Empirical Software Engineering
An Empirical Study of Token-based Micro Commits
--Manuscript Draft--

Manuscript Number:	EMSE-D-23-00166R1
Full Title:	An Empirical Study of Token-based Micro Commits
Article Type:	Research Papers
Keywords:	Empirical Study; Micro Commits; Tokens; Mining Software Repositories
Corresponding Author:	Masanari Kondo Kyushu University - Ito Campus: Kyushu Daigaku Fukuokashi Nishiku, JAPAN
Corresponding Author Secondary Information:	
Corresponding Author's Institution:	Kyushu University - Ito Campus: Kyushu Daigaku
Corresponding Author's Secondary Institution:	
First Author:	Masanari Kondo
First Author Secondary Information:	
Order of Authors:	Masanari Kondo
	Daniel M. German
	Yasutaka Kamei
	Naoyasu Ubayashi
	Osamu Mizuno
Order of Authors Secondary Information:	
Funding Information:	Japan Society for the Promotion of Science (JP22K17874) — Dr Masanari Kondo
	Japan Society for the Promotion of Science (JP21H04877) — Dr Yasutaka Kamei
	Japan Society for the Promotion of Science (JP22K18630) — Dr Yasutaka Kamei
	Natural Sciences and Engineering Research Council of Canada — Dr Daniel M. German
	Inamori Research Institute for Science, Kyoto, Japan — Dr Yasutaka Kamei
Abstract:	In software development, developers frequently apply maintenance activities to the source code that change a few lines by a single commit. A good understanding of the characteristics of such small changes can support quality assurance approaches (e.g., automated program repair), as it is likely that small changes are addressing deficiencies in other changes; thus, understanding the reasons for creating small changes can help understand the types of errors introduced. Eventually, these reasons and the types of errors can be used to enhance quality assurance approaches for improving code quality. While prior studies used code churns to characterize and investigate the small changes, such a definition has a critical limitation. Specifically, it loses the information of changed tokens in a line. For example, this definition fails to distinguish the following two one-line changes: (1) changing a string literal to fix a displayed message and (2) changing a function call and adding a new parameter. These are definitely maintenance activities, but we deduce that researchers and practitioners are interested in supporting the latter change. To address this limitation, in this paper, we define micro commits, a type of small change based on changed tokens. Our goal is to quantify small changes using changed tokens. Changed tokens allow us

	to identify small changes more precisely. In fact, this token-level definition can distinguish the above example. We investigate defined micro commits in four OSS projects and understand their characteristics as the first empirical study on token-based micro commits. We find that micro commits mainly replace a single name or literal token, and micro commits are more likely used to fix bugs. Additionally, we propose the use of token-based information to support software engineering approaches in which very small changes significantly affect their effectiveness.

Responses to Reviewers

EMSE-D-23-00166
submitted to
Empirical Software Engineering

M.Kondo, D.German, Y.Kamei, N.Ubayashi, and O.Mizuno

March 8, 2024

Dear Editor and Reviewers,

We would like to thank the editor and the three reviewers (i.e., R1, R2, R3) for all your comments, which further improved our paper.

Response to the Editor

E.1 Many thanks for your submission to ESE Journal. Based on reviewer feedback, I am recommending a major revision with a focus on the following points:

1) Clarify the definition of the most essential construct, micro commits, and ensure its consistent use throughout the analyses and reporting (R1 and R2) 2) Clarify methodological details, particularly about the coding process (R1, R2, R3, confidence interval calculations (R2), and the effect of other factors that may have impacted the results (R3) 3) Reconsider the presentation of results to improve readability and comprehension (R2) 4) Clarify the novelty by contrasting the contribution with the gap in existing work (R3) 5) Discuss mitigation strategies you have used, or recommend for others to use, in addition to listing the validity threats (R3)

We would like to thank the editor and three reviewers (i.e., R1, R2, and R3) for your insightful and useful suggestions. Below we summarize our changes to the paper and the links to specific comments:

1. **Micro commit definition.** The terminology related to micro commits in the paper has been updated. We decided, for the sake of clarity, to define micro commit as a commit that adds at most five tokens and removes at most five tokens of source code. We have revised and reorganized our research questions accordingly. This addresses issues R1.1, R2.1, R2.2, R2.3, R2.5, R2.6, R3.1
2. **Coding process.** We have updated the description to clarify the coding process. This addresses issues R2.8, R2.9, R3.4.
3. **Clarification of the presentation of the results.** We have revised the result presentation. For instance, we improved the contrast in Figure 3. Related to R2.7.
4. **Clarification of the novelty.** We have added “Section 9.3 Knowledge Gap in Previous Studies” and discussed the novelty of this study. R3.3.

Listing 1: An example micro commit in Linux
retrieved from: 092734b4bb227faddf241b116af14357645d963c

```
@@ −385 +385 @@ EXPORT_SYMBOL(bt878_device_control);
−struct cards card_list[] __devinitdata = {
+static struct cards card_list[] __devinitdata = {
```

5. **Mitigation strategies for the threats.** We have included mitigation strategies for each threat. This addresses issues R3.6.

6. **Other issues.** Related to R1.2, R1.3, R1.4, R2.4, R2.10, R2.11, R3.2, R3.5.

Response to Reviewer R1

Summary In this paper, the authors conduct an empirical analysis of micro commits, defined as commits that modify a maximum of five tokens. The authors justify this definition by analyzing one-line commits and discovering that almost 90% of the studied systems' commits add or remove no more than five tokens. The authors then explore the characteristics of micro commits and the types of changes made. They conducted this study on four large systems with development histories spanning several years.
Strengths
- Interesting topic approached from a perspective that remained out of the focus of similar studies
- Well-written, easily readable paper
- Interesting findings and discussion of results
- Replication package with Python scripts and dataset

Weaknesses
- A small sample of four systems
- The maximum criterion of 5 tokens was determined based on one-line commits
- It seems that the manual classification could have been automatized to study the entire dataset of micro commits

Summary: We thank Reviewer 1 for your constructive comments. Based on your comments, we've updated the rationale behind the definition of micro commits. Additionally, we have shared the results of automatically classifying micro commits instead of manual classification. Finally, we have further discussed the threats to validity due to our selection of only 400 micro commits from all projects. The details are described below.

R1.1 However, I see two major issues:
- There appears to be a flaw in establishing the maximum 5 tokens criterion. The paper argues that relying on lines or churns may not accurately capture the extent of a change and therefore suggests using tokens instead for micro commits. However, the paper then sets the upper limit for tokens in micro commits at 5 based on the observation that 90% of one-line commits modify no more than 5 tokens. A better approach would have been to analyze the distribution of the number of tokens in all commits to determine the maximum. Moreover, 5 tokens are unlikely to span multiple lines, whereas 10 tokens may represent a small change that spans multiple lines. The paper acknowledges this limitation in the Threats to Validity section. So I am pointing this out as a weakness of the submission. I do not believe that additional analysis is necessary.

Response: Thank you very much for raising the concern about the definition of micro commits. Similar concerns were raised by two other reviewers.

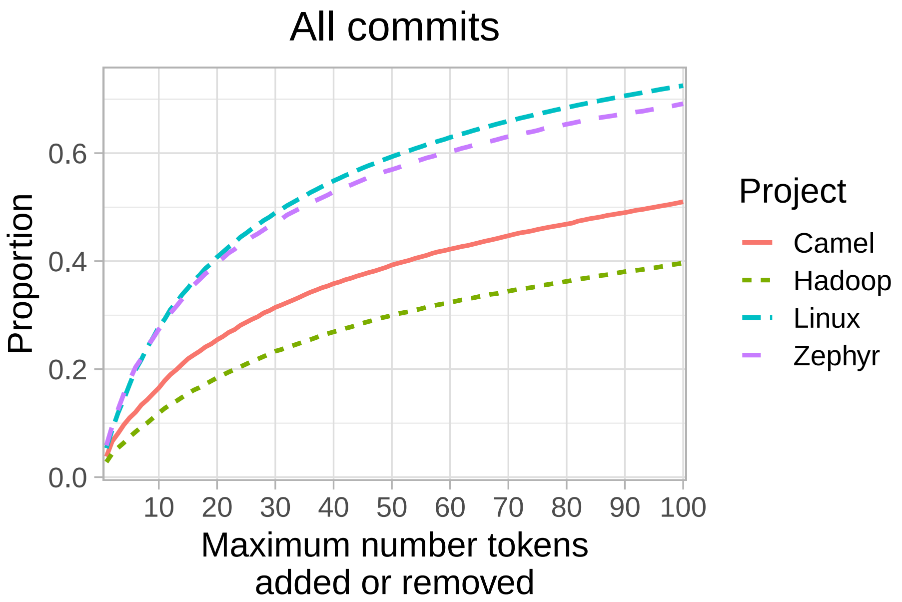


Figure 1: Accumulated distribution of all commits in terms of the maximum number of added or deleted tokens (up to 100 tokens)

Listing 2: An example micro commit with multiple changed lines in Linux retrieved from: 0ce6e62bd6591777bd92873e2db93fdbc5228122

```
@@ -1143,2 +1143,3 @@ int path_lookup_open(const char *name, unsigned int lookup_flags,
-int path_lookup_create(const char *name, unsigned int lookup_flags,
- struct nameidata *nd, int open_flags, int create_mode)
+static int path_lookup_create(const char *name, unsigned int lookup_flags,
+ struct nameidata *nd, int open_flags,
+ int create_mode)
```

As suggested, we looked at the distribution of commits by the number of tokens added and removed (Figure 1). We noted that this distribution (for a small number of tokens) grows in a relatively stable manner.

Therefore we proceeded to define micro commits with a threshold of 5 tokens (as in the original paper). This is similar to other papers that have arbitrarily defined small commits (e.g., Purushothaman et al. [5] defined *small commits* as those modifying less than 10 lines, and Alali et al. [1] defined *extra-small commits* as those adding at most 5 lines of code).

We support our decision as follows: (1) In the languages being studied (C and Java), it is highly unlikely to add a new statement with only five tokens, suggesting that such commits carry out minor modifications. For example, within five tokens, developers can only add a function call with one parameter and an ending semicolon: `name(parm);` includes two identifiers, two parentheses, and one semicolon. (2) In the systems we studied, between 7.45 and 17.95% of all studied commits add at most 5 tokens and remove at most 5 tokens.

Also, we have relocated the original RQ1 to RQ3 in the revised paper. RQ3 now compares micro commits with one-line commits to clarify their differences. Lastly, we have expanded the discussions in the threats to validity section. The important description from the paper (Abstract, Section 1, and Section 3.2) is quoted below.

Abstract

...
To address this limitation, in this paper, we define *micro commits*, a type of small change based on changed tokens. Our goal is to quantify small changes using changed tokens. Changed tokens allow us to identify small changes more precisely. In fact, this token-level definition can distinguish the above example.
...

Section 1 Introduction

...
While prior studies [2, 18, 37] use churn (number of lines added and removed) to identify small changes (e. g., small commits and One-line commits [37], or extra-small commits [2]), it has one significant limitation: they consider the line to be the finest-grained entity of changed source code. More specifically, such a definition overlooks the details of what has changed in a line [11, 34, 38]. For instance, when several lines have a small change (such as an identifier being renamed in a few places), these modifications might appear as one line added and one line removed for each change, rather than a single identifier change.

Another problem is that splitting or joining a line of code that is being modified can result in noise. For example, splitting a line into two would be reflected as a change to multiple lines in version control systems (e. g., Git), and this type of change can add noise to the analysis of the history of the development process.

These limitations sometimes cause researchers to fail in accurately quantifying small changes. For example, Listing 2 shows a commit in the Linux repository that changes a few lines (i. e., three added and two deleted lines). While this commit corresponds with a multiple-line change and may not correspond to a One-line commit, it only adds a token "static". This is similar to Listing 1 corresponding with a One-line commit that only adds a token "static". Studying the actual changed tokens instead of the lines can provide a better understanding of the characteristics of the small changes.

In this paper, we define a new class of commits: *micro commits*. **Micro commits are commits that add at most five tokens and remove at most five tokens of source code**. We aim to quantify small changes using the token-level definition (i. e., micro commits) rather than relying on the line-level definition (i. e., One-line commits). This token-level definition allows us to identify small changes more accurately, and use token information to characterize them. We conducted an empirical study on four large, mature open-source projects to: a) demonstrate that micro commits are common, accounting for between 7.45 and 17.95% of all studied commits in the studied projects, b) understand their qualitative and quantitative characteristics, and c) show our definition of micro commits (a threshold of 5 added and removed tokens) includes approximately 90% of all One-line commits, yet only approximately 40–50% of micro commits are One-line commits.
...

