) Quantitative analysis of faults and failures in a complex software system. IEEE Trans Softw Eng (TSE) 26(8):797–814

Fereday J, Muir-Cochrane E (2006) Demonstrating rigor using thematic analysis: a hybrid approach of inductive and deductive coding and theme development. Int J Qualitat Methods 5(1):80–92

Forward A, Lethbridge TC (2002) The relevance of software documentation, tools and technologies: a survey. In: ACM symposium on document engineering (DocEng), pp 26–33

Fritz T, Murphy GC (2010) Using information fragments to answer the questions developers ask. In: International conference on software engineering (ICSE), pp 175–184

Fusch PI, Ness LR (2015) Are we there yet? Sata saturation in qualitative research. Qual Rep 20(9):1408

Garmus D, Herron D (2001) Function point analysis: measurement practices for successful software projects. Addison-Wesley Longman Publishing Co

Garrison DR, Cleveland-Innes M, Koole M, Kappelman J (2006) Revisiting methodological issues in transcript analysis: negotiated coding and reliability. Internet High Educ 9(1):1–8

Gil Y, Lalouche G (2017) On the correlation between size and metric validity. Empir Softw Eng 22(5):2585–2611

Gligoric M, Eloussi L, Marinov D (2015) Practical regression test selection with dynamic file dependencies. In: Proceedings of the 2015 international symposium on software testing and analysis, ACM, pp 211–222

Goodman LA (1961) Snowball sampling. Ann Math Statist, 148–170

Gopher D, Armony L, Greenshpan Y (2000) Switching tasks and attention policies. J Exp Psychol Gen 129(3):308

Gousios G, Zaidman A, Storey MA, van Deursen A (2015) Work practices and challenges in pull-based development: the integrator's perspective. In: International conference on software engineering (ICSE), pp 358–368

Gousios G, Storey MA, Bacchelli A (2016) Work practices and challenges in pull-based development: the contributor's perspective. In: Proceedings of the 38th international conference on software engineering (ICSE). ACM, pp 285–296

Grinter RE (1995) Using a configuration management tool to coordinate software development. In: Proceedings of conference on organizational computing systems. ACM, pp 168–177

Guimarães ML, Silva AR (2012) Improving early detection of software merge conflicts. In: International conference on software engineering (ICSE), pp 342–352

Guo J, Rahimi M, Cleland-Huang J, Rasin A, Hayes JH, Vierhauser M (2016) Cold-start software analytics. In: International conference on mining software repositories (MSR), pp 142–153

Guzzi A, Bacchelli A, Riche Y, van Deursen A (2015) Supporting developers' coordination in the IDE. In: Computer supported cooperative work & social computing (CSCW), pp 518–532

Hattori LP, Lanza M (2008) On the nature of commits. In: International workshop on automated engineering of autonomous and run-time evolving systems (ARAMIS). ASE Workshops, pp 63–71

Hattori L, Lanza M (2010) Syde: a tool for collaborative software development. In: International conference on software engineering (ICSE), pp 235–238

Hindle A, German DM, Godfrey MW, Holt RC (2009) Automatic classification of large changes into maintenance categories. In: International conference on program comprehension (ICPC), pp 30–39

Hudson W (2013) Card sorting. In: The encyclopedia of human-computer interaction. Interaction Design Foundation

Hunt JJ, Tichy WF (2002) Extensible language-aware merging. In: International conference on software maintenance (ICSM), pp 511–520

Kasi BK, Sarma A (2013) Cassandra: proactive conflict minimization through optimized task scheduling. In: International conference on software engineering (ICSE), pp 732–741

Ko AJ, DeLine R, Venolia G (2007) Information needs in collocated software development teams. In: International conference on software engineering (ICSE), pp 344–353

Li C, Ding C, Shen K (2007) Quantifying the cost of context switch. In: Workshop on experimental computer science (ExpCS). FCRC Workshop, p 2

Lippe E, van Oosterom N (1992) Operation-based merging. In: Proceedings of the fifth ACM SIGSOFT symposium on software development environments (SDE). ACM, pp 78–87

McCabe TJ (1976) A complexity measure. IEEE Trans Softw Eng (TSE) 4:308–320

McKee S, Nelson N, Sarma A, Dig D (2017) Software practitioner perspectives on merge conflicts and resolutions. In: 2017 IEEE international conference on software maintenance and evolution (ICSME). IEEE, pp 467–478

