Causes of Merge Conflicts: A Case Study of ElasticSearch

Wardah Mahmood, Moses Chagama, Thorsten Berger, Regina Hebig
Chalmers | University of Gothenburg, Sweden

ABSTRACT

Software branching and merging allows collaborative development and creating software variants, commonly referred to as clone & own. While simple and cheap, a trade-off is the need to merge code and to resolve merge conflicts, which frequently occur in practice. When resolving conflicts, a key challenge for developer is to understand the changes that led to the conflict. While merge conflicts and their characteristics are reasonably well understood, that is not the case for the actual changes that cause them.

We present a case study of the changes—on the code and on the project-level (e.g., feature addition, refactoring, feature improvement)—that lead to conflicts. We analyzed the development history of ElasticSearch, a large open-source project that heavily relies on branching (forking) and merging. We inspected 40 merge conflicts in detail, sampled from 534 conflicts not resolvable by a semi-structured merge tool. On a code (structural) level, we classified the semantics of changes made. On a project-level, we categorized the decisions that motivated these changes. We contribute a categorization of code- and project-level changes and a detailed dataset of 40 conflict resolutions with a description of both levels of changes. Similar to prior studies, most of our conflicts are also small; while our categorization of code-level changes surprisingly differs from that of prior work. Refactoring, feature additions and feature enhancements are the most common causes of merge conflicts, most of which could potentially be avoided with better development tooling.

CCS CONCEPTS

• **Software and its engineering → Collaboration in software development**; *Software configuration management and version control systems*; *Software evolution*; *Maintaining software.*

KEYWORDS

software merging, case study, conflict resolution

ACM Reference Format:
Wardah Mahmood, Moses Chagama, Thorsten Berger, Regina Hebig. 2020. Causes of Merge Conflicts: A Case Study of ElasticSearch. In *Proceedings of the 14th International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS '20), February 5–7, 2020, Magdeburg, Germany*. ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3377024.3377047

VaMoS '20, February 5–7, 2020, Magdeburg, Germany

ACM ISBN 978-1-4503-7501-6/20/02…$15.00
https://doi.org/10.1145/3377024.3377047

1 INTRODUCTION

Branching and merging is a common practice in software development, often employed to create different variants of a software system, to facilitate collaborative software development [34], or to evolve systems in isolation (e.g., using feature branches). Branching and merging supports focused development while bringing together developers with diverse skill sets and allowing them to work independently on their parts of the software.

Creating software variants using branching and merging, also known as clone & own [11, 16, 20, 38], is a simple and readily available method. As opposed to more systematic methods, such as establishing a configurable software platform (a.k.a., software product line [5, 9]), it allows customizing systems via cloned variants to address different stakeholder requirements without investing into a software platform [7, 29, 30, 41]. While clone & own is the more common method [8], and can be supported by branching strategies [39, 40] and clone-management frameworks [2, 4, 18, 27, 33–36], it comes at the cost of software merging.

Merge conflicts can easily occur when two or more developers simultaneously contribute to the same piece of code—with studies reporting merge conflicts for around 16 % [10] to 43 % [28] of software merges. Often, merge tools resolve conflicts automatically [32], relying on three-way, structured, or semi-structured merge [6] techniques. The latter rely on textual merging, but exploit programming-language syntax to some extent. Still, despite increasingly intelligent merge-conflict resolution tools, developers need to resolve conflicts manually and then require expertise in conflict resolution, but they also need to understand the changes that led to the conflict and that were done to both the conflicting versions.

Various studies on merge-conflict resolution exist. They report challenges related to: understanding the changes that led to a conflict [31], practitioners' lack of knowledge on conflict resolution [23], and handling breaking changes made by other developers (on other branches) [25]. Resolving conflicts is in fact one of the major tool-related challenges in collaborative development [22]. However, while types of merge conflicts and their characteristics are reasonably well understood, that is not the case for their causes, especially not from a higher, project-level perspective, also taking domain knowledge into account. A detailed, manually curated dataset about the characteristics of changes that lead to merge conflicts could help developers understand the causes of conflicts and ways to avoid them through enhanced tooling and better coordination.

We present a case study on conflict resolution in ElasticSearch—a large open-source project excessively using branching and forking. Our focus is not on analyzing the merge conflicts, but on understanding the changes that led to merge conflicts, which we analyze from a lower-level code perspective and from a higher-level project

perspective (e.g., feature addition, refactoring, feature improvement). In other words, the former explains the semantics of changes made, and the latter the decisions that motivated these changes.

To this end, we formulated our research questions as:
RQ1: What code-level changes led to merge conflicts? We identify code changes done before the conflict and categorize conflicts based on structural changes in conflicting code.
RQ2: What project-level changes led to merge conflicts? We explain and categorize the changes from a project-level perspective from the documentation and our obtained domain knowledge.

To answer our research questions, we first identified 260 merge commits in ElasticSearch's mainline repository (master branch) and then replayed the merge commits using a semi-structured merge tool to identify 534 conflicts that cannot be resolved automatically. We then identified and manually inspected the conflicting chunks of the changesets, categorized them, and identified the changes that led to them—on the code and project-level.

In summary, we contribute:

- A dataset with 534 merge conflicts in an online appendix [1].
- Detailed descriptions of a sample of 40 conflicting changes: conflict category, mainline change description (on code-level and on project-level, the latter explained using domain knowledge), and branch change description (like for mainline).
- A categorization of conflicts and changes leading to them on the two levels.
- Findings related to the relation of both levels of changes, their characteristics, and other relevant insights related to the current state-of-the-art literature on merge conflicts.

To the best of our knowledge, we are the first to contribute a manually curated dataset of 40 merge conflicts with detailed explanations on the rationales on a domain level—analyzing one conflict took around one day. We are also the first who attempt to analyze and classify the rationale behind changes made to code that lead to merge conflicts. We hope that our results and the dataset are useful to improve conflict-resolution tools, especially tools that support developers understanding the changes that were done.

2 MOTIVATION

We briefly present a prestudy on ElasticSearch we conducted to obtain a first picture on the extent and the detailed challenges of merge conflicts, which motivated our study, together with the current literature on merge conflicts, discussed below.