Section 3.2 One-line Commits and Micro Commits

...
We extracted micro commits based on the hunks provided by Git repositories processed by `cregit`. Micro commits refer to commits that include a maximum of five added tokens and five deleted tokens across all hunks. This number was chosen for the following reasons.

- In the languages being studied (C and Java), it is highly unlikely to add a new statement with only five tokens, suggesting that such commits carry out minor modifications. For example, within five tokens, developers can only add a function call with one parameter and an ending semicolon: `name(parm);` includes two identifiers, two parentheses, and one semicolon.
- In the systems we studied, between 7.45 and 17.95% of all studied commits add at most 5 tokens and remove at most 5 tokens.

...

R1.2 - It is unclear why classifying commits into low-level maintenance activities required manual annotation. This classification is based on the operation (add/delete/replace) and the entity type (identifier, constant, declaration, statement, control flow, other/no), which could be automatically detected. SrcML could assist in identifying the added or removed entity based on its type. Additionally, tools are available to perform AST diff off commits (e.g., https://github.com/GumTreeDiff/gumtree). Therefore, there is no need to limit the analysis to the small subset of 383 sample commits when an automatic approach could classify the entire dataset. There are two possible solutions: either explain why the manual analysis cannot be achieved automatically or implement a simple automatic approach and run it on the full dataset of micro commits. In the latter scenario, the manually classified sample set could serve as an oracle to evaluate the accuracy of the automatic approach. This needs to be addressed in the paper.

Response: Thank you for your valuable feedback. We agree with your suggestion that having an approach to automatically classify targets and operations would be beneficial for future studies. We implemented a rule-based approach to validate the automatic classification. Unfortunately, achieving high performance can be challenging. Therefore, we retain the manual annotation in the revised paper. We discuss this limitation in the Threats to Validity section. For future studies, we have included this rule-based approach in the replication package. The outcome of our rule-based approach is outlined below. However, you may choose to skip this section as we have not included these results in the revised paper.

The outcome of a rule-based approach: While we initially believed that manual annotation was necessary because of the complexity of classification, we attempted to implement an automatic classification approach and assess its performance. Specifically, we implemented a rule-based approach to automatically classify micro commits into the appropriate targets. For example, if the micro commit includes the token "if", this approach classifies the commit as belonging to the control flow. This approach also employs rules to classify the targets. For instance, if a commit consists solely of literal tokens, it will be classified as a constant.

Table 1 shows the accuracy of classification for micro commit targets using our automatic classification approach. Overall, the approach achieves an accuracy of approximately 81.2% for categorizing micro commits into their targets. In contrast, we observe that although the approach achieves 100% accuracy in certain targets, it performs worse in others, such as multi, declaration, and control flow. These misclassifications occurred due to the classification complexity.

For example, classifying micro commits into the multi targets is complex. Listing 3 shows a micro commit that we manually classified as belonging to the multi(control flow, expression). However, the automatic approach classified it solely as belonging to the control flow. In this code, the "ifdef" statement is changed to "if" and the "IS_ENABLED" function call is added. The change from "ifdef" to "if" is a control flow change, while the addition of the "IS_ENABLED" function call is an expression change. These changes are classified as multi(control flow, expression) according to the manual classification.

Listing 4 displays the tokenized commit of Listing 3 using cregit. As mentioned earlier, the automatic classification approach categorizes commits based on the changed tokens and their token types. Hence, this approach uses the tokens and token types present in Listing 4.

The tokens and token types related to the expression in this commit are "IS_ENABLED" (name) and "(" (parentheses). However, these tokens are also commonly used in the control flow. For example, when adding a new "if" statement, developers will write the keyword "if", followed by the name token and the parentheses to indicate the condition. Additionally, this commit contains tokens strongly related to the control flow, such as "if". Consequently, our automatic classification approach categorizes this as the control flow. This is a common challenge in other multi targets as well.

Not only in multi targets, but there are also cases where problems can occur with rule-based approaches. To illustrate, Listing 5 presents an example commit of misclassification by our automatic classification approach. In this commit, the definition of "*addr" is changed, which would be classified as a declaration change in our manual classification. However, our automatic classification approach classifies this change as an identifier change. This is because our automated classification approach uses a rule: it considers the change as an identifier if its target is a token of the name type token. Since this code only modifies name type tokens (Listing 6, which is a tokenized version by cregit), it is classified as an identifier according to this rule. However, based on our manual analysis, it is clear that this is a declaration change. This is an example where not only the token type but also the semantic elements need to be classified into the appropriate target. Therefore, relying solely on rule-based automatic classification approaches may not be the best alternative to manual annotation. Note that we did not implement the automatic approach to classify micro commits into operation categories because it does not achieve perfect performance in target classification tasks.

Target	declaration	constant	identifier	control flow	statement	expression	no	multi	Total
Accuracy	65.4(34/52)	100.0(56/56)	100.0(69/69)	73.1(19/26)	100.0(40/40)	87.9(87/99)	66.7(2/3)	10.5(4/38)	81.2(311/383)

Table 1: Accuracy of our automatic classification approach for targets.

Listing 3: An original Linux commitretrieved from: 907aa265fde6589b8059dc51649c6d1f49ade2f3

```
@@ −305 +305 @@ EXPORT_SYMBOL(of_drm_find_panel);
−#ifdef CONFIG_BACKLIGHT_CLASS_DEVICE
+#if IS_ENABLED(CONFIG_BACKLIGHT_CLASS_DEVICE)
```

Listing 4: A token-level Linux commit (org: 907aa265fde6589b8059dc51649c6d1f49ade2f3)

```
@@ −775,3 +775,5 @@ end_endif
−begin_ifdef
−ifdef|#
−directive|ifdef
+begin_if
+if|#
+directive|if
+name|IS_ENABLED
+argument_list|(
@@ −779 +781,2 @@ name|CONFIG_BACKLIGHT_CLASS_DEVICE
−end_ifdef
+argument_list|)
+end_if
```

Listing 5: An original Linux commit retrieved from: 6b80778d3d7424b8f0a45052742d065ec491abd8

```
@@ −2039 +2039 @@ static int smc911x_drv_probe(struct platform_device *pdev)
− unsigned int *addr;
+ void __iomem *addr;
```

Listing 6: A token-level Linux commit (org: 6b80778d3d7424b8f0a45052742d065ec491abd8)

```
@@ −9289,2 +9289,2 @@ decl_stmt|;
−name|unsigned
−name|int
+name|void
+name|__iomem
```

Section 8.3 Internal Validity

...
An alternative solution is to use an automatic classification approach rather than manual analysis. We developed a heuristic-based method to classify micro commits into their corresponding targets automatically. However, this method does not yield perfect results. To facilitate replication of this approach, we have included it in our replication package.
...

R1.3 Apart from these issues, I do not see additional major weaknesses/problems with the manuscript. This is a nice submission which could be an interesting article in the journal. However, I noticed some additional parts of the text that require clarifications. Therefore, I suggest accepting the paper after a revision has been made to address the manual annotation issue and fix any minor issues.

Response: Thank you very much for your valuable feedback. Below, we describe our updates.

R1.4 In the rest of my review, I list the minor issues that need clarification and suggested improvements.

- P3, "First, we empirically define micro commits as those that add at most five tokens and/or remove at most five tokens (replacing a token would imply one removal and one addition)." => The "and/or" makes this definition ambiguous and is not clarified later in the paper. This should be fixed.

Response: Thank you very much for pointing out our writing issue. By incorporating feedback from another reviewer's comment, we have revised not only this statement but also other statements in order to improve the clarity of our definition for micro commits. Our micro commit definition should use "and" in this statement.

- P4, Table 1: It seems that the number of commits listed in the "studied commits" column for these projects may not be accurate. For instance, the text mentions that Linux has over one million commits, but Table 1 only shows 748,618. It is possible that only a portion of the development history was analyzed. If that is the case, include the time ranges examined in the table.

Response: Thank you for your comment. This column "#studied commits" represents the studied commits that were extracted from all commits. The procedure is the same as described in Section 3.3. Therefore, we remove certain commits, such as those that do not modify any source code. We clarify this in the text. Additionally, to make it clear that this column does not represent the total number of commits, we have added a new column "#total commits" that represents the total number of commits.

Section 2 Motivating Example

...
As in [37], we use the diffs generated by Git to identify one-line commits. Table 1 shows the proportion of one-line commits. The proportion was computed by using the "#studied commits" column. It only shows the commits that have made changes to the source code. Our analysis is conducted based on these commits. The detailed procedure for extracting commits is explained in Section 3.2 and Section 3.3. We observe 4.28–8.20% of one-line commits.
...

- The text says, "Git has four algorithms to compute diffs." => This is clear for the example, but then which algorithm was used with cregit to construct the dataset?

  Response: Thank you for bringing attention to this missing information. We use the *myers* algorithm, which is the default algorithm in Git. We have now included this information in the paper.

  > **Section 3.3 Data Collection**
  >
  > ...
  > We record the same for token-based diffs (replacing lines with tokens–including their types). Note that when obtaining diffs with Git, we use the *myers* algorithm, which is the default algorithm. Also, we record commit messages.
  > ...

- I do not see a reason to exclude comment changes. Comments are integral parts of the source code, and changes made to them are important for maintenance purposes. Why are they excluded from the analysis?

  Response: Thank you. We agree with the reviewer's thoughts. Making changes to comments is important for maintenance purposes in practice. The primary reason for excluding comments is to prioritize maintenance activities for code logic. As mentioned in Section 1, our intention is to support various software engineering approaches (e.g., defect prediction), which typically prioritize code logic over comments. Indeed, defect prediction studies typically do not take into account comment issues when identifying target defects. While we acknowledge the importance of changes made to comments for maintenance purposes, this perspective is beyond the scope of our paper. We have added this reason. Furthermore, we have discussed this perspective in future work. This is because maintenance activities for comments are also of interest and should be considered when defining other types of micro commits.

  > **Section 3.2 One-line Commits and micro commits**
  >
  > ...
  > Source code comments are important for source code and making changes to comments are also maintenance activities. However, in this paper, we exclude comments and execute our analysis. The reason is to prioritize maintenance activities for code logic. As mentioned in Section 1, our intention is to support various software engineering approaches (e.g., defect prediction), which typically prioritize code logic over comments. Indeed, defect prediction studies typically do not take into account comment issues when identifying target defects [19, 24, 33]. While we acknowledge the importance of changes made to comments for maintenance purposes, this perspective is beyond the scope of our paper.
  > ...

  > **Section 8.2 Construct Validity**
  >
  > ...
  > Hence, we believe our definition can be acceptable. However, our definition may not be the best; thus, future studies are necessary to find a better definition than our first one. For example, future studies can investigate different thresholds for the number of tokens. Also, studying different thresholds for added and deleted tokens (e.g., 3 added tokens and 5 deleted tokens) can be beneficial. Additionally, they can consider changes to source code comments. This definition would encompass not only maintenance activities

related to code logic, but also various other maintenance activities.
...

- According to Footnote 7, it appears that header files were not considered for C projects, despite their importance as integral parts of the C code. Can you confirm if the ".h" files were indeed ignored and provide an explanation as to why?

Response: Thank you for raising this concern. We concur with the comments and have added the header file as the target files. Consequently, our analysis results have slightly changed. However, the findings and implications generally remain unchanged.

- I suggest specifying the Java (and C) versions of the projects supported by the analysis. Java has undergone significant changes in recent major versions, which can impact the identifiable tokens and results of the analysis.

Response: Thank you for raising this concern. Our analysis supports up to C11 for C, and Java SE8 Edition for Java. The supported versions of Java and C for our analysis depend on cregit (and srcML) used to tokenize the source code files. This poses a threat to the validity of our analysis. Therefore, we have added this point to the paper.

Section 8.3 Internal Validity

...
The tool "cregit" used to tokenize the source code files utilizes srcML. Therefore, our analysis can only be applied to specific versions of Java (Java SE8 Edition) and C (up to C11) that are supported by srcML. We can find the supported versions on the official homepage[a]. To extend our analysis to different versions of Java and C, it is necessary to update srcML and apply our analysis to those versions.

[a] https://www.srcml.org/#home

- Table 3 could benefit from a more precise description. What do "n" and "Pro" mean?

Response: Thank you for pointing out this writing issue. We have added the description.

Section 5.2 Results

...
Each row indicates a set of token types modified by a single micro commit and their appearances (i.e., # of micro commits). The "n" column indicates the number of appearances, while "Pro" indicates the proportion. In this paper, we use the same column name in the other tables.
...

- P15, "If at least one of the keywords is included in the commit messages, we classify the commit into that category. Otherwise, we do not label commits. We used the keyword list defined by Levin et al. [22]" => Provide more details about the keyword matching. In particular, were common techniques such as stemming and case folding applied? The replication package shows that the Python NLTK library was used in the scripts. Please clarify this.