Meiran N (2000) Modeling cognitive control in task-switching. Psychol Res 63(3):234–249

Mens T (2002) A state-of-the-art survey on software merging. IEEE Trans Softw Eng (TSE) 28(5):449–462

Meyer M (2014) Continuous integration and its tools. IEEE Softw 31(3):14–16

Nabi T, Sweeney KM, Lichlyter S, Piorkowski D, Scaffidi C, Burnett M, Fleming SD (2016) Putting information foraging theory to work: community-based design patterns for programming tools. In: Symposium on visual languages and human-centric computing (VL/HCC), pp 129–133

Nishimura Y, Maruyama K (2016) Supporting merge conflict resolution by using fine-grained code change history. In: International conference on software analysis, evolution, and reengineering (SANER), pp 661–664

Panichella S, Canfora G, Di Penta M, Oliveto R (2014) How the evolution of emerging collaborations relates to code changes: an empirical study. In: Proceedings of the 22nd IEEE international conference on program comprehension (ICPC). ACM, pp 177–188

Phillips S, Sillito J, Walker R (2011) Branching and merging: an investigation into current version control practices. In: International workshop on cooperative and human aspects of software engineering (CHASE), pp 9–15

Ragavan SS, Pandya B, Piorkowski D, Hill C, Kuttal SK, Sarma A, Burnett M (2017) PFIS-V: modeling foraging behavior in the presence of variants. In: Proceedings of the 2017 SIGCHI conference on human factors in computing systems (CHI). ACM, pp 6232–6244

Ritchie J, Lewis J, Nicholls CM, Ormston R et al (2013), Qualitative research practice: a guide for social science students and researchers. Sage

Robillard MP, Manggala P (2008) Reusing program investigation knowledge for code understanding. In: The 16th IEEE international conference on program comprehension (ICPC). IEEE, pp 202–211

Sarma A (2008) Palantir: enhancing configuration management systems with workspace awareness to detect and resolve emerging conflicts. PhD thesis, University of California, Irvine

Sarma A, Noroozi Z, Van Der Hoek A (2003) Palantír: raising awareness among configuration management workspaces. In: Proceedings of the 25th international conference on software engineering (ICSE). IEEE, pp 444–454

Sarma A, Bortis G, Van Der Hoek A (2007) Towards supporting awareness of indirect conflicts across software configuration management workspaces. In: Proceedings of the twenty-second IEEE/ACM international conference on automated software engineering. ACM, pp 94–103

Seaman CB (2008) Qualitative methods. In: Guide to advanced empirical software engineering. Springer, pp 35–62

Servant F, Jones JA, van der Hoek A (2010) CASI: preventing indirect conflicts through a live visualization. In: Proceedings of the 2010 ICSE workshop on cooperative and human aspects of software engineering (CHASE). ACM, pp 39–46

Sillito J, Murphy GC, De Volder K (2006) Questions programmers ask during software evolution tasks. In: International symposium on foundations of software engineering (FSE), pp 23–34

Spencer D (2009) Card sorting: designing usable categories. Rosenfeld Media

Sun X, Li B, Li Y, Chen Y (2015) What information in software historical repositories do we need to support software maintenance tasks? An approach based on topic model. Springer, pp 27–37

Symons CR (1988) Function point analysis: difficulties and improvements. IEEE Trans Softw Eng (TSE) 14(1):2–11

Tao Y, Dang Y, Xie T, Zhang D, Kim S (2012) How do software engineers understand code changes? An exploratory study in industry. In: International symposium on the foundations of software engineering (FSE), p 51

Tian J (2005) Software quality engineering, testing, quality assurance, and quantifiable improvement. Wiley

Wang S, Lo D (2014) Version history, similar report, and structure: putting them together for improved bug localization. In: The 22nd IEEE international conference on program comprehension (ICPC). ACM, pp 53–63

Weinberg GM (1992) Quality software management, vol. 1: systems thinking. Dorset House Publishing Co.