2.1 Prestudy: Integrate Features from Forks

To obtain a first understanding of merge conflicts in ElasticSearch, we let five groups of seven student developers (from a project course on software evolution) identify and integrate features from ElasticSearch forks into the mainline. The goal of the prestudy was to gain initial insights on merge conflicts and form hypothesis to motivate the manual analysis. In total, five groups integrated 16 features from 14 forks. The groups' reports and interviews with two group members (in total) confirmed that resolving merge conflicts was perceived as the most challenging task during integration.

We dug deeper and inspected the steps that the groups performed for each feature to reach a successful integration. Specifically, we replayed eight merges that had resulted in merge conflicts, and we replayed the manual changes applied by the students to reach a fully functioning system. This inspection revealed that the encountered conflicts were often due to surprisingly minor changes. This indicated that: (i) conflicts might not always be necessary and (ii) that their resolution poses unnecessary distraction to the developers, even if resolving the conflicts does not require much effort. This insight was confirmed by multiple other group members we specifically asked, who perceived it as challenging to understand the implementation of the features, which they considered crucial to correctly resolve conflicts. This result is also supported by McKee et al. [31], who conduct interviews with practitioners to investigate the factors influencing the difficulty of merge conflicts. They find that the complexity and size of conflicting chunks, as well as the atomicity of changesets have a direct impact on the difficulty of merge conflicts. Overall, these prestudy results motivated us to further investigate how merge conflicts in practice really look like and what kinds of changes are causing them.

2.2 Merge Conflicts in the Literature

The literature on merge conflicts classifies into empirical studies and into techniques to avoid, reduce or resolve merge conflicts.

Characteristics of Merge Conflicts. Various studies aim at understanding the characteristics of merge conflicts.

Accioly et al. [3] study conflict characteristics to understand patterns of conflicts, their causes, and extents in real-world scenarios. Specifically, they analyze merge conflicts in 123 open-source Java projects by replaying merge scenarios using the semi-structured merge tool FSTMerge [6], leading to nine conflict patterns reported. Their study's goals partly overlap with ours, as they also provide recommendations, implications, and a classification of merge conflicts. However, as we will show, only two of our conflict categories overlap with their conflict patterns. Also, while their patterns focus on object-oriented programming, our categories are more generic. We also look into method bodies, while their patterns primarily focus class, method, and field declarations. and we also study the project-level—decisions that drive the code-level changes.

Ghiotto et al. [21] analyze merge conflicts in 2,731 open-source Java projects. They identify the conflicting chunks in the changesets of conflicting commits, and then characterize those chunks based on the number of chunks (rarely more than two), their size, and the programming language constructs involved. They identify constructs that frequently occur in the conflicting chunks, and provide ten association rules presenting the relationships between different language constructs. The idea of pairwise analysis of changes to see which of them occur together also forms the basis of the derivation of our categories. While conceptually similar, we observe no overlap between their association rules and our conflict categories (which relate the changes on the code level) we identified.

McKee et al. [31] study the practitioners' perspective on conflict resolution through interviews and a survey. Their results indicate that the complexity of the conflicting chunks, the developer's expertise in the domain of conflicting code, the number of lines in the conflicting changesets, and the atomicity of the conflicting changesets are factors that influence the resolution complexity. They also deduce that the perceived complexity of conflicts and the developer's experience in the project determine how developers

approach the resolution. Specifically, a developer manually assesses a merge conflict and then decides whether she can resolve the conflict or whether she should revert the merge and merge later, or pass the task to a colleagues more familiar with the particular change.

Merge-Conflict Management. Most works aim at reducing or avoiding merge conflicts and their negative impacts.

Guimarães and Silva [24] present the tool WeCode, which detects conflicts by continuously merging changes (committed and uncommitted) into a mainline in the background. Brun et al. [10] present the tool Crystal for conflict identification, management, and prevention. It builds on the concept of *speculative analysis*, where it diagnoses the significant classes of conflicts and possible ways to avoid them. Then it runs those potential solutions in the background, and reports the results to developers who can then decide to avoid or manage conflicts. Kasi and Sarma [28] present a tool to proactively reduce conflicts by restricting concurrent changes on shared files through task (planned change) orders—ordered sets of tasks given to each developer, avoiding parallel work on the same or dependent code parts. The tool determines tasks that would result in conflicts and restricts developers respectively to reduce conflicts.

Sarma et al. [37] focus on *workspace awareness* of changes in workspace artifacts, that is, files in a software configuration management system. Their tool Palantir, which monitors artifact changes and shares this information with developers—raising awareness about parallel modifications in the same artifact. Palantir also provides visual information about conflicting changes to help developers avoid artifact dependencies and prevent conflicts.

3 METHODOLOGY

We now briefly explain our subject system ElasticSearch as well as our methodology for extracting, analyzing, and classifying changes. The term *mainline* refers to the mainline branch, and *topic* branch to the one accommodating the changes merged into the mainline.

3.1 Subject System: ElasticSearch

ElasticSearch is a an open-source full-text search and analytics engine that allows to store, search, and analyze big volumes of data quickly and in near real-time [17]. With over 10,000 forks and 948 contributors, it is a highly popular Java-based open-source project. It supports a distributed architecture providing full-text search by indexing JSON documents. ElasticSearch is ranked to be the most popular search engine used by enterprises according to a survey by DB-Engines [13]. At the time of the study, the master branch (which targeted version 7.0.0-alpha) consisted of 5,455 Java files and 682,240 lines of code.

The development culture for ElasticSearch extensively relies on branching and forking, making it a suitable candidate for our case study. To implement new features and improve existing ones, ElasticSearch core members and contributors mostly use forks. Occasional or new contributors, who do not belong to the core team and, therefore, have no write access to the repository, require a core member to integrate their contribution by merging it into their individual forks. As a next step, the core member resolves any merge conflicts or might even address issues such as missing tests before merging into the mainline branch. During this integration, the core members and contributors mostly use `git_rebase`. Only smaller defect fixes, especially in the documentation, are typically integrated using `git_merge`. Merge conflicts can occur for both kinds of integration.