Response: Thank you for pointing out the unclear description. We have added steps to classify

Criteria	First Time	Second Time	Third Time
Operations	0.686	0.669	0.832
Targets	0.425	0.671	0.754

Table 5: Fleiss' Kappa scores for each repetition

Listing 6: Example "replace identifier" commit diff retrieved from `f72e6c3e17be568138d8e4855ac2734d251a6913` in Linux.

```
- strlcpy(drvinfo->bus_info, pci_name(mdev->pdev),
+ strlcpy(drvinfo->bus_info, dev_name(mdev->device),
```

commits into categories. Also, in response to the concerns raised by Reviewer 2, we have moved this analysis to Section 7.3.

Section 7.3 Program Repair

...

The detailed procedure is as follows.

Step 1: Apply preprocessing to the commit messages using the NLTK package[a] in Python by following the steps below:

- Tokenize the text and convert all words to lowercase.
- Remove stopwords and punctuation.
- Perform stemming on all words.

Step 2: Check if the stemmed commit message contains a keyword for each category.

Step 3: Classify each commit into categories based on keywords present in the commit message.

...

[a] `https://www.nltk.org/`

- I suggest moving the manual coding details from the appendix to the main text, as it is a crucial element of the study method. It might be more appropriate to place Table 5 in the appendix instead.

Response: Thank you for this suggestion. We have merged and updated the description in Section 6.1. In response to the suggestions made by Reviewer 2, we have decided to retain Table 5 in Section 6.1. Additionally, in response to the two comments below and comments from Reviewer 2, we have included details of our manual coding in Section 6.1. Due to the reorganization of the RQs, Section 6.1 has been moved to Section 5.1 in the revised version.

Section 5.1 Approach

...

Our manual inspection consisted of two phases: (1) *constructing a coding guide* and (2) *manual classification*. Constructing a coding guide for manual classification/annotation

Criteria	Candidate	Description	Example Commits and Their Diffs in Linux
Operations	add	Add a new entity	122503683169b21d9cdb90380a20caad7ba994b6 Diff: Listing 12
	replace	Replace an existing entity	b7a90e8043e7ab1922126e1c1c5c004b470f9e2a Diff: Listing 13
	remove	Remove a completely existing entity	b95b4e1ed92a203f4bdfc55f53d6e9c2773e3b6d Diff: Listing 14
	multi	Operations on multiple targets	e3ae0cac00042d7fb76914c30c5f991f918e65b4 Diff: Listing 9
	no	Non-functional modification	a092532483e3200a53c8b1170b3988cc668c07ef Diff: Listing 15
Targets	declaration	Change in a type signature	36f062042b0fd9f8e41b97a472f52139886ca26f Diff: Listing 16
	constant	A constant (e. g., literal)	1db76c14d215c8b26024dd532de3dcaf66ea30f7 Diff: Listing 17
	identifier	An identifier (e. g., function calls)	70e8b40176c75d3544024e7c934720b11a8a11bf Diff: Listing 18
	control flow	Modifies the control flow	415a1975923722f729211a9efca550c60c519bf3 Diff: Listing 19
	statement	A the majority of a statement (delimited by semicolon)	b95b4e1ed92a203f4bdfc55f53d6e9c2773e3b6d Diff: Listing 14
	expression	A part of a statement and not classified into other categories	40cc394be1aa18848b8757e03bd8ed23281f572e Diff: Listing 20
	multi	Multiple targets are altered	e3ae0cac00042d7fb76914c30c5f991f918e65b4 Diff: Listing 9
	no	Non-functional modification	a092532483e3200a53c8b1170b3988cc668c07ef Diff: Listing 15

Table 6: The description of each candidate in our manual inspection

is a common practice in the field of mining software repositories [6, 13, 14, 42, 43, 46]. To create the coding guide, we referred to previous studies [6, 13, 14, 42, 43, 46] and followed the process detailed below.

The initial coding guide was first discussed by the first and second authors. Since this is the first study to classify micro commits, we examined both micro commits and other types of commits to develop the initial coding guide. After constructing the initial coding guide, we aimed to reach a consensus among the first three authors for this guide and refine the guide. Specifically, we independently annotated 20 micro commits from a subset of all micro commits. This subset consists of micro commits that only change less than or equal to five tokens in main files (.c or .java files) in the Linux, Hadoop, and Zephyr projects to investigate a single operation commit for refining the coding guide. We computed the agreement rate for these 20 micro commits using Fleiss' Kappa [9] that is used to demonstrate inter-rater agreement when there are more than two raters. It is also frequently applied in the field of mining software repositories [6,13]. The Kappa coefficient is commonly interpreted using the following

Listing 7: Example "replace expression" commit diff retrieved from 8b58f261113c442717b9d205ab187e51c3823597 in Linux.

```
- dqm->total_queue_count++;
+ dqm->total_queue_count--;
```

scale [41]: Slight agreement ($0.01 \leq k \leq 0.20$), Fair agreement ($0.21 \leq k \leq 0.40$), Moderate agreement ($0.41 \leq k \leq 0.60$), Substantial agreement ($0.61 \leq k \leq 0.80$), Almost perfect agreement ($0.81 \leq k \leq 0.99$). Then we discussed the coding guide along with any inconsistencies in categorization to reach a consensus. We repeated this process until our categorization substantially matched, indicating that our coding guide was successfully constructed. We, therefore, repeated this process three times (i. e., independently classifying 60 commits). Finally, our agreement rate achieved substantial agreement in two consecutive iterations. Table 5 shows all agreement rates across three authors for each repetition.

Through this process, we identified two perspectives: *operation* and *target*. The operation indicates what kind of operations are applied, such as adding a new statement or changing an expression; the target indicates source code elements where the operation is applied, such as expressions. The details of the coding guide are described below.

We utilized the following coding guide to categorize micro commits in terms of the operations.

- *add:* This refers to operations that add a completely new entity.
- *replace:* This refers to operations that modify an entity.
- *remove:* This refers to operations that completely remove an entity.
- *multi:* This code indicates that multiple operations are applied.
- *no:* This code indicates that no functional change is applied.

We utilized the following coding guide to categorize micro commits in terms of the targets.

- *identifier:* This refers to commits that only modify identifiers, such as variable names. If other entities, such as parentheses, are included, it would not be labelled as an identifier but would be considered an expression.
- *statement:* This refers to commits that modify a complete statement, including the semicolon (;), such as an entire function call with its semicolon. C's #include preprocessor statement is also regarded as a statement.
- *constant:* This refers to commits that only modify literals, such as strings or numbers. If other entities, such as parentheses, are included, they would not be identified as a constant but would be considered an expression.
- *declaration:* This refers to commits that modify declarations, such as variable declarations. However, if the commit can be classified as "identifier" or "constant", it should be categorized under these two categories rather than "declaration".

– *control flow:* This refers to commits that modify the control flow of execution, such as adding a new "else" statement.

– *expression:* If a commit does not match other categories and involves modifying a part of a statement, it would be classified into this category. Additionally, this category includes transformations from constants to variables or vice versa, as well as conversions from a variable to a pointer and vice versa.

– *multi:* This code suggests that operations are performed on multiple targets.

– *no:* This code indicates that no functional change is applied.

Table 6 shows the summary and examples. This represents different types of activities performed by micro commits. Let us describe two example commits. Listing 6 shows an example commit. This commit changes a function call and its argument. More specifically, the identifiers of the function call and the argument value are replaced so that we classify this commit as operations=replace, and targets=identifier. The commit of Listing 7 replaces an expression "++" into "−−". Hence, we classify this commit as operations=replace, and targets=expression.

Because our agreement rates for operation and target based on our coding guide achieved almost perfect and substantial agreement respectively (Table 5) and we made an internal consensus of the coding guide, only the first author manually classified the 400 micro commits, similar to previous studies [13, 14, 42, 43]. All our manual categorizations are available in our sheet[a]. The sample size in manual inspection was determined as a statistical representative with a confidence level of 95% and a confidence interval of 5% for 150,967 micro commits from all studied projects.[b] The confidence interval, also known as the margin of error, indicates the potential percentage difference between the characteristics obtained from the sampled micro commits and those obtained from the population. The minimum sample size with this confidence level and this confidence interval is 383. For safety, we also inspect 17 additional micro commits. Therefore, we classify a total of 400 micro commits.
...

[a] https://docs.google.com/spreadsheets/d/1M6ifKvufH2JV_ZAYcG6j_eEuSeeEjexWavf2eimHyQk/edit?usp=sharing
[b] https://www.surveysystem.com/sscalc.htm

- The method used to select the 383 sample commits from the projects is not explained clearly. Furthermore, there seems to be an uneven representation of the projects and programming languages in the sample set. This is evident from the fact that most of the commits in the "manual_analysis_result" spreadsheet of the replication package are from the Linux project. This raises concerns about the validity of the sampling strategy.

Response: Thank you very much for raising this concern. To mitigate this risk, we manually inspected additional micro commits from each project. We do not observe significant differences across the projects. We have added this to the list of threats.

Section 8.3 Internal Validity

...
Finally, we randomly sampled 400 micro commits from all projects. Therefore, our sampled micro commits may be biased by the size of the original projects. To mitigate

this risk, we manually inspected additional micro commits from each project. We do not observe significant differences across the projects.
...

- P16, "with a confidence level of 95% and a confidence interval of 5" => 5 is the margin of error. The confidence interval is 95+-5%.

Response: Thank you very much. We have revised the description.

Section 5.1 Approach

...
The sample size in manual inspection was determined as a statistical representative with a confidence level of 95% and a confidence interval of 5% for 150,967 micro commits from all studied projects.[a] The confidence interval, also known as the margin of error, indicates the potential percentage difference between the characteristics obtained from the sampled micro commits and those obtained from the population.
...

[a] `https://www.surveysystem.com/sscalc.htm`

- P17, "micro commits are more likely to be defect-fixing activity than other commits." => Interestingly, this corroborates the findings of Hattori and Lanza [1] (a paper missing from the related work), who found that "Tiny commits are more related to corrective activities, followed by forward engineering and reengineering, that alternate positions."

Response: Thank you for pointing out the missing reference. We have now included it in Section 7.3.

Section 7.3 Program Repair

...
Also, this finding confirms our initial assumption that micro commits are used more frequently for maintenance purposes than non-micro commits. Interestingly, Hattori and Lanza [15] found similar results, noting that tiny commits are often associated with corrective activities.
...

- P17, Fig. 6.: It is worth noting that approximately 50% of the commits have been categorized. What has happened to the remaining commits? If the classification method cannot identify them, it poses a significant risk to the accuracy of the results, which should be noted in Section 8.3 Internal Validity.

Response: Thank you very much for raising this concern. We acknowledge your concern and have added it to the list of threats.

Section 8.3 Internal Validity

...
While we believe the maintenance activities defined by Swanson are acceptable, future studies are necessary to use other sets. Also, if commit messages do not contain any keywords, we exclude those commits from the analysis. However, it is possible that

these commits are related to maintenance activities. Using more precise methods would enhance the validity of this analysis.
...

References:
[1] Hattori, Lile P., and Michele Lanza. "On the nature of commits." 2008 23rd IEEE/ACM International Conference on Automated Software Engineering-Workshops. IEEE, 2008.

Response to Reviewer R2

Summary The manuscript "An Empirical Study of Token-based micro commits" provides a new perspective on micro commits. In comparison to prior research, which considered micro commits based on the number of lines within a textual diff (e.g., a change only to a single line), this paper rather uses the number of tokens that are added and removed as part of a change to characterize the micro commits. The data shows that most one-line changes are micro commits according to this token-based definition, but also that there are other commits that span multiple lines that fall within this definition. The characterization of the commits further shows that such commits are typically a replacement of a single token. Which type of token seems to be language dependent as this was observed to be different between two C and two Java projects.

Overall, I like the concept of the paper and the motivation to better understand small changes. However, I do not think this work is ready for publication yet and requires further revisions. Please find my detailed comments below:

Summary: We thank Reviewer 2 for your constructive comments. Based on your comments, we've updated the rationale behind the definition of micro commits. Additionally, we have updated the coding process description and results presentation to enhance readability. The details are as follows.

R2.1 Major issues:
1) The definition of micro commit seems to be a moving target within the paper. At the beginning, it seems like this will be only based on tokens. Then, suddenly, there is the parallel notion of one-line changes as micro commits. Then, the definition is shifted again to be based on added and changed tokens. This is sometimes referred to as "5 added and changed tokens" and sometimes as "5 added and/or changed tokens". Both wordings are ambiguous and could mean either five tokens total or five added and five deleted tokens (ten total). A careful reading of section 4.2 seems to rather imply that the definition that was used is all commits where the maximum number of either added or removed tokens is less than or equal to five. This needs to be cleaned up such that there is one consistent and clear definition that is introduced once.