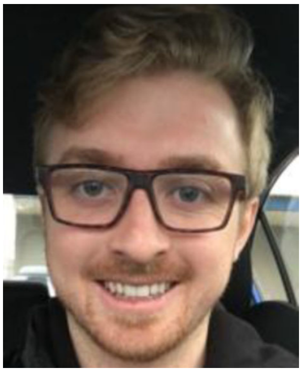
Westfechtel B (1991) Structure-oriented merging of revisions of software documents. In: Proceedings of the 3rd international workshop on software configuration management (SCM). ACM, pp 68–79

Yamauchi K, Yang J, Hotta K, Higo Y, Kusumoto S (2014) Clustering commits for understanding the intents of implementation. In: International conference on software maintenance and evolution (ICSME), pp 406–410

Yan Y, Menarini M, Griswold W (2014) Mining software contracts for software evolution. In: International conference on software maintenance and evolution (ICSME), pp 471–475

Nicholas Nelson is a Computer Science PhD student at Oregon State University. He received his BS in Computer Science from Oregon State University in 2015. His research focuses on supporting cognitive problem-solving processes in integrated development environments. Prior work has been published in peer-reviewed software engineering conferences (FSE, ICSME) and journals (EMSE), two of which have won best paper awards.

Caius Brindescu is a Computer Science PhD Candidate at Oregon State University. He received a BS in Computer Science from the "Politehnica" University of Timişoara, Romania. His research focuses on understanding the problems developers face when working collaboratively, particularly while resolving merge conflicts. Prior work has been published at top peer-review conferences (ICSE, FSE, ESEM) and journals (EMSE). He has also served as a PC member for the MSR'17 Data Mining Challenge, SCAM'15 Tool Demo track, and the VL/HCC'17 Showpiece track.



Shane McKee is a software engineer who received his MEng of Computer Science from Oregon State University in 2016. His prior work was published at ICSME 2017, which received a best paper award. Shane has worked in web development and AI/Machine Learning in the past, and he currently works as an integration engineer who now experiences all of the same difficulties that he once studied.

Anita Sarma is an Associate Professor at Oregon State University. Before this she was an Assistant Professor at University of Nebraska, Lincoln; a post-doctoral scholar at Carnegie Mellon University, and a doctoral student at University of California, Irvine. Dr. Sarma's research crosscuts Software Engineering and Human Computer Interaction and focuses on how software can be designed to support developers and end user programmers in their development efforts. Her work explores how socio-technical dependencies affect team work, how onboarding barriers in OSS can be alleviated, and how software can be made gender-inclusive. Overall, Dr. Sarma's research has resulted in 42 peer-reviewed publications (33 conferences, 9 journals), four of which have won awards. Her work has been funded through NSF and Airforce (AFOSR). She has been the recipient of the NSF CAREER award. She regularly serves on program committees in software engineering conferences (ICSE, ASE, VL/HCC, ICGSE), has been PC co-chair for ICGSE 2016, VL/HCC 2016, ICSE NIER 2019, ICSE Demo 2014, and is an Associate Editor for TSE.

Danny Dig is an associate professor of computer science in the School of EECS at Oregon State University, and an adjunct professor at University of Illinois. He enjoys doing research in Software Engineering, with a focus on interactive program transformations that improve programmer productivity and software quality. He successfully pioneered interactive program transformations by opening the field of refactoring in cutting-edge domains including mobile, concurrency and parallelism, component-based, testing, and end-user programming. He earned his Ph.D. from the University of Illinois at Urbana-Champaign where his research won the best Ph.D. dissertation award, and the First Prize at the ACM Student Research Competition Grand Finals. He did a postdoc at MIT. He (co-)authored 50+ journal and conference papers that appeared in top places in SE/PL. According to Google Scholar his publications have been cited 3000+ times. His research was recognized with 8 best paper awards at the flagship and top conferences in SE (FSE'17, ICSME'17, FSE'16, ICSE'14, ISSTA'13, ICST'13, ICSME'15), 4 award runner-ups, and 1 most influential paper award (N-10 years) at ICSME'15. He received the NSF CAREER award, the Google Faculty Research Award (twice), and the Microsoft Software Engineering Innovation Award (twice). He has started two popular workshops: Workshop on Refactoring Tools, and Hot Topics On Software Upgrades, both had at least five instances. He chaired or co-organized 14 workshops and 1 conference (MobileSoft'15), and served as a member of 35 program committees for all top conferences in his area. His research is funded by NSF, Boeing, IBM, Intel, Google, and Microsoft.

Affiliations

Nicholas Nelson[1] · **Caius Brindescu**[1] · **Shane McKee**[1] · **Anita Sarma**[1] · **Danny Dig**[1]

Caius Brindescu
brindesc@oregonstate.edu

Shane McKee
mckeesh@outlook.com

Anita Sarma
anita.sarma@oregonstate.edu

Danny Dig
daniel.dig@oregonstate.edu

[1] Oregon State University, Corvallis, OR 97331, USA