3.2 Identification of Conflicts

We cloned ElasticSearch's master branch (version 7.0.0-alpha1), which has 682 KLOC in 5,455 Java files. Using scripts, we automatically extracted all *merge commits* (those with two parents), yielding 2,965 commits. We identified merge conflicts through replaying each merge commit by recreating two branches (mainline and topic branch) as a copy of each parent commit, and merged the two branches using the command `git merge`. In total, we obtained 260 conflicting merge commits (i.e., merge commits that had conflicts). To identify their changesets' conflicting chunks, for each conflicting merge commit, we retrieved its two parents' commit hashes and the merge date. We used the merge tool Meld, which provides a *three-way* visualization of the merge (common ancestor, mainline changeset, and branch changeset), to visualize the conflicts. We observed that many arose from whitespace and comment changes, and also due to different orderings of program elements, such as methods (*ordering conflicts* [6]). To exclude conflicts that can be resolved by a structured merge tool, we relied on jFSTMerge [12]. It supports *semi-structured merge* [6], which combines advantages of unstructured and structured merges when resolving merge conflicts.

We observed that some conflicts arise at multiple parts of a file from cross-cutting changes, which implies that conflicts can be cross-cutting in nature. In those cases, instead of treating the respective chunks separately, we manually identified such changes and then also combined these into the conflicting parts, treating those conflicts as a single conflict. As a result, we collected 534 merge conflicts from the dataset of 260 merge commits with conflicts. For each identified conflict we collected: merge commit hash, merge parent commit hashes, merge date, file in the conflict, mainline branch version, and topic branch version.

3.3 Analysis of Conflicts

We manually analyzed a random sample of 40 conflicts and the changes causing them, the latter from two perspectives: (i) code-level, i.e., what code edits on both sides led to conflicts, and (ii) project-level, i.e., what type of development activities led to the code-level changes.

To categorize the merge conflicts, we first compared the conflicting changeset parts, that is, the mainline and topic branch versions, with the common ancestor version. We then identified the structural changes of the conflicting lines. Based on the observed structural changes, we derived merge conflict categories. We derived the structural changes directly from the case study, and the resultant categories of code-level changes also emerged during the analysis. Each category is a pair of changes, one representing the change in mainline, and the other in the topic branch. For example, if in the mainline version a method is renamed and in the topic branch a parameter is added in the same method, then a category would be *method rename and parameter addition*. In a second iteration, we combined similar categories and refined the categories to remove duplicates.

To obtain a complete picture on each change, we studied the complete changes, i.e., also parts that did not lead to conflicts. As a support we used `git diff` to get changes between two branches. The

challenge hereby was to identify what observed code-level changes belong to the same logical change, since the branches often contain multiple unrelated changes of which not all are related to the conflict. This part of the analysis required the most (manual) effort.

As a second source for understanding the changes, we used information from relevant commits. Since these can be all commits in the branch, we used `git_blame` to view commits that made the code-level change relevant for the studied commit. In this process, we analyzed the commit messages and pull request labels using keywords, such as refactor, improve, add, introduce, fix, remove, cleanup, problem, feature, separating, and test. If we needed deeper knowledge, we read the description in the first message of the conversation associated to the pull request. In rare cases, we read the complete conversation to understand the documented goal of the change. One of the authors cross checked the results to validate that they were consistent with the code changes before the goal of the change was finally documented. Based on all studied changes for the merge conflicts, we grouped the documented goals into project-level categories.

Overall, analyzing the changes in each conflict took a day on average, mainly since in-depth domain knowledge of the subject system was required to analyze the changes in the source code and to describe those changes. We documented all 40 conflicts in detail, as well as the identified conflict categories in our appendix [1].

4 RESULTS

We now present the identified merge conflicts, categorized into 6 categories and quantified based on the categories of code-level changes leading to them. We then explain the project-level changes, categorized into 8 categories.

4.1 Merge Conflicts and Code-Level Changes

Table 1 provides an overview on all conflicts, which are detailed in our online appendix [1]. The appendix explains all conflicts, their categories, and that of the changes in detail, as well as it provides examples. To illustrate this documentation, Fig. 1 shows that of Conflict 1. Table 2 shows a summary of the number of conflicts for each of the merge conflict category identified in the sample dataset,

In the following, we explain each conflict category.

Change of Method Call or Object Creation (MC_OC). With 25 conflicts, this category accounts for the majority of merge conflicts identified. Note that we group object creation and method calls in one category for brevity. The conflicts in this category occur due to the change in parameter type, value or amount in both branches. More precisely, we identify the six sub-categories: *Addition and/or Removal of Parameter Values* (values added in one branch with or without removal of parameters in the other branch); *Addition or Removal of Parameter Values and Change of Parameter Value Types* (values added or removed in one branch, while in the other, one or more parameter values changed due to changes of parameter types); *Addition or Removal and Modification of Parameter Values* (value(s) added or removed in one branch and changed in the other without changing the parameter type); *Change of Parameter Values* (parameter values changed in both branches without changing parameter types); *Change of Reference Variable Declaration*: (reference variable renamed in one branch, and type changed in the other, and vice versa; also, variable type or name may be changed in both branches).

A.2.1 Conflict 1

Category: Change of Method call or object creation

Mainline Change

Category: Test improvement

Project level perspective. Some aggregations tests were changed to use correct document field datatypes. The change was made to not test metrics aggregations (such as min or max) that expect a numeric value with a document field datatype such as IP (IPv4 and IPv6 addresses). The aggregations tests were modified so that not to return wrong aggregations which are also used in other tests.

Code level perspective. Listing below shows an example of a change in aggregations tests to use correct document field datatypes. In the example, a parameter that return random selected `DocValueFormat` (document field datatype) was replaced with a method `randomNumericDocValueFormat` that return random selected format from a list of datatype formats. The list of datatype formats contains Raw, Geohash, and IP datatypes.

```
    protected InternalMin createTestInstance(String name, List<PipelineAggregator
        ↪ > pipelineAggregators, Map<String, Object> metaData) {
-       return new InternalMin(name, randomDouble(),
-               randomFrom(DocValueFormat.BOOLEAN, DocValueFormat.GEOHASH,
    ↪ DocValueFormat.IP, DocValueFormat.RAW), pipelineAggregators,
-               metaData);
+       return new InternalMin(name, randomDouble(), randomNumericDocValueFormat
    ↪ (), pipelineAggregators, metaData);
    }
```

Branch Change

Category: Feature introduction

Project level perspective. Parsing from xContent, which is an abstraction on top of content such as Json, was added to some of the aggregations. The aggregations are max, min, avg, sum and value count. The change was part of high level REST client aggregations parsing feature implementation.