Response: Thank you for pointing out the unclear definition. To clarify this point in the paper, we have updated the descriptions. For example, in the revised paper, we do not refer to one-line changes as micro commits. Also, we clearly define micro commits in Section 1. micro commits refer to commits that add at most five tokens and remove at most five tokens. A similar reviewer comment and our response can be found in R1.1. The response of R1.1 is related to the responses to R2.2, R2.3, and R2.5.

Abstract

...
To address this limitation, in this paper, we define *micro commits*, a type of small change based on changed tokens. Our goal is to quantify small changes using changed tokens. Changed tokens allow us to identify small changes more precisely. In fact, this token-level definition

can distinguish the above example.
...

Section 1 Introduction

...
While prior studies [2, 18, 37] use churn (number of lines added and removed) to identify small changes (e. g., small commits and One-line commits [37], or extra-small commits [2]), it has one significant limitation: they consider the line to be the finest-grained entity of changed source code. More specifically, such a definition overlooks the details of what has changed in a line [11, 34, 38]. For instance, when several lines have a small change (such as an identifier being renamed in a few places), these modifications might appear as one line added and one line removed for each change, rather than a single identifier change.

Another problem is that splitting or joining a line of code that is being modified can result in noise. For example, splitting a line into two would be reflected as a change to multiple lines in version control systems (e. g., Git), and this type of change can add noise to the analysis of the history of the development process.

These limitations sometimes cause researchers to fail in accurately quantifying small changes. For example, Listing 2 shows a commit in the Linux repository that changes a few lines (i. e., three added and two deleted lines). While this commit corresponds with a multiple-line change and may not correspond to a One-line commit, it only adds a token "static". This is similar to Listing 1 corresponding with a One-line commit that only adds a token "static". Studying the actual changed tokens instead of the lines can provide a better understanding of the characteristics of the small changes.

In this paper, we define a new class of commits: *micro commits*. **Micro commits are commits that add at most five tokens and remove at most five tokens of source code**. We aim to quantify small changes using the token-level definition (i. e., micro commits) rather than relying on the line-level definition (i. e., One-line commits). This token-level definition allows us to identify small changes more accurately, and use token information to characterize them. We conducted an empirical study on four large, mature open-source projects to: a) demonstrate that micro commits are common, accounting for between 7.45 and 17.95% of all studied commits in the studied projects, b) understand their qualitative and quantitative characteristics, and c) show our definition of micro commits (a threshold of 5 added and removed tokens) includes approximately 90% of all One-line commits, yet only approximately 40–50% of micro commits are One-line commits.
...

Section 3.2 One-line Commits and Micro Commits

...
We extracted micro commits based on the hunks provided by Git repositories processed by `cregit`. Micro commits refer to commits that include a maximum of five added tokens and five deleted tokens across all hunks. This number was chosen for the following reasons.

- In the languages being studied (C and Java), it is highly unlikely to add a new statement with only five tokens, suggesting that such commits carry out minor modifications. For example, within five tokens, developers can only add a function call with one parameter and an ending semicolon: `name(parm);` includes two identifiers, two parentheses, and one semicolon.

- In the systems we studied, between 7.45 and 17.95% of all studied commits add at most 5 tokens and remove at most 5 tokens.

...

R2.2 2) I am really confused what the message of the results for RQ1 should be and how these results motivate the threshold of five adds/deletes for the subsequently used definition of micro commits. I believe this is in part because I cannot really extract a lot of information from the figures. Figure 1 suffers from a bad binning, which basically hides the information beyond the fact that about 50% of changes have one added and deleted token. The color-scale hinders me from seeing more, as the contrast between other values is weak. With Figure 2, I wonder why only the maximum is considered and not the total number. Since I could not really read Figure 1, do not understand the impact of this choice. Figure 4 yields more information as this seems to imply that commits are mostly replacements (diagonal) or additions (bottom line) and that even when all commits are considered, this is often relatively small. But why any of this then yields a meaningful definition of micro, is something I cannot really follow. In other words: why not 4 tokens? Or 3? Or 7? I acknowledge that there is an intent to derive this threshold from data, rather than to define a magic threshold like "one-line changes". Its just that, at least I, do not really see how this is not just a different kind of magic right now. I would wish for a clearer explanation here.

Response: Thank you very much for raising the concern about the definition of micro commits. Similar concerns were raised by two other reviewers. Therefore we proceeded to define micro commits with a threshold of 5 tokens. This is similar to other papers that have arbitrarily defined small commits (e.g., Purushothaman et al. [5] defined *small commits* as those modifying less than 10 lines, and Alali et al. [1] defined *extra-small commits* as those adding at most 5 lines of code).

We support our decision as follows: (1) In the languages being studied (C and Java), it is highly unlikely to add a new statement with only five tokens, suggesting that such commits carry out minor modifications. For example, within five tokens, developers can only add a function call with one parameter and an ending semicolon: `name(parm);` includes two identifiers, two parentheses, and one semicolon. (2) In the systems we studied, between 7.45 and 17.95% of all studied commits add at most 5 tokens and remove at most 5 tokens.

Also, we have relocated the original RQ1 to RQ3 in the revised paper. RQ3 now compares micro commits with one-line commits to clarify their differences. We revised the experiments and the presentation of results for RQ3 in response to this comment. Lastly, we have expanded the discussions in the threats to validity section. This update discusses the impact of the threshold on our analysis.

Abstract

...
To address this limitation, in this paper, we define *micro commits*, a type of small change based on changed tokens. Our goal is to quantify small changes using changed tokens. Changed tokens allow us to identify small changes more precisely. In fact, this token-level definition can distinguish the above example.
...

Section 1 Introduction

...

While prior studies [2, 18, 37] use churn (number of lines added and removed) to identify small changes (e. g., small commits and One-line commits [37], or extra-small commits [2]), it has one significant limitation: they consider the line to be the finest-grained entity of changed source code. More specifically, such a definition overlooks the details of what has changed in a line [11, 34, 38]. For instance, when several lines have a small change (such as an identifier being renamed in a few places), these modifications might appear as one line added and one line removed for each change, rather than a single identifier change.

Another problem is that splitting or joining a line of code that is being modified can result in noise. For example, splitting a line into two would be reflected as a change to multiple lines in version control systems (e. g., Git), and this type of change can add noise to the analysis of the history of the development process.

These limitations sometimes cause researchers to fail in accurately quantifying small changes. For example, Listing 2 shows a commit in the Linux repository that changes a few lines (i. e., three added and two deleted lines). While this commit corresponds with a multiple-line change and may not correspond to a One-line commit, it only adds a token "static". This is similar to Listing 1 corresponding with a One-line commit that only adds a token "static". Studying the actual changed tokens instead of the lines can provide a better understanding of the characteristics of the small changes.

In this paper, we define a new class of commits: *micro commits*. **Micro commits are commits that add at most five tokens and remove at most five tokens of source code**. We aim to quantify small changes using the token-level definition (i. e., micro commits) rather than relying on the line-level definition (i. e., One-line commits). This token-level definition allows us to identify small changes more accurately, and use token information to characterize them. We conducted an empirical study on four large, mature open-source projects to: a) demonstrate that micro commits are common, accounting for between 7.45 and 17.95% of all studied commits in the studied projects, b) understand their qualitative and quantitative characteristics, and c) show our definition of micro commits (a threshold of 5 added and removed tokens) includes approximately 90% of all One-line commits, yet only approximately 40–50% of micro commits are One-line commits.

...

Section 3.2 One-line Commits and Micro Commits

...

We extracted micro commits based on the hunks provided by Git repositories processed by `cregit`. Micro commits refer to commits that include a maximum of five added tokens and five deleted tokens across all hunks. This number was chosen for the following reasons.

- In the languages being studied (C and Java), it is highly unlikely to add a new statement with only five tokens, suggesting that such commits carry out minor modifications. For example, within five tokens, developers can only add a function call with one parameter and an ending semicolon: `name(parm);` includes two identifiers, two parentheses, and one semicolon.
- In the systems we studied, between 7.45 and 17.95% of all studied commits add at most 5 tokens and remove at most 5 tokens.

...

Project	#total commits	#studied commits	#One-line commits	Proportion(%)
Camel	60,911	38,458	2,405	6.25
Hadoop	69,997	53,796	2,302	4.28
Linux	1,048,688	802,726	65,858	8.20
Zephyr	40,883	25,542	1,979	7.75

Table 1: The proportion of One-line commits in the studied projects

Project	#intersects	#One-line	#Micro	%One-line	%Micro
Camel	2,131	2,405	4,230	88.6	50.4
Hadoop	2,069	2,302	4,010	89.9	51.6
Linux	59,836	65,858	138,142	90.9	43.3
Zephyr	1,849	1,979	4,585	93.4	40.3

Table 2: The number and proportion of the intersection between One-line commits and micro commits in each commit type (i. e., One-line or micro). The column of "#intersects" indicates the intersection; the columns of "#One-line" and "#Micro" indicate the number of One-line commits and micro commits; the column of "%One-line" and "%Micro" indicate the proportion of intersection in each commit type (i. e., One-line commits and micro commits).

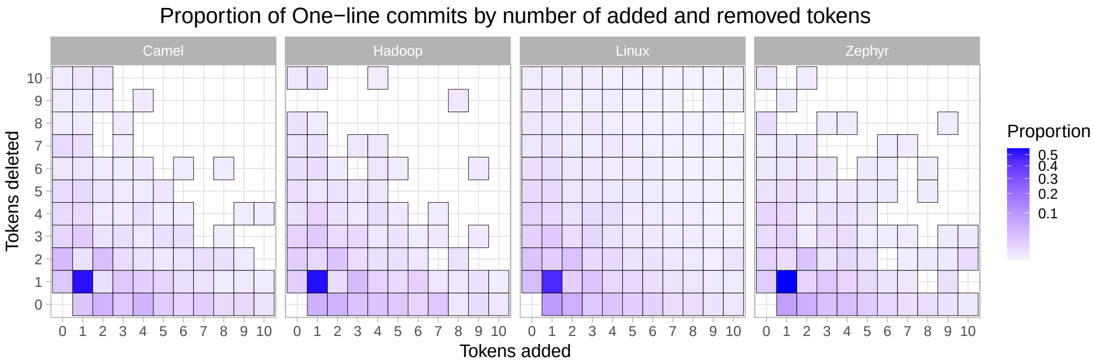


Figure 3: Proportion of **One-line commits** by the number of tokens added or removed. The x and y-axis show the added and deleted tokens, and each cell indicates the proportion of commits.

Section 6.2 Results

Approximately 90% of One-line commits consist of at most five tokens. Table 1 shows the number of One-line commits. As described in Section 2, there are a non-negligible number of these commits in the studied projects (4.28–8.20%). Figure 3 shows the proportion of One-line commits according to the number of tokens that they have added and removed between 0 and 10. As can be seen, there are a significant number of One-line commits that remove and add exactly one token (between approximately 50 and 63% of all One-line commits). Furthermore, except for the case in the Hadoop project where no commits add or delete five tokens, all cells with five or fewer added and deleted tokens have more than one One-line commit across all projects. This implies that there are no empty cells within five added or deleted tokens except for one cell in the Hadoop project. Also, the distribution of One-line commits, with more than five tokens, varies across the projects. For instance,

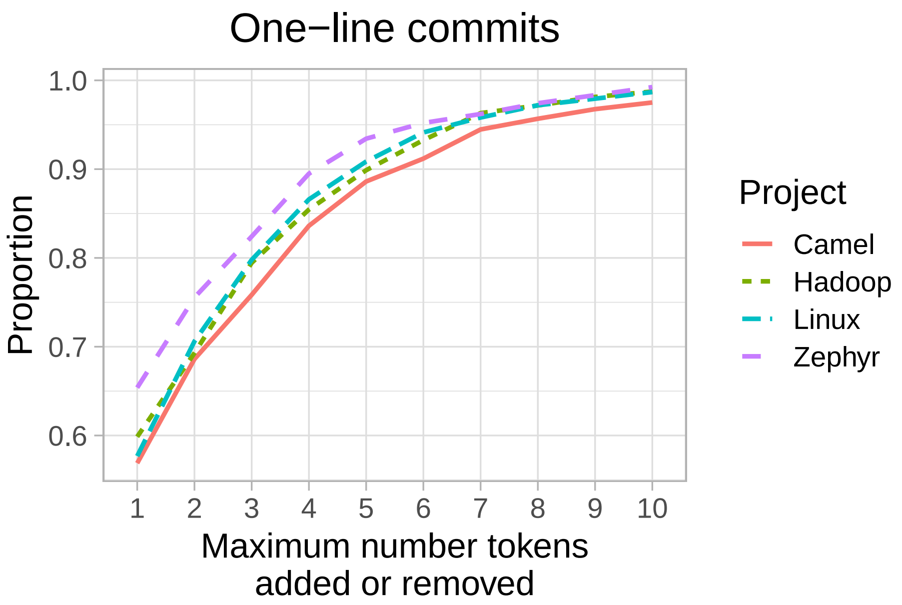


Figure 4: Accumulated distribution of One-line commits in terms of the maximum number of added or removed tokens