Code level perspective. Listing below shows a change due to addition of xContent parsing to metrics aggregations. In the listing, test parameters for document's field value and format were modified. The document's field value was improved to randomly select infinity values. On the other hand, the document's field format was updated to remove geohash format and replace with decimal format.

```
    protected InternalMin createTestInstance(String name, List<PipelineAggregator
        ↪ > pipelineAggregators, Map<String, Object> metaData) {
-       return new InternalMin(name, randomDouble(),
-               randomFrom(DocValueFormat.BOOLEAN, DocValueFormat.GEOHASH,
    ↪ DocValueFormat.IP, DocValueFormat.RAW), pipelineAggregators,
-               metaData);
+       double value = frequently() ? randomDouble() : randomFrom(new Double[] {
    ↪ Double.NEGATIVE_INFINITY, Double.POSITIVE_INFINITY });
+       DocValueFormat formatter = randomFrom(new DocValueFormat.Decimal("#####"
    ↪ ), DocValueFormat.BOOLEAN, DocValueFormat.RAW);
+       return new InternalMin(name, value, formatter, pipelineAggregators,
    ↪ metaData);
    }
```

Figure 1: Documentation of a conflict

Change of an Assert Statement Expression (AS_EXP). It refers to a change made on an assert statement as a result of other changes, such as refactoring of the class or method name, and changes on the strings which are passed in the assert expression.

Addition of Statements in the Same Area (ADD_STMT). It refers to one or more statements added in the same area of the code in both mainline and topic branch version. The statements can be added in a control-flow statement, method call or in object creation.

Modification and Removal of Statements (MOD/RMV_STMT). The statements are modified in one version and removed in the other version. The modification includes addition and removal of statements inside control statements, that is, branching statements, decision-making statements, and looping statements. It also include exception handler blocks, that is, `try`, `catch`, and `finally` blocks.

Changes in Different Statements in the Same Area (D_STMT). It refers to changes made in different, but adjacent statements.

Change of IF Statement condition (IF_C) It refers to changes made in the `if` statement condition(s) in both conflicting versions (may be due to changes in the method call used in the condition, which may also be changed due changes in the method declaration).

4.2 Project-Level Changes

We identified eight categories of project-level changes from our sample. Table 3 shows their frequencies in both the mainline and the topic branch, with the most frequent ones being Refactoring and Feature Introduction. Specifically, the categories are as follows.

Table 1: Analyzed merge conflicts in ElasticSearch and characterization of changes causing them

	category	code-level changes		project-level changes	
		mainline	branch	mainline	branch
1	MC_OC	Parameter value refactoring in an object creation	Parameter value refactoring in an object creation	Test improvement	Feature introduction
2	MC_OC, ADD_STMT	Method declaration change (changed return type and a parameter name) Variable modifier change Removal of statements	Addition of statement Addition of parameter value in an object creation	Refactoring	Feature introduction
3	ADD_STMT	IF-statement change (addition of statements and ELSE statements)	Addition of statement	Refactoring	Feature introduction
4	MC_OC	Removal of parameter value in a method call Modification of a method call (object removed in a method call)	Addition of parameter value in a method call	Framework removal	Feature introduction
5	MC_OC	Addition of parameter value in an object creation	Addition of parameter value in an object creation	Refactoring	Feature introduction
6	MC_OC	Method call refactoring (method call assigned to a variable) Modification of a parameter value in a method call	Addition of parameter value in an object creation	Refactoring	Feature introduction
7	MC_OC	Addition of parameter value in a method call	Addition of parameter value in a method call	Feature enhancement	Feature introduction
8	AS_EXP	Change of parameter value in a method call	Change of parameter value in a method call	Feature enhancement	Feature introduction
9	MOD/RMV_STMT	Removal of statements	Modification of TRY-statement	Bug fix	Feature introduction
10	ADD_STMT	Addition of statements	Addition of statements	Refactoring	Feature introduction
11	MOD/RMV_STMT	Removal of statements Removal of statements	Modification of statements (change of FOR-loop to WHILE-loop)	Refactoring	Refactoring
12	MC_OC	Removal of parameter value in an object creation	Addition of parameter value in an object creation	Refactoring	Feature introduction
13	MC_OC	Addition of statement	Method declaration change	Breaking change fix	Feature enhancement
14	MC_OC	Method call declaration change Addition of statement, Modification of statement	Modification of parameter value Removal of statement (IF-statement replaced with a statement) Addition of statement (IF-statement)	Refactoring	Feature enhancement
15	MC_OC	Change of parameter value in a method call	Change of parameter value in a method call	Feature enhancement	Test improvement
16	MC_OC	Removal of parameter value in a method call (1)	Removal of parameter values in a method call (3)	Feature enhancement	Refactoring
17	ADD_STMT	Addition of statements	Addition of statement	Feature enhancement	Feature enhancement
18	MC_OC	Modification of a parameter value in a method call (a parameter value was added in a method call passed as a parameter value)	Addition of parameter value in a method call	Framework removal	Feature introduction
19	MC_OC	Addition of parameter value in object creation Removal of parameter value in a method call	Addition of statement Addition of parameter value in a method call	Feature enhancement	Feature introduction
20	MC_OC	Removal of parameter value in a method call	Addition of parameter value in a method call	Feature enhancement	Feature introduction
21	MC_OC	Modification of parameter values	Addition of parameter value in a method call	Feature enhancement	Feature introduction
22	MC_OC	Addition of parameter values in an object creation	Modification of a parameter value in an object creation	Bug fix	Refactoring
23	MC_OC	Refactoring parameter values in a method call Addition of statement	Addition of parameter value in a method call Addition of statement	Feature enhancement	Feature introduction
24	MC_OC	Method declaration change	Method declaration change	Test improvement	Feature enhancement
25	ADD_STMT	Change of method parameter type (which led to change of parameter's value name in method calls)	Removal of statements (addition of new statements)	Bug fix	Bug fix
26	ADD_STMT	Addition of statements	Addition of statements	Refactoring	Refactoring
27	MC_OC	Modification of a parameter value in a method call Removal of a parameter value in a method call	Addition of parameter value in a method call	Refactoring	Feature introduction
28	MC_OC	Modification of a parameter value in an object creation	Modification of a parameter value in an object creation	Refactoring	Feature introduction
29	MC_OC	Modification of parameter value in a method call	Modification of parameter value in a method call	Refactoring	Feature introduction
30	ADD_STMT	Addition of statements	Addition of statements	Feature enhancement	Feature introduction
31	ADD_STMT	Addition of statement Addition of parameter values in method calls	Addition of statement	Feature enhancement	Feature introduction
32	MC_OC	Addition of parameter value in a method call	Removal of a method call Variable renaming (method call)	Feature enhancement	Feature introduction
33	MC_OC	Modification of variable type Removal of statements	Variable renaming (objection creation)	Refactoring	Library removal
34	AS_EXP	Modification of parameter value in a method call	Modification of parameter value in a method call	Refactoring	Refactoring
35	ADD_STMT	Addition of statements	Addition of statements	Refactoring	Refactoring
36	MC_OC	Modification of parameter value in a method call	Modification of parameter value in a method call	Refactoring	Refactoring
37	IF_C	Change of IF-statement condition	Change of IF-statement condition	Library removal	Refactoring
38	MC_OC	Modification of a method declaration (led to modification of a method call)	Modification of a parameter value in a method call	Refactoring	Test improvement
39	MOD/RMV_STMT	Removal of statement (a TRY-statement and a SWITCH-statement added)	RETURN-statement refactoring (in a TRY-statement)	Refactoring	Refactoring
40	D_STMT	Modification of IF-statement condition	Modification of a parameter value in a method call	Bug fix	Refactoring

Feature Introduction. This category refers to a change that introduces or is part of introducing a new feature. The change may include tests added for the feature and necessary refactoring.

Refactoring. This category refers to a change that is not part of a particular feature introduction or enhancement, but is part of the maintenance. The change includes redesigning of an API,

Table 2: Code-level merge conflicts

conflict category	conflict sub-category	# of conflicts
Change of Method Call or Object Creation (MC_OC)	Addition and/or Removal of Parameter Values	7
	Addition or Removal of Parameter Values and Change of Parameter Value Types	2
	Addition or Removal and Modification of Parameter Values	10
	Change of Parameter Values	4
	Change of Reference Variable Declaration	2
Change of an Assert Statement Expression (AS_EXP)		2
Addition of Statements in the Same Area (ADD_STMT)		9
Modification and Removal of Statements (MOD/RMV_STMT)		3
Change of IF Statement condition (IF_C)		1

extraction of a method or class, replacing control flow statements, or renaming a variable, class or method.

Feature Enhancement. This category refers to a change that is added to an existing feature. The change may improve the quality of the feature or extend its capabilities.

Test Improvement. This category refers to a change that improves or corrects one or more tests. This do not add new tests as they are part of feature introduction or feature enhancement. The change may correct or add new test infrastructure.

Bug Fix. It is a change that corrects wrong requirement, their implementation or logic, such as sequencing of statements.

Framework Removal. It is a change that removes a framework or frameworks from the project code base. The change may include the refactoring of the API after the framework has been removed.

Breaking Change Fix. This category refers to a change that corrects the changes which break the usage of the API. The breaking change may affect backward compatibility and may require actions such as upgrading the version of the software.

Library Removal. This category refers to a change that removes a library or libraries from the project code base. The change may be followed with required refactoring.

5 FINDINGS AND RECOMMENDATIONS

We discuss preliminary findings and propose recommendations for future research, for instance, for improving existing tools for merge-conflict resolution. We also believe that our findings can guide the developers to coordinate their activities in a way that minimizes the occurrence of merge conflicts.

5.1 Conflict and Change Sizes

Most merge conflicts were due to minor changes in terms of size, e.g., the addition of parameter values. 28 out of 40 conflicts only consisted of one-line changes. Only twelve of the conflicts were associated with changes involving two or more lines.

> *The conflict causes are small in ElasticSearch.*
>
> Observation 1 (Change Sizes)

This high number of small changes may be due to the fact that developers in the project under study were working on separate branches, while often incorporating changes from a mainline branch into these individual branches. "Small" merge conflicts were primarily categorized as *Addition of Statements in the Same Area* (ADD_STMT) and *Addition and Removal of Statements* (MOD-/RMV_STMT).

Table 3: Project-level changes

Change category	# of Changes	
	Mainline	Topic Branch
Feature Introduction	0	22
Refactoring	18	10
Feature Enhancement	12	4
Test Improvement	2	2
Bug Fix	4	1
Framework Removal	2	0
Breaking Change Fix	1	0
Library Removal	1	1

Note that the size of a change leading to a merge conflict does not imply that the conflicts is minor. The resolution of such "small" merge conflicts can require larger changes as well. In fact, the answer to the question whether conflict resolution is a complex problem or not is inconclusive. It is perceived to be a challenging task according to the participants of our prestudy and the results from a few other contributions (McKee et al. [31], Gousios et al. [23], Guzzi et al. [25]). We aim to report on these facts after completing our ongoing study with GitHub developers.

> *In general, the complexity of conflict resolution is still undetermined.*
>
> Observation 2 (Complexity of Conflict Resolution)

Recommendation: Impact Visualization. The state-of-the-art tooling can be enhanced to incorporate visualizations which generate views to show the complexity of merge conflicts. In the simplest form, the size of the conflicting blocks can be used as a measure. Having visualizations that quantify some aspects of merge conflicts can help to lower their psychological impact on developers. Additional developer studies can help to better understand the perceived complexity of merge conflict resolution in a diverse set of software systems and organizations.