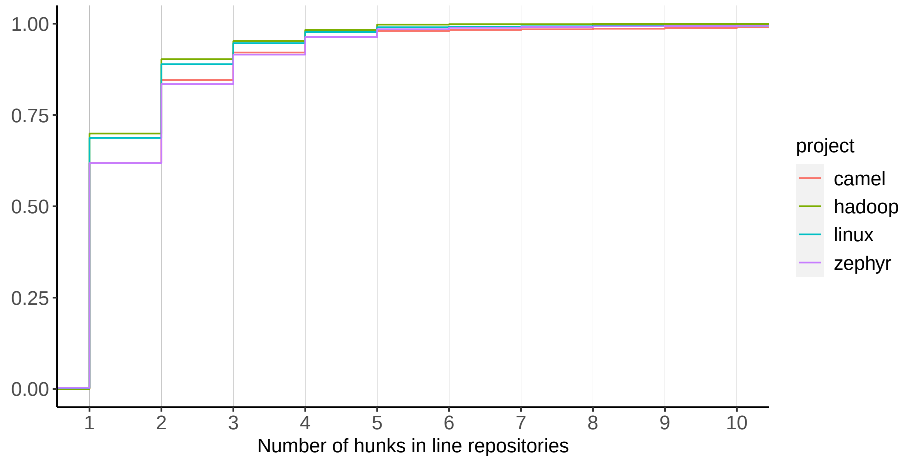


Figure 5: Accumulated distribution of micro commits ($N = 5$) in terms of the number of hunks included

Project	Micro commits	Prop (%)	One-token commits	Prop (%)
Camel	4,230	11.00	1,319	3.43
Hadoop	4,010	7.45	1,288	2.39
Linux	138,142	17.21	32,973	4.11
Zephyr	4,585	17.95	1,247	4.88

Table 3: Number of micro commits and One-token commits and their proportion with respect to all source-code commits.

in the Hadoop and Zephyr projects, there are cells with no One-line commits of more than five deleted tokens and less than or equal to one added token. In contrast, every cell in the Camel and Linux projects has at least one One-line commit. Hence, the majority of One-line commits add or remove at most five tokens, and this finding is generally consistent across all projects.

Figure 4 shows the accumulated distribution of One-line commits according to the maximum number of tokens they add or remove. We use the maximum number of tokens added or removed in this figure. This is because our definition of a micro commit applies the same threshold of five tokens to both the number of added and removed tokens. As can be seen, between approximately 57% and 65% add-or-remove at most one token, between 76% and 82% add-or-remove at most three tokens, and between 89% and 93% add-or-remove at most five tokens. Thus, approximately 90% of One-line commits can be covered by our micro commits.

The number of modified hunks is also a crucial characteristic of commits. By our definition, One-line commits only modify one location in the source code (i. e., one hunk). We define micro commits based on the number of tokens, so even if a commit is spread across multiple locations (i. e., multiple hunks), it it still considered a micro commit if the number of modified tokens is below a certain threshold. This is a significant distinction compared to One-line commits. Therefore, we do not impose any limits on the number of modified hunks.

Figure 5 illustrates the accumulated distribution of the number of hunks included in micro commits to investigate their difference from One-line commits. Approximately 70% (Linux and Hadoop) or 60% (Zephyr and Camel) of micro commits contain a single hunk, while the remaining commits encompass two or more hunks. Hence, while approximately 70% or 60% of micro commits share characteristics with One-line commits, the remaining 30% or 40% represent commits that one-line commits do not detect, even if they modify the same number of tokens.

In conclusion, although micro commits can encompass nearly all One-line commits, the reverse is not typically true: One-line commits do not generally cover micro commits. Indeed, Table 2 in Section 3.3 reveals that around 90% of one-line commits can be encapsulated by micro commits. However, only approximately 40% (for Linux and Zephyr) or 50% (for Camel and Hadoop) of micro commits can be encapsulated by one-line commits. Therefore, micro commits provide new insights compared to one-line commits.

Summary of RQ3

Approximately 90% of One-line commits add or remove at most five tokens. Therefore, nearly all one-line commits can be covered by micro commits. In contrast, 30 to 40% of micro commits include two or more hunks that are not covered by one-line commits. In fact, only approximately 40% (for Linux and Zephyr) or 50% (for Camel and Hadoop) of micro commits can be encapsulated by one-line commits. Therefore, the characteristics of micro-commits can help us understand the attributes of small changes, including those in one-line commits and commits not identified by one-line commits.

Section 8.2 Construct Validity

We define micro commits based on the number of changed tokens. However, micro commits are a general term, and we can make different definitions. The key characteristic of micro commits is that such commits change a small code fragment. Our analysis (RQ1 and 2) shows that our definition is consistent with this characteristic. Hence, we believe our definition can be acceptable. However, our definition may not be the best; thus, future studies are necessary to find a better definition than our first one. For example, future studies can investigate different thresholds for the number of tokens. Also, studying different thresholds for added and deleted tokens (e. g., 3 added tokens and 5 deleted tokens) can be beneficial. Additionally, they can consider changes to source code comments. This definition would encompass not only maintenance activities related to code logic, but also various other maintenance activities.
...

R2.3 3) Part of the motivation is pretty paradox. On the one hand, the critique of one-line changes as criterion for micro changes is strong. On the other hand, this is used immediately afterwards to understand properties of micro changes. This happens both in the introduction and in Section 2 with the motivation, as well as later when the definition of micro changes is changed. Together with the above weak argument for the threshold and the confusing definition of micro changes used here, I think this is a key aspect that needs to be revisited and cleaned up. My suggestion (feel free to ignore): use the one-line changes to derive the token-based threshold, then identify how many additional commits you now identify, likely because of bad diffs as in the example in Listing 2.

Response: Thank you for pointing out this unclear explanation. Similar to the response for R2.1 and R2.2, we've revised the justification for the micro commits and clarified their definition. Please refer to our responses to R2.1 and R2.2.

R2.4 4) The motivation often stresses that a "significant effort" is invested in the creation of micro commits. Is there any support for this statement? Why is the effort not "micro" as well. At least for me, changes that touch only few tokens/lines are less effort for coding, testing, reviewing, and documenting than larger changes. What is so bad about such changes? These statements need either a strong support or need to be modified.

Response: Thank you for your feedback. We have revised the statement to tone it down. We now focus on the observation that small changes often address deficiencies introduced by other modifications [5]. A good understanding of the characteristics of such small changes can help understand the types of errors introduced. This knowledge can be used to improve quality assurance approaches, such as automated program repair. We have revised the Abstract and Section 1 based on this point.

Abstract

In software development, developers frequently apply maintenance activities to the source code that change a few lines by a single commit. A good understanding of the characteristics of such small changes can support quality assurance approaches (e.g., automated program repair), as it is likely that small changes are addressing deficiencies in other changes; thus, understanding the reasons for creating small changes can help understand the types of errors introduced. Eventually, these reasons and the types of errors can be used to enhance quality assurance approaches for improving code quality.
...

Section 1 Introduction

...
Improving code quality is an ultimate goal for software engineering researchers, and several quality assurance approaches have been widely studied so far, such as automated program repair (APR), defect prediction, and fault localization. A good understanding of the characteristics of very small changes can support such approaches, as it is likely that such changes are addressing deficiencies in the system [5]; thus, understanding the characteristics of creating small changes can help understand the types of errors that other changes introduce and potentially help with program repair. Eventually, the information can be used to enhance quality assurance approaches for improving code quality.
...

R2.5 5) The filter for micro changes seems to be too strict. Notably, header files for the C code are dropped as well. Thus, commits that touch multiple code files (a .c file and the associated .h header) could still be considered micro. Additionally, I miss a criterion that only "single-hunk commits" are considered as candidates for micro changes. Otherwise, there needs to be a reason why changes that are touching multiple locations in a file or even multiple files, can be considered micro changes.

Response: Thank you for highlighting the inclusion criteria of micro commits. We concur with the comment that the header file is crucial for the C code. Therefore, we have removed this exclusion criterion and re-executed the experiments. Consequently, our analysis results have slightly changed. However, the findings and implications generally remain unchanged. We do not use the single-hunk criterion. We would like to define micro commits as those that change the same number of tokens, regardless of whether they affect multiple locations. Otherwise, our micro commits would be similar to the one-line commits.

Furthermore, based on our discussion, we have decided to remove the single-file criterion from the definition of micro commits. If we include the header file as a target, we may need to analyze commits that simultaneously change both the header file and the C file. We have clarified these points further in the paper.

Section 1 Introduction

...
In this paper, we define a new class of commits: *micro commits*. **Micro commits are commits that add at most five tokens and remove at most five tokens of source code**. We aim to quantify small changes using the token-level definition (i. e., micro commits) rather than relying on the line-level definition (i. e., One-line commits). This token-level definition allows us to identify small changes more accurately, and use token information to characterize them. We conducted an empirical study on four large, mature open-source

projects to: a) demonstrate that micro commits are common, accounting for between 7.45 and 17.95% of all studied commits in the studied projects, b) understand their qualitative and quantitative characteristics, and c) show our definition of micro commits (a threshold of 5 added and removed tokens) includes approximately 90% of all One-line commits, yet only approximately 40–50% of micro commits are One-line commits.
...

Section 3.2 One-line Commits and Micro Commits

...
We extracted micro commits based on the hunks provided by Git repositories processed by `cregit`. Micro commits refer to commits that include a maximum of five added tokens and five deleted tokens across all hunks. This number was chosen for the following reasons.

- In the languages being studied (C and Java), it is highly unlikely to add a new statement with only five tokens, suggesting that such commits carry out minor modifications. For example, within five tokens, developers can only add a function call with one parameter and an ending semicolon: `name(parm);` includes two identifiers, two parentheses, and one semicolon.
- In the systems we studied, between 7.45 and 17.95% of all studied commits add at most 5 tokens and remove at most 5 tokens.

...

R2.6 6) Related to the filters above, the definition of micro credits is currently also misleading in a different respect: the "micro" part is *only* for the code. Changes to comments or non-code files can be arbitrarily large. This should be reflected appropriately in the motivation of the paper, including an explanation why it is okay to ignore other aspects and which aspects those are. As a side note: these other aspects that are changed could be one of the reasons that change effort for micro commits is larger (per changed token) than for other commits and could possibly even add data regarding what you claim regarding the "significant effort" (see comment 4).

Response: Thank you. We agree with the reviewer's thoughts. Making changes to comments is important for maintenance purposes in practice. The primary reason for excluding comments and non-code files is to prioritize maintenance activities for code logic. As mentioned in Section 1, our intention is to support various software engineering approaches (e.g., defect prediction), which typically prioritize code logic over non-code. Indeed, defect prediction studies typically do not take into account comment issues when identifying target defects. While we acknowledge the importance of changes made to comments for maintenance purposes, this perspective is beyond the scope of our paper. We have added this reason. Furthermore, we have discussed this perspective in future work. This is because maintenance activities for comments are also of interest and should be considered when defining other types of micro commits.

Section 3.2 One-line Commits and micro commits

...
Source code comments are important for source code and making changes to comments are also maintenance activities. However, in this paper, we exclude comments and execute our analysis. The reason is to prioritize maintenance activities for code logic. As mentioned in Section 1, our intention is to support various software engineering approaches (e.g., defect prediction), which typically prioritize code logic over comments. Indeed, defect prediction

studies typically do not take into account comment issues when identifying target defects [19, 24, 33]. While we acknowledge the importance of changes made to comments for maintenance purposes, this perspective is beyond the scope of our paper.
...

Section 8.2 Construct Validity

...
Hence, we believe our definition can be acceptable. However, our definition may not be the best; thus, future studies are necessary to find a better definition than our first one. For example, future studies can investigate different thresholds for the number of tokens. Also, studying different thresholds for added and deleted tokens (e.g., 3 added tokens and 5 deleted tokens) can be beneficial. Additionally, they can consider changes to source code comments. This definition would encompass not only maintenance activities related to code logic, but also various other maintenance activities.
...