5.2 Changes In Method Calls

25 out of 40 conflicts are a result of changes made in method invocations or object creation—the most common cause for conflicts.

> *Changes in method calls are the most common cause of conflicts in ElasticSearch.*
>
> Observation 3 (Changes in Method Calls)

This is somewhat surprising, as we would have expected to see merge conflicts in method or constructor signatures as well. However, the results are in-line with previous findings of de Menezes [14], who found that *method invocation* is the language construct most frequently involved in merge conflicts. While this needs more investigation, one possible explanation could be that developers delaying the update of method calls after having updated the signatures. This could cause other developers to fix the method call in parallel. However, changes made to the calls in branch and mainline are often not the same (e.g. change of parameters in the branch, while a parameter addition happens in the mainline), indicating that other mechanisms are at play. This could be volatile interfaces in a fast changing part of the system, indicating the need for better upfront communication between developers.

Recommendation: Refactoring Support. Having tool support that propagates changes in signatures to the respective calls can help prevent conflicts occurring due to change in method invocations. Especially the large number of conflicts due to parameter type changes could be addressed with good refactoring support in IDEs.
Recommendation: Co-Evolution tooling for API design/ usage. Another interesting direction for future work is to investigate whether tools should support development-time merging rather than commit-time merging. This can be useful in cases of APIs that are in phases of rapid change to prevent effort redundancy.

5.3 Addition of Statements

The second most common cause for conflicts is an addition of statements in the same area. Furthermore, all code-level changes leading to conflicts in our sample occurred within method bodies. This confirms a finding by Accioly et al. [3], who found that most merge conflicts (85%) are due to changes that are made in method bodies.

The addition of statements is the second most common cause of merge conflicts in ElasticSearch.
Observation 4 (Addition of Statements)

Recommendation: Look-ahead support. This observation indicates a need for supporting developers in identifying what part of the code others are working on. One strategy could be to add a built-in support in IDEs that notifies developer about current or recent edit locations of colleagues. Similarly, IDEs could be extended to facilitate communication about planned tasks among team mates'.
Recommendation: Control- or data-flow based merges. Another interesting possibility is to further investigate how often this type of conflict indicates incompatible/contradicting changes to the implementation. It is well possible that a large amount of such conflicts is caused by code pieces that is independent and just happens to be in the same place. In this case it can be interesting to build an automated alignment resolution support that uses control- and data-flow of software code to synchronize multiple implementations.

5.4 Changes on the Project-Level

Refactoring was the most frequent project-level change with 28 out of our 40 conflicts. In 15 of these 28 cases, refactoring led to changed method invocations on the code level. 11 of these conflicts featured the addition or removal of statements, or the modification of existing statements in the code. So, refactoring often collides, either refactorings on the other branch, or other project-level changes such as feature introduction or feature enhancement. This suggests that improved tool support for refactoring propagation might help, ideally integrated with version-control systems [15, 19]. The second most frequent change on the project-level were feature introductions (21 instances). 15 of these involved code-level changes in the way methods are invoked, whereas 7 of them involved changes in statements. The third most frequent cause of merge conflicts were feature enhancements, which occurred in 16 scenarios. The code-level changes corresponding to feature enhancements were mainly made in method invocations. Finally, we observed five cases of bug fixes and four cases of test improvement. Framework removal, library removal and breaking change fix constituted least amount of changes leading to merge conflicts (two, two and one respectively).

Refactoring was the most frequent change in ElasticSearch, which often collided with other refactorings or feature introductions and enhancements on the other branch.
Observation 5 (Project-Level Changes)

Recommendation: Classification support for project-level changes. We believe that keeping project-level decisions in perspective adds rationale to the code-level changes. Our finding indicates that it can be possible to build automated support for classifying code-level changes and associated project activities into project-level changes. Heuristics might be used to achieve that. Such kind of tool support can help in feature location, feature traceability and annotations. Furthermore, this tool might help to facilitate the *refactoring support* recommended above in Section 5.2. While a propagation of refactorings among forks can be supported, other activities, such as feature development or enhancements could be more easily isolated to prevent merge conflicts.

5.5 State-of-the-Art Conflict Categories

While the merge conflict categories identified by us relate to conflict patterns and classifications of other researchers, we still identified new distinctions to be relevant. This can be seen in one of the example of two merge conflict patterns found by Accioly et al. [3]. The first conflict pattern describes a situation where methods or constructors are added with the same signature and different bodies. This pattern is similar to our MC_OC category, since it involves changes inside methods. However, in our sample data set there are no merge conflicts due to changes in method or constructor declaration. The second example of a conflict pattern by Accioly et al. [3] consists of different edits to the same or consecutive lines of a method or constructor. This pattern relates to multiple of our categories: MC_OC, AS_EXP and D_STMT as they are all based on changes performed on same or adjacent statements. Similarly, de Menezes [14] identified language constructs involved in most changes that led to merge conflicts: method invocation, method declaration, method signature, variable, import, if statement, and commentary. Accordingly de Menezes [14] work with a classification of merge conflicts that bases on a combination of unique language constructs found in conflicting changes. Their *method invocation* and *variable* kind of conflicts seem to relate to our MC_OC category. The *if statement* category may relate to multiple of our categories: ADD_STMT, MOD/RMV_STMT, D_STMT and IF_C. On the other hand, our sample did not include any merge conflicts that involve import, method declaration, and method signature language constructs. Similarly, merge conflicts that involved only comments were ignored in our study.

While conflict categorizations were created in prior work, our categorization upon ElasticSearch only partly overlaps with them.
Observation 6 (Conflict Categories)

Recommendation: Holistic catalog of conflict causes. This calls for a holistic catalog of changes that can lead to merge conflicts. Such a catalog can help developers understand the properties of changes and assist in the manual resolution of those merge conflicts. Furthermore, it can foster research towards finding generic patterns for merge-conflict resolution, and ways to automate them.

5.6 Practical Implications

Besides recommendations for researchers and tool providers, we can also formulate some lessons learned for practitioners.