R2.7 7) I do not understand the message of Figure 5. Figure 4 shows that upwards of 60% of all replacements are names and literals for all projects. In contrast, Figure 4 looks at exact token values - and thereby more or less cannot consider these upwards 60% of tokens in the analysis of most common replacements. This really reduces the value of Figure 5. Consequently, I wonder: what is the intended message here? That Java has a . that is replaced (method chaining, imports, . . .) and that this does not appear for a language in which this character does not have a similar importance does not really motivate this analysis for me.

Response: Thank you for your comment. Originally, the message of Figure 5 is that the most frequently touched tokens in micro commits differ between Java and C. This suggests that at the token-level, we need to evaluate software engineering methods (such as automated program repair) across different programming languages to assess their generalizability. However, we concur that considering the token types would enhance this analysis.

Therefore, we have redesigned this analysis to focus on tokens of the top-3 most frequently touched token types. Due to the revision of the micro commits definition based on reviewers' comments, there have been slight changes to Figure 4, which represents token types. Specifically, we observed the top-3 token types, as opposed to the top-4 in the first version. These top-3 token types account for approximately 80% in Java and 60% in C.

The new results (Figure 5) also indicate that the tokens corresponding to the top-3 token types vary between Java and C, a finding consistent with the first version. In contrast, we observed similar tokens within the programming language. Specifically, in Java, condition values (e. g., true/false, null, and numeric values) are observed, while in C, parentheses and types (e. g., int, u32_t, and u8_t) are observed. This result suggests that even when the most frequently touched token types are identical, their proportions and actual tokens vary across programming languages. However, within the same language, similar tokens may be observed. Additionally, to enhance the value of this analysis, we have included further discussion about these results in Section 7.3, details of which are provided below.

Note that due to the reorganization of RQs, Figure 4 and Figure 5 in the original paper are now Figure 1 and Figure 2 in the revised paper.

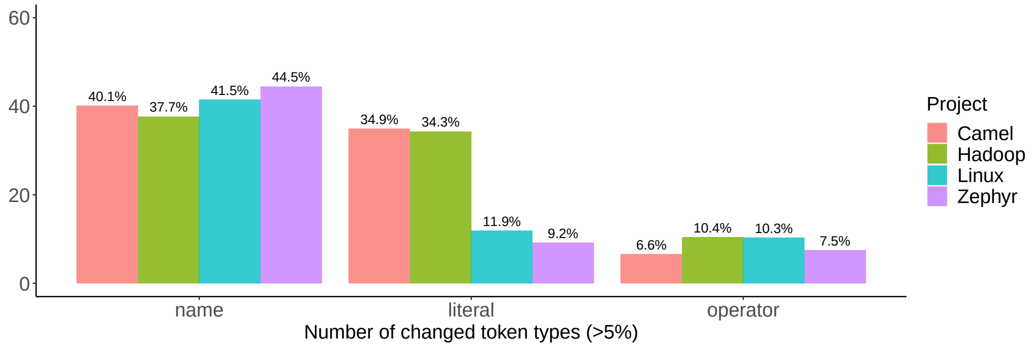


Figure 1: Proportions of changed token types (> 5%)

Section 4.2 Results

...

The top-3 most frequently touched token types in micro commits are generally the name, literal, and operator token types. Figure 1 shows the frequently added/removed token types by micro commits that account for more than 5% in all projects. We found three token types, which we refer to as the top-3 most frequently touched token types. The token type most frequently included in micro commits is the name token (e.g., the name of a variable or function), the second one is the literal token (e.g., 123, ‘a’, “test”), and the third one is the operator token (such as +). Also, the proportion of the literal token is significantly different between Java and C. Specifically, while the proportion of name tokens is more than three times larger than that of literal tokens in the projects written in C, the difference is less than two times in the projects written in Java. While the proportion of the operator tokens is relatively small, these are also included in the top-3 most frequently touched token types. Hence, micro commits usually modify name, literal, and operator tokens in most cases, but their proportions may differ between programming languages and their token types.

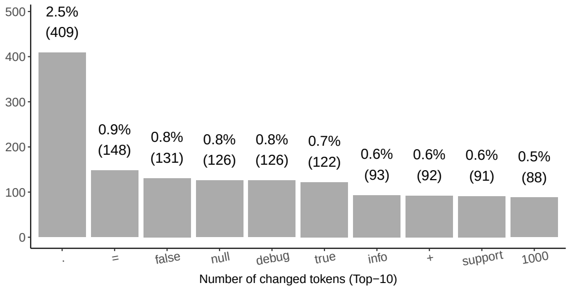
While the tokens corresponding to the top-3 token types differ between Java and C, we observe similar tokens within the same language. Figure 2 shows the top-10 most frequently occurring tokens for the top-3 token types. In Java, boolean literals (e. g., true/false, null), and numeric literals were the most commonly observed, while in C, they were the tokens for 0/1, parentheses and names for types (e. g., int, u32_t, and u8_t).

In conclusion, the types of tokens most frequently changed are the same in both programming languages, but the actual tokens are different.

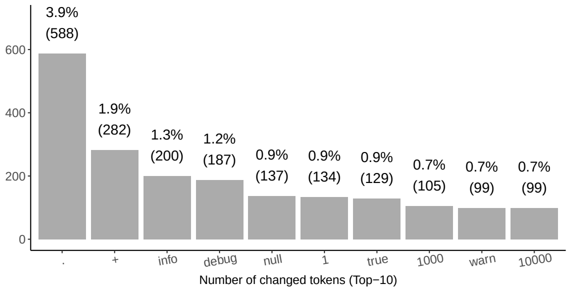
...

Section 7.3 Program Repair

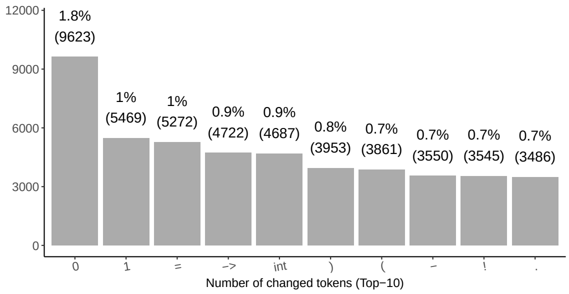
The results of RQ1 showed that micro commits frequently modify a single token, and its token type is name, literal, or operator. **Studying micro commits could help understand how software is modified with such a tiny amount of change, and provide datasets that improve methods that attempt to modify software automatically.** For example, datasets based on micro commits might improve data-driven program repair approaches that have been studied so far [20,29,31]. One potential idea involves utilizing our observations of frequently modified token types and tokens in Java and C. Our observations indicate that while the types of frequently modified tokens are similar, the actual tokens differ across



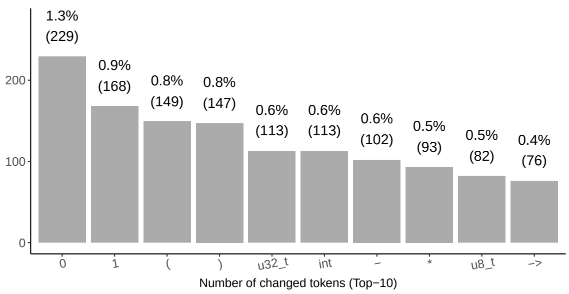
(a) Camel (Java)



(b) Hadoop (Java)



(c) Linux (C)



(d) Zephyer (C)

Figure 2: Numbers of changed tokens

languages. This information is important for developing a program repair approach. When dealing with multiple languages, focusing on token types is crucial. However, when focusing on a specific language, actual tokens can also be beneficial.
...

R2.8 8) The description of the "manual coding" needs to be improved. Currently, I am not sure if open or closed coding was conducted here: did these categories emerge during the coding of the first 20/40/60 commits or were they pre-defined? Why are pair-wise agreements for Cohen's kappa reported instead of Fleiss' kappa which generalizes the interrater agreement to multiple raters? How exactly is "substantially matched" for the agreement defined here? Are 20 commits sufficient to estimate a reliable value for kappa to measure this substantial matching, or - in other words - how big is the risk that kappa is just randomly high enough due to a good agreement on a small and simple subsample? Why is this important part of the study protocol moved to the appendix, but other sections with similar details are in the main body? What is the expected noise given the choice to use a single rater for most of the data? How does this noise possibly affect subsequent results and conclusions? (Please note: I really like that there was this manual inspection of the data and believe that this is the most valuable and interesting part of the paper)

Response: Thank you for your comment. We have updated the description in Section 5.1 to clarify the process. We used an open coding process. Additionally, we have combined the details of our coding process (Appendix) with Section 5.1. Please refer to the updated Section 5.1 below. Finally, we've updated the Section 8.1 Internal Validity section to discuss the threats of this process in more detail.

Section 5.1 Approach

...
Our manual inspection consisted of two phases: (1) *constructing a coding guide* and (2) *manual classification*. Constructing a coding guide for manual classification/annotation is a common practice in the field of mining software repositories [6, 13, 14, 42, 43, 46]. To create the coding guide, we referred to previous studies [6, 13, 14, 42, 43, 46] and followed the process detailed below.

The initial coding guide was first discussed by the first and second authors. Since this is the first study to classify micro commits, we examined both micro commits and other types of commits to develop the initial coding guide. After constructing the initial coding guide, we aimed to reach a consensus among the first three authors for this guide and refine the guide. Specifically, we independently annotated 20 micro commits from a subset of all micro commits. This subset consists of micro commits that only change less than or equal to five tokens in main files (.c or .java files) in the Linux, Hadoop, and Zephyr projects to investigate a single operation commit for refining the coding guide. We computed the agreement rate for these 20 micro commits using Fleiss' Kappa [9] that is used to demonstrate inter-rater agreement when there are more than two raters. It is also frequently applied in the field of mining software repositories [6,13]. The Kappa coefficient is commonly interpreted using the following scale [41]: Slight agreement ($0.01 \leq k \leq 0.20$), Fair agreement ($0.21 \leq k \leq 0.40$), Moderate agreement ($0.41 \leq k \leq 0.60$), Substantial agreement ($0.61 \leq k \leq 0.80$), Almost perfect agreement ($0.81 \leq k \leq 0.99$). Then we discussed the coding guide along with any inconsistencies in categorization to reach a consensus. We repeated this process until our categorization substantially matched, indicating that our coding guide was successfully constructed. We, therefore, repeated this process three times (i. e., independently classifying 60 commits). Finally, our agreement rate achieved substantial

Criteria	First Time	Second Time	Third Time
Operations	0.686	0.669	0.832
Targets	0.425	0.671	0.754

Table 5: Fleiss' Kappa scores for each repetition

agreement in two consecutive iterations. Table 5 shows all agreement rates across three authors for each repetition.
...

Section 8.3 Internal Validity

...
Also, another threat exists in our manual analysis (RQ2). In this analysis, we performed manual labeling to micro commits according to our coding guide. Because this process is performed manually by the first author, the result may have false-positive and false-negative results. Therefore, we have made all labels publicly available to facilitate the validation of future studies. Also, to construct the coding guide, the first three authors independently inspected 20 micro commits three times. This process may also include errors. However, our agreement rate achieved substantial agreement in two consecutive iterations. Hence, we believe the coding guide is reliable. An alternative solution is to use an automatic classification approach rather than manual analysis. We developed a heuristic-based method to classify micro commits into their corresponding targets automatically. However, this method does not yield perfect results. To facilitate replication of this approach, we have included it in our replication package. Finally, we randomly sampled 400 micro commits from all projects. Therefore, our sampled micro commits may be biased by the size of the original projects. To mitigate this risk, we manually inspected additional micro commits from each project. We do not observe significant differences across the projects.
...

R2.9 9) I am not sure how the confidence interval calculation worked. Which assumptions were used? For *what exactly* is this confidence interval calculated? What does it mean that the confidence interval is 5? 5 what? I checked the homepage and it is very generic and seems to be for confidence intervals for mean values (e.g., average age, voting preferences) when conducting surveys. How does this translate to this setting? This requires more details, before I can understand if the used strategy is sound.

Response: As you pointed out, we utilized the general method to determine the sample size. Specifically, we regard the category of micro commits as the population proportion. Our goal is to obtain a large enough sample size to analyze the characteristics of the micro commits category within the population. The confidence interval, also known as the margin of error, indicates the potential percentage difference between the characteristics obtained from the sampled micro commits and those obtained from the population. Therefore, with a 5% confidence interval, the proportion of categories observed in this study may differ by 5% from the proportion present in the overall population. We have revised the description in the paper for clarity.

Additionally, we've increased the sample size to 400 cases to enhance reliability, as the minimum requirement was to analyze 383 micro commits. Moreover, this is a typical method for performing manual classification in empirical software engineering. We will reference additional previous studies to further illustrate this point [2, 3, 7, 8].