Coordinate Feature Introduction. Feature introduction was involved in a large number of the observed merge conflicts and is an important component of the software development life cycle. The tasks that most commonly cause merge conflicts when performed in parallel to feature introductions are refactoring and feature enhancement. This could imply that practitioners need to spend more effort in sharing and discussing work on new features before implementing them. Furthermore, practitioners working in enhancing existing features or performing refactorings should try to raise awareness about their work in the team.

Coordinate and Prioritize Refactorings. As seen in Table 1, there are 6 pairs of parallel refactoring changes. These parallel refactorings should be carefully planned since the inherent nature of refactoring involves changing the structure of code. It is therefore advised to decide on an abstract level what the developer is trying to achieve by refactoring. It might be that developers working in parallel are trying to achieve similar results by using different approaches. A better coordination could further reduce development time and effort. In addition, requirement prioritization approaches could also be incorporated to rank the importance of every refactoring with respect to the goal it aims to achieve.

Reflect Regularly on Conflict Causes. In general, conflicts between feature implementations and feature enhancements can be seen as hints that the features are too coupled in the system. Similarly, we observed a conflict caused by two feature enhancements, hinting that that either both enhancements addressed the same feature or that both features are coupled. It would be interesting to investigate in future work whether teams can benefit from regular reflection sessions about change types typically involved in their merge conflicts. This might uncover needs for architectural refactorings. Furthermore such regular reflections can help identifying potentials for improving communication. For example, one observed merge conflict was caused by two bug fix changes. It is surprising that different bugs are fixed by changing same part of the code. This might either hint on a too entangled system, or on a situation where two related bugs were not identified as such upfront.

6 THREATS TO VALIDITY

Construct Validity. A threat is the categorization of merge conflicts based on the changes that caused them, possibly categorized differently based on the author's understanding of the changes. So, we first created pairs of changes (in mainline and topic branch), then identified similar characteristics based on the code syntax.

Internal Validity. The scripts we wrote to extract conflicting merges and the jFSTMerge tool could have bugs, leading to false conflicts. So, we tested the scripts several times and manually reviewed the conflicts. Furthermore, we might have misclassified the changes leading to conflicts. To reduce this threat, we conducted a thorough analysis using more than one source to understand the changes. The commit and pull request messages together with source code were used to obtain the understanding of the changes.

Another factor affecting the internal validity is that changes in code were primarily analyzed by one author. However, another author inspected the findings and discussed the results with the primary analyst for understanding and validation.

External Validity. Our sample of 40 merge conflicts might not be representative, and our results might not generalize to the whole subject. To mitigate this threat, we randomly reviewed the conflicting changes that were not selected for the in-depth analysis to observe whether there are cases that were not represented especially for merge conflict categorization.

Our results might not generalize to systems beyond ElasticSearch, programming languages or commercial software. However, ElasticSearch is large and highly popular with 948 contributors and over 10,000 forks. Our merge-conflict categories are also not specific to Java, so we believe that the results generalize to other projects of similar scale. As future work, it can be interesting to look into ways to automatically classify projects according to their branching strategy by recovering information about process types [26].

Reliability. To enhance reliability, we described our study methodology with replication in mind, publish the dataset of conflicts and changes with detailed descriptions in our online appendix [1] together with relevant scripts.

7 CONCLUSION

We presented a case-study of merge conflicts in ElasticSearch. We contribute two datasets of conflicting changes that cannot be resolved by a semi-structured merge tool. The first dataset contains 534 conflicting changes in all extracted merge conflicts; the second dataset contains detailed descriptions of a sample of 40 conflicting changes, both from a code- as well as from a project-level perspective. As such, the datasets can be used beyond our study, for instance, to train future techniques that detect the change type and appropriately visualize the changes. In fact, we learn that most changes leading to merge conflicts are either through refactorings or through parallel feature-oriented evolution. Detecting or visualizing refactorings, or making features more explicit to foster a more isolated development, are valuable directions for future work. In addition to identifying the most frequent causes of conflicts, we found that conflicts are often confined to method bodies, which is in line with a related study [3]. However, the conflict categories we identified are different from that study [3], suggesting further work to consolidate such conflict categories. Furthermore, our study largely relied on manual work (analyzing each conflict took around one day; most effort was spent on analyzing the project-level changes). As such, a limitation is the sample size, even though, it is a random sample from a large and vibrant open-source project. We plan to extend the sample, but also conceive automated classification techniques based on our findings, as well as we plan to look into other systems.

We plan to replicate our study to a wider and diverse set of merge conflicts to validate and refine our findings. We also aim to explore better ways of isolating feature-related tasks (both development and enhancement) and to recommend edits to developers based on the higher-level goal (project-level decision). Another future direction is to improve detection and visualization of refactorings. We believe that our findings and implications can be formalized to act as a baseline for better tool support that prevents, manages, and resolves merge conflicts in collaborative software development.