Section 5.1 Approach

...
Because our agreement rates for operation and target based on our coding guide achieved almost perfect and substantial agreement respectively (Table 5) and we made an internal consensus of the coding guide, only the first author manually classified the 400 micro commits, similar to previous studies [13,14,42,43]. All manual categorizations are available in our sheet[a]. The sample size in manual inspection was determined as a statistical representative with a confidence level of 95% and a confidence interval of 5% for 150,967 micro commits from all studied projects.[b] The confidence interval, also known as the margin of error, indicates the potential percentage difference between the characteristics obtained from the sampled micro commits and those obtained from the population. The minimum sample size with this confidence level and this confidence interval is 383. For safety, we also inspect 17 additional micro commits. Therefore, we classify a total of 400 micro commits.

[a] https://docs.google.com/spreadsheets/d/1M6ifKvufH2JV_ZAYcG6j_eEuSeeEjexWavf2eimHyQk/edit?usp=sharing
[b] https://www.surveysystem.com/sscalc.htm

R2.10 10) How does the choice to use a simple keyword approach to determine to which of Swanson's categories commits belong affect the results? Keyword-based approaches (and ML approaches as well) are known to induce a fairly large amount of noise in the labels (see, e.g., https://doi.org/10.1016/j.jss.2020.110821) such that mislabels for at least 30% of the data would not be surprising. This impact on the findings needs either to be estimated or, even better, should be mitigated somehow.

Response: Thank you for rising this concern. Reviewer 1 also expressed concern about this analysis (R1.4) because it only evaluates the commits containing specific keywords. Hence, we (1) transferred this analysis from the RQ to the discussion section and (2) manually verified the false-positive commits identified by this analysis.

Because Swanson categories analysis only reveals that micro commits are mainly used to fix bugs, and our manual coding analysis sufficiently addresses the original RQ3, we have relocated Swanson categories analysis from the original RQ3 to the Discussion section under "Section 7.3 Program Repair". In the Discussion section, we focus solely on the *corrective* category, as this finding is a crucial point for program repair techniques.

To verify the accuracy of the categorization into the corrective category, we manually inspect 20 micro commits and 20 non-micro commits identified as corrective, classifying them into three failure types within the corrective category defined by Swanson [6]. If we can't associate any failure types, those would be considered false positive corrective commits. This allows us to estimate the actual number of corrective micro commits and corrective non-micro commits in this categorization. We do not need to examine non-corrective commits to determine the proportion of false-negative corrective commits. This discussion only reports the minimum percentage of corrective commits.

Our manual inspection revealed that there were no false-positive corrective micro commits.[1] In contrast, we found 8 out of 20 false positive corrective commits in non-micro commits. This finding confirms our initial assumption that micro commits are used more frequently for maintenance purposes than non-micro commits. Interestingly, Hattori and Lanza [4] found similar

[1] https://docs.google.com/spreadsheets/d/17cqps6oSkA86GPuUmin3W1FoRZXslQfIuJAkX3vqH28/edit?usp=sharing

results, noting that tiny commits are often associated with corrective activities. In this discussion, we manually inspect only 20 micro and non-micro commits as an initial analysis. Future studies could improve the validity of these initial findings.

Section 7.3 Program Repair

...

Finally, we present initial analysis results for micro commits regarding their maintenance activities. As stated in the introduction, we hypothesize that commits involving the addition or removal of a few tokens are likely intended for maintenance purposes. Therefore, we deduce that exploring micro commits could be beneficial for program repair. To validate this hypothesis, we identify micro commits that fall under the corrective maintenance category as defined by Swanson [40]. Corrective maintenance is performed in response to failures. If corrective maintenance makes up a large proportion of micro commits compared to non-micro commits, it would confirm our hypothesis.

To identify the corrective commits, we followed the methodologies used in prior studies [35, 37], which use keywords in commit messages. More specifically, if at least one of the keywords is included in the commit messages, we classify the commit into the corrective maintenance category. Otherwise, we do not label commits. We used the keyword list defined by Levin et al. [27] as follows: "fix", "esolv", "clos", "handl", "issue", "defect", "bug", "problem", "ticket".

The detailed procedure is as follows.

Step 1: Apply preprocessing to the commit messages using the NLTK package[a] in Python by following the steps below:

- Tokenize the text and convert all words to lowercase.
- Remove stopwords and punctuation.
- Perform stemming on all words.

Step 2: Check if the stemmed commit message contains a keyword.

Step 3: Identify commits that fall under the corrective category.

Micro commits are more likely to be failure-fixing activity than other commits. Figure 6 shows the proportion of corrective micro and non-micro commits. In this figure, we compare the tendency of micro commits (light gray) and non-micro commits (dark gray). Corrective micro commits are larger than non-micro commits. Hence, micro commits distinguishably correspond to the corrective commits. This result shows that micro commits are usually applied to the source code to fix failures.

Also, this finding confirms our initial assumption that micro commits are used more frequently for maintenance purposes than non-micro commits. Interestingly, Hattori and Lanza [15] found similar results, noting that tiny commits are often associated with corrective activities.

It should be noted that the keyword-based approach generally lacks accuracy [3,25]. To verify the accuracy of the identification, we manually inspect 20 micro commits and 20 non-micro commits identified as corrective, classifying them into three failure types within the corrective category defined by Swanson [40]. If we cannot associate any failure types, those would be considered false positive corrective commits. This allows us to estimate the actual number of corrective micro commits and corrective non-micro commits in the

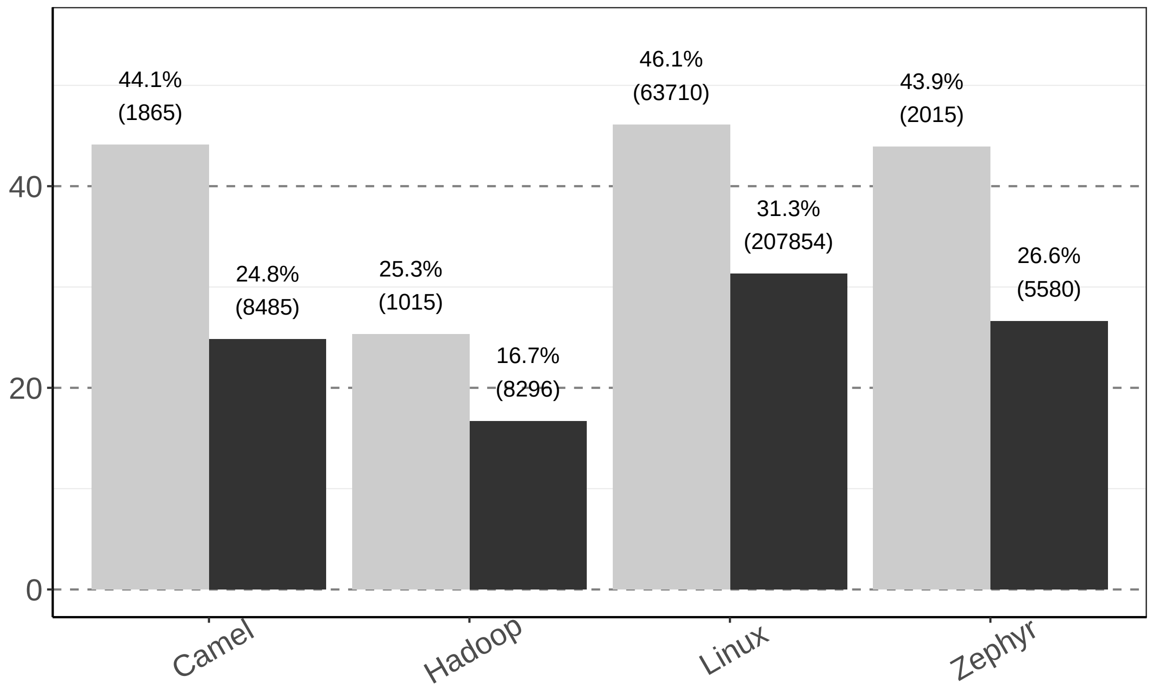


Figure 6: The proportion of commits for the corrective category. Light gray indicates the proportion in micro commits; dark gray indicates the proportion in non-micro commits.

identified commits. We do not examine non-corrective commits to determine the proportion of false-negative corrective commits. This discussion only reports the minimum percentage of corrective commits. Our manual inspection revealed that there were no false-positive corrective micro commits. In contrast, we found 8 out of 20 false positive corrective commits in non-micro commits. This finding suggests that the percentage of corrective micro commits may not change significantly, while the proportion of corrective non-micro commits could decrease. Therefore, our conclusion remains unchanged. In this manual inspection, we inspect only 20 micro and non-micro commits. Future studies could improve the validity of our findings. Our inspection is avaiable in the following spreadsheet.[b]

[a] `https://www.nltk.org/`

[b] `https://docs.google.com/spreadsheets/d/17cqps6oSkA86GPuUmin3W1FoRZXslQfIuJAkX3vqH28/edit?usp=sharing`

Section 8.3 Internal Validity

...

In the discussion, we use keywords to identify the commits related to the corrective maintenance activity as defined by Swanson [40]. While the keyword identification is widely used to categorize commits [18,22,27,28,35,37], it is not perfect [3,25]. To mitigate this threat, we manually review identified commits and estimate their accuracy. Also, there are other sets of maintenance activities that can be used to classify commits, such as the IEEE standard [1]. While we believe the maintenance activities defined by Swanson are acceptable, future studies are necessary to use other sets. Also, if commit messages do not contain any keywords,

we exclude those commits from the analysis. However, it is possible that these commits are related to maintenance activities. Using more precise methods would enhance the validity of this analysis.
...

R2.11 Minor:

- Please use the Swanson's terminology consistently. Section 4.1 currently deviates from this.

  Response: Thank you for the comment. I think this comment pertains to Section 6.1. We have removed the description of the Swanson category from Section 6.1. Additionally, the new discussion in Section 7.3 adheres to the terminology.

- The statement that multi operation micro commits are important for developers should be reconsidered or better motivated based on the results. 10% of micro commits means that these make up at most 1.6% of commits overall (based on the 16% of micro commits from Zephyr), i.e. a very small ratio overall.

  Response: Thank you for the comment. We have revised the sentence as follows.

  Section 6.2 Results

  ...
  Hence, multi-operation micro commits account for a non-negligible portion of the micro commits.
  ...

- "Each hunk includes context lines" (Page 7) is wrong, hunks may contain context lines but do not have to. Case in point: the paper uses a config-option to prevent context lines.

  Response: Thank you for rising this writing issue. We have revised the sentence.

  Section 3.2 One-line Commits and micro commits

  ...
  Each hunk can include *context* lines
  ...

- Some information is redundant. I already mentioned the repeated and sometimes conflicting definition of micro commits above. Section 2 overlaps to a large part with the introduction. Similarly, Section 3.2 overlaps mostly with Section 2. Both times, some (few) new details are added, but most of it is rewording of what was mentioned before. This should be restructured to avoid such duplications.

  Response: Thank you for your suggestion. We have clarified the definition of micro commits. The introduction provides a summary of the entire paper; Section 2 delves into the motivation behind the study; Section 3.2 elaborates on how to manage the output from Git. Hence, while we revised the definition of micro commits, a large portion of these sections remains the same. We believe that each section provides important details to replicate our study.

- I suggest to avoid rhetorical questions.

Response: Thank you for your suggestion. We have removed such questions from Section 4.1 and Section 5.2.

Response to Reviewer R3

Summary The research problem treated in the paper seems interesting. However, I think the paper needs to improve some points before it can be accepted. While my detailed comments are reported below, I noticed that (1) the novelty is unclear, (2) some external factors that may have biased the study were not mentioned, (3) part of the research method applied could be clarified, and (4) lack of threats to validity.

EVALUATION AND COMMENTS FOR AUTHORS

STRENGTHS and WEAKNESSES:
+ The paper is well-written and easy to read;
+ The paper provides a replication package
- Some external factors that might bias the study were not considered (or discussed);
- The novelty of the paper should be clarified;
- Some details/choices could be better explained;
- Lack of threats to validity;

Summary: We thank Reviewer 3 for your constructive comments. In response to your feedback, we have mainly made the following updates to the manuscript: (1) clarified the novelty of the study, (2) included potential factors that may bias the findings, (3) provided more details on the research methods, and (4) expanded the section on threats to validity. The details are described below.

R3.1 1) The paper seems to suggest that the study involved all commits in the four projects. Were the commit messages also analyzed? Were also commits in which minor fixes were clearly reported considered? I expect these types of commits to always fall under micro commits, and in this case, I assume they would have a different impact.

Response: Thank you for presenting this new perspective. We agree with the comment that such commit messages would provide us with additional insights for defining micro commits. However, our intention is to define micro commits based on the size perspective, specifically the number of tokens. Therefore, this perspective is beyond the scope of this paper. We have explained why we excluded comments and also discussed future work from this perspective.