REFERENCES

[1] [n.d.]. Online Appendix. https://bitbucket.org/easelab/elasticsearchstudy.
[2] Hadil Abukwaik, Andreas Burger, Berima Andam, and Thorsten Berger. 2018. Semi-automated feature traceability with embedded annotations. In *ICSME*.
[3] Paola Accioly, Paulo Borba, and Guilherme Cavalcanti. 2018. Understanding semi-structured merge conflict characteristics in open-source java projects. *Empirical Software Engineering* 23, 4 (2018), 2051–2085.
[4] Berima Andam, Andreas Burger, Thorsten Berger, and Michel Chaudron. 2017. FLOrIDA: Feature LOcatIon DAshboard for Extracting and Visualizing Feature Traces. In *VaMoS*.
[5] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines*. Springer.
[6] Sven Apel, Jörg Liebig, Benjamin Brandl, Christian Lengauer, and Christian Kästner. 2011. Semistructured Merge: Rethinking Merge in Revision Control Systems. In *ESEC/FSE*. ACM, 190–200.
[7] Wesley K. G. Assunção, Roberto E. Lopez-Herrejon, Lukas Linsbauer, Silvia R. Vergilio, and Alexander Egyed. 2017. Reengineering Legacy Applications into Software Product Lines: A Systematic Mapping. *Empirical Software Engineering* 22, 6 (2017), 2972–3016.
[8] Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M. Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej Wąsowski. 2013. A Survey of Variability Modeling in Industrial Practice. In *VaMoS*.
[9] Thorsten Berger, Jan-Philipp Steghöfer, Tewfik Ziadi, Jacques Robin, and Jabier Martinez. 2019. The State of Adoption and the Challenges of Systematic Variability Management in Industry. *Empirical Software Engineering* (2019). Preprint.
[10] Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. 2011. Proactive Detection of Collaboration Conflicts. In *ESEC/FSE*. New York, NY, USA.
[11] John Businge, Openja Moses, Sarah Nadi, Engineer Bainomugisha, and Thorsten Berger. 2018. Clone-Based Variability Management in the Android Ecosystem. In *ICSME*.
[12] Guilherme Cavalcanti, Paulo Borba, and Paola Accioly. 2017. Evaluating and Improving Semistructured Merge. OOPSLA (Oct. 2017).
[13] DB-Engines. 2019. Ranking of Search Engines. https://db-engines.com/en/ranking/search+engine.
[14] Gleiph Ghiotto Lima de Menezes. 2016. *On the nature of software merge conflicts*. Ph.D. Dissertation. Federal Fluminense University.
[15] Danny Dig, Tien N Nguyen, and Ralph Johnson. 2006. *Refactoring-aware software configuration management*. Technical Report UIUCDCS.
[16] Yael Dubinsky, Julia Rubin, Thorsten Berger, Slawomir Duszynski, Martin Becker, and Krzysztof Czarnecki. 2013. An Exploratory Study of Cloning in Industrial Software Product Lines. In *CSMR*.
[17] Elastic. 2017. Getting Started. https://www.elastic.co/guide/en/elasticsearch/reference/current/getting-started.html.
[18] Sina Entekhabi, Anton Solback, Jan-Philipp Steghöfer, and Thorsten Berger. 2019. Visualization of Feature Locations with the Tool FeatureDashboard. In *SPLC*.
[19] Tammo Freese. 2006. Refactoring-aware version control. In *ICSE*.
[20] Sergio Garcia, Daniel Strueber, Davide Brugali, Alessandro Di Fava, Philipp Schillinger, Patrizio Pelliccione, and Thorsten Berger. 2019. Variability Modeling of Service Robots: Experiences and Challenges. In *VaMoS*.
[21] Gleiph Ghiotto, Leonardo Murta, Márcio Barros, and André van der Hoek. 2018. On the Nature of Merge Conflicts: a Study of 2,731 Open Source Java Projects Hosted by GitHub. *IEEE Transactions on Software Engineering* 99, 1 (2018), 1–25.
[22] G. Gousios, M. A. Storey, and A. Bacchelli. 2016. Work Practices and Challenges in Pull-Based Development: The Contributor's Perspective. In *ICSE*.
[23] Georgios Gousios, Andy Zaidman, Margaret-Anne Storey, and Arie van Deursen. 2015. Work Practices and Challenges in Pull-based Development: The Integrator's Perspective. In *ICSE*.
[24] M. L. Guimarães and A. R. Silva. 2012. Improving early detection of software merge conflicts. In *ICSE*.
[25] Anja Guzzi, Alberto Bacchelli, Yann Riche, and Arie van Deursen. 2015. Supporting Developers' Coordination in the IDE *(CSCW '15)*.
[26] Abram Hindle. 2010. Software process recovery: recovering process from artifacts. In *WCRE*.
[27] Wenbin Ji, Thorsten Berger, Michal Antkiewicz, and Krzysztof Czarnecki. 2015. Maintaining Feature Traceability with Embedded Annotations. In *SPLC*.
[28] Bakhtiar Khan Kasi and Anita Sarma. 2013. Cassandra: Proactive Conflict Minimization Through Optimized Task Scheduling. In *ICSE*.
[29] Jacob Krueger and Thorsten Berger. 2020. Activities and Costs of Re-Engineering Cloned Variants Into an Integrated Platform. In *VaMoS*.
[30] Max Lillack, Stefan Stanciulescu, Wilhelm Hedman, Thorsten Berger, and Andrzej Wasowski. 2019. Intention-Based Integration of Software Variants. In *ICSE*.
[31] S. McKee, N. Nelson, A. Sarma, and D. Dig. 2017. Software Practitioner Perspectives on Merge Conflicts and Resolutions. In *ICSME*.
[32] Tom Mens. 2002. A state-of-the-art survey on software merging. *IEEE Transactions on Software Engineering* 28, 5 (2002), 449–462.
[33] Leticia Montalvillo and Oscar Díaz. 2015. Tuning GitHub for SPL development: branching models & repository operations for product engineers. In *SPLC*.
[34] Leticia Montalvillo, Oscar Díaz, and Thomas Fogdal. 2018. Reducing coordination overhead in SPLs: peering in on peers. In *SPLC*.
[35] Julia Rubin and Marsha Chechik. 2013. A Framework for Managing Cloned Product Variants. In *ICSE*.
[36] Julia Rubin, Krzysztof Czarnecki, and Marsha Chechik. 2015. Cloned product variants: from ad-hoc to managed software product lines. *Software Tools for Technology Transfer* 17, 5 (2015), 627–646.
[37] A. Sarma, D. F. Redmiles, and A. van der Hoek. 2012. Palantır: Early Detection of Development Conflicts Arising from Parallel Code Changes. *IEEE Transactions on Software Engineering* 38, 4 (July 2012), 889–908.
[38] Ştefan Stănciulescu, Sandro Schulze, and Andrzej Wąsowski. 2015. Forked and Integrated Variants in an Open-Source Firmware Project. In *ICSME*.
[39] Ştefan Stănciulescu, Sandro Schulze, and Andrzej Wąsowski. 2015. Forked and integrated variants in an open-source firmware project. In *ICSME*.
[40] Mark Staples and Derrick Hill. 2004. Experiences Adopting Software Product Line Development Without a Product Line Architecture *(APSEC)*.
[41] Daniel Strueber, Mukelabai Mukelabai, Jacob Krueger, Stefan Fischer, Lukas Linsbauer, Jabier Martinez, and Thorsten Berger. 2019. Facing the Truth: Benchmarking the Techniques for the Evolution of Variant-Rich Systems. In *SPLC*.