Section 3.2 One-line Commits and micro commits
... Source code comments are important for source code and making changes to comments are also maintenance activities. However, in this paper, we exclude comments and execute our analysis. The reason is to prioritize maintenance activities for code logic. As mentioned in Section 1, our intention is to support various software engineering approaches (e.g., defect prediction), which typically prioritize code logic over comments. Indeed, defect prediction studies typically do not take into account comment issues when identifying target defects[19, 24, 33]. While we acknowledge the importance of changes made to comments for maintenance purposes, this perspective is beyond the scope of our paper. ...

Section 7.4 Size-perspective vs. Semantic-perspective for Defining micro commits

In this paper, we define micro commits through size metrics (i. e., the number of tokens). This is because we would like to assist with software engineering research, such as program repair. However, micro is a general term, and micro commits can be defined not only by size but also by semantic aspects. For instance, tangled commits [8,16,23] can be considered non-micro, whereas non-tangled commits can be categorized as micro. Additionally, defect-fixing commits can be categorized as micro or non-micro depending on the difficulty of the bug being fixed. We could explore these aspects using non-source code resources, such as source code comments, issue reports, and mailing lists. Exploring these semantic-based micro commits can also contribute to software engineering research.

Section 8.2 Construct Validity

...
Hence, we believe our definition can be acceptable. However, our definition may not be the best; thus, future studies are necessary to find a better definition than our first one. For example, future studies can investigate different thresholds for the number of tokens. Also, studying different thresholds for added and deleted tokens (e. g., 3 added tokens and 5 deleted tokens) can be beneficial. Additionally, they can consider changes to source code comments. This definition would encompass not only maintenance activities related to code logic, but also various other maintenance activities.
...

R3.2 2) Another concern is related to the projects' developers. Have their profiles been analyzed in any way? Again, I expect that the knowledge or otherwise experience of the developers may impact the results of the study because their experience and knowledge could impact the commit. In the study, there is no reference to either this point or point 1. I would have expected at least a discussion on the threats to validity.

Response: Thank you for your comment. We agree with the comment. There are several factors that can influence commits. For instance, the way developers write commits can vary depending on the developer and the project. To reduce these influences, we chose and analyzed four projects that involve a large number of developers. As a result, we expect the impact of such influences has been minimized. We have added this point in the threats to validity section.

Section 8.2 Construct Validity

...
There are several factors that can influence commits. For instance, the way developers write commits can vary depending on the developer and the project. To reduce these influences, we chose and analyzed four projects that involve a large number of developers. As a result, we expect the impact of such influences has been minimized.

R3.3 3) I am a bit concerned with the overall novelty of the paper. Looking at the introduction and the related works, it is not clear what is the knowledge gap and how this work improved the literature. The motivation reported "improve the maintenance activities" appears weak and too high-level.

Response: Thank you for rising this concern. Compared to prior studies, this research is the first to define micro commits at a fine granularity, specifically at the token level. The key novelty

Criteria	First Time	Second Time	Third Time
Operations	0.686	0.669	0.832
Targets	0.425	0.671	0.754

Table 5: Fleiss' Kappa scores for each repetition

of this study lies in conducting the analysis at the token level and providing implications for software engineering research. For example, the findings of RQ1 in this study have implications for research in program repair. These findings indicate the need to explore approaches for fixing bugs caused by a single name or literal token. This is because existing automated program repair approaches (e.g., GenProg) may not be effective in such scenarios due to a lack of information to repair the code. These findings and implications were obtained because the analysis was conducted at the token level. It would have been difficult to obtain such findings and implications using a line-level analysis. We have added the description to this point in Section 9.3.

Section 9.3 Knowledge Gap in Previous Studies

Compared to these prior studies, this research is the first to define micro commits at a fine granularity, specifically at the token level, through empirical analysis. Small commits defined at the line level, which previous studies often used, may overlook important information for improving existing software engineering research. Our research addresses this knowledge gap by conducting the analysis at the token level.

For instance, as explained in Section 7.3, the findings of RQ1 in this study have implications for research in program repair. These findings indicate the need to explore approaches for fixing bugs caused by a single name or literal token. This is because existing automated program repair approaches [20, 26, 30] may not be effective in such scenarios due to a lack of information to repair the code. These findings and implications were obtained because the analysis was conducted at the token level. It would have been difficult to obtain such findings and implications using a line-level analysis. The novelty of this study lies in conducting the analysis at the token level and providing these implications. The details of our findings and implications can be found in Sections 5, 6 and 7.

R3.4 4) Concerning the manual coding, some details could be added to improve the reproducibility of the work. For instance, how was the discussion performed? What is the initial agreement and the final agreement? How was the analysis performed?

Response: Thank you for pointing out this unclear description. We have updated the description of our manual annotation to clarify the process. In the revised manuscript, Fleiss' Kappa was used to compute the first, second, and third agreements (Table 5). Please refer to the updated Section 5.1 below.

Section 5.1 Approach

...
Our manual inspection consisted of two phases: (1) *constructing a coding guide* and (2) *manual classification*. Constructing a coding guide for manual classification/annotation is a common practice in the field of mining software repositories [6, 13, 14, 42, 43, 46]. To create the coding guide, we referred to previous studies [6, 13, 14, 42, 43, 46] and followed

the process detailed below.

The initial coding guide was first discussed by the first and second authors. Since this is the first study to classify micro commits, we examined both micro commits and other types of commits to develop the initial coding guide. After constructing the initial coding guide, we aimed to reach a consensus among the first three authors for this guide and refine the guide. Specifically, we independently annotated 20 micro commits from a subset of all micro commits. This subset consists of micro commits that only change less than or equal to five tokens in main files (.c or .java files) in the Linux, Hadoop, and Zephyr projects to investigate a single operation commit for refining the coding guide. We computed the agreement rate for these 20 micro commits using Fleiss' Kappa [9] that is used to demonstrate inter-rater agreement when there are more than two raters. It is also frequently applied in the field of mining software repositories [6,13]. The Kappa coefficient is commonly interpreted using the following scale [41]: Slight agreement ($0.01 \leq k \leq 0.20$), Fair agreement ($0.21 \leq k \leq 0.40$), Moderate agreement ($0.41 \leq k \leq 0.60$), Substantial agreement ($0.61 \leq k \leq 0.80$), Almost perfect agreement ($0.81 \leq k \leq 0.99$). Then we discussed the coding guide along with any inconsistencies in categorization to reach a consensus. We repeated this process until our categorization substantially matched, indicating that our coding guide was successfully constructed. We, therefore, repeated this process three times (i. e., independently classifying 60 commits). Finally, our agreement rate achieved substantial agreement in two consecutive iterations. Table 5 shows all agreement rates across three authors for each repetition.

Through this process, we identified two perspectives: *operation* and *target*. The operation indicates what kind of operations are applied, such as adding a new statement or changing an expression; the target indicates source code elements where the operation is applied, such as expressions. The details of the coding guide are described below.

We utilized the following coding guide to categorize micro commits in terms of the operations.

- *add:* This refers to operations that add a completely new entity.
- *replace:* This refers to operations that modify an entity.
- *remove:* This refers to operations that completely remove an entity.
- *multi:* This code indicates that multiple operations are applied.
- *no:* This code indicates that no functional change is applied.

We utilized the following coding guide to categorize micro commits in terms of the targets.

- *identifier:* This refers to commits that only modify identifiers, such as variable names. If other entities, such as parentheses, are included, it would not be labelled as an identifier but would be considered an expression.
- *statement:* This refers to commits that modify a complete statement, including the semicolon (;), such as an entire function call with its semicolon. C's #include preprocessor statement is also regarded as a statement.
- *constant:* This refers to commits that only modify literals, such as strings or numbers. If other entities, such as parentheses, are included, they would not be identified as a constant but would be considered an expression.

- *declaration:* This refers to commits that modify declarations, such as variable declarations. However, if the commit can be classified as "identifier" or "constant", it should be categorized under these two categories rather than "declaration".
- *control flow:* This refers to commits that modify the control flow of execution, such as adding a new "else" statement.
- *expression:* If a commit does not match other categories and involves modifying a part of a statement, it would be classified into this category. Additionally, this category includes transformations from constants to variables or vice versa, as well as conversions from a variable to a pointer and vice versa.
- *multi:* This code suggests that operations are performed on multiple targets.
- *no:* This code indicates that no functional change is applied.

Table 6 shows the summary and examples. This represents different types of activities performed by micro commits. Let us describe two example commits. Listing 6 shows an example commit. This commit changes a function call and its argument. More specifically, the identifiers of the function call and the argument value are replaced so that we classify this commit as operations=replace, and targets=identifier. The commit of Listing 7 replaces an expression "++" into "−−". Hence, we classify this commit as operations=replace, and targets=expression.

Because our agreement rates for operation and target based on our coding guide achieved almost perfect and substantial agreement respectively (Table 5) and we made an internal consensus of the coding guide, only the first author manually classified the 400 micro commits, similar to previous studies [13, 14, 42, 43]. All our manual categorizations are available in our sheet[a]. The sample size in manual inspection was determined as a statistical representative with a confidence level of 95% and a confidence interval of 5% for 150,967 micro commits from all studied projects.[b] The confidence interval, also known as the margin of error, indicates the potential percentage difference between the characteristics obtained from the sampled micro commits and those obtained from the population. The minimum sample size with this confidence level and this confidence interval is 383. For safety, we also inspect 17 additional micro commits. Therefore, we classify a total of 400 micro commits.
...

[a] https://docs.google.com/spreadsheets/d/1M6ifKvufH2JV_ZAYcG6j_eEuSeeEjexWavf2eimHyQk/edit?usp=sharing

[b] https://www.surveysystem.com/sscalc.htm

R3.5 5) The Introduction and the Conclusion Sections need to be better structured. Usually, with these two sections, a reader should understand what was done in the study and what results were obtained. In this case, neither section shows the results achieved but only refers to the examples provided in the discussion.

Response: Thank you for your advice. We have revised the conclusion section. Specifically, we have highlighted the significant findings that align with the aim of this study. Additionally, we have emphasized the significant implications.

The introduction section also presents the significant findings that contribute to the aim of this study. These findings are not examples, but rather important results and implications that should be emphasized in the introduction section. We have chosen these findings because a lengthy introduction that includes all results would be challenging to read. However, we agree with the comment regarding the importance of understanding the content of the paper through the introduction section. Therefore, we have revised the introduction to explicitly state that it highlights significant findings, while detailed results can be found in the results section.

Section 1 Introduction

Specifically, we answer the following research questions (RQs). We have also provided a summary of the key findings for each RQ. The detailed results are described in Section 4, Section 5, and Section 6.

Section 10 Conclusion

In this paper, we defined micro commits (add at most five tokens and remove at most five tokens) and investigated their characteristics. This research is the first to define micro commits at a fine granularity, specifically at the token level. The key novelty of this study lies in conducting the analysis at the token level and providing implications for software engineering research.

Below, we present a summary of the findings from our empirical analysis:

- Our defined micro commits account for between 7.45–17.95% of all studied commits. Approximately 1 in 3 or 4 these changes (2.39–4.88% of all studied commits) involve replacing one token with another. Furthermore, RQ3 demonstrates that approximately 90% of One-line commits are micro commits, but only approximately 40–50% of micro commits are One-line commits. In fact, approximately 30–40% of micro commits include two or more hunks.
- The results of RQ1 show that micro commits primarily affect name token types (37.7–44.5%), literal token types (9.2–34.9%), or operator token types (6.6–10.4%). The most frequently affected tokens vary: the period in Java (2.5% in Camel and 3.9% in Hadoop) and the 0/1 in C (1.8 and 1.0% in Linux and 1.3 and 0.9% in Zephyr). Furthermore, the most frequently observed pattern is the modification of a single token. In Java projects, this modification is typically a single literal token. On the other hand, in C projects, the modification is usually a single name token.
- The results of RQ2 indicate that approximately 86% of micro commits involve a single operation on a single target, with the main focus being the replacement of existing targets. The multi-operation micro commits primarily involve changing the order of statements (19.3%).

In the discussion, we presented the following four implications of micro commits on future research based on the findings:

- Based on RQ3, it is observed that almost all One-line commits are micro commits, whereas only 40–50% of the micro commits are One-line commits. Therefore, token-based complexity metrics offer supplementary information to the commonly used line-based complexity metrics. Designing metrics to measure token-based complexity is a potential area for future research.
- Based on the statistics of micro commits, they account for a non-negligible proportion of all studied commits (7.45–17.95%). Additionally, according to Section 7.3, these commits are more likely used to fix bugs. Therefore, supporting the development of micro commits is an important area for future research.

- Based on RQ1, micro commits frequently modify a single token, with the token type often being either a name or a literal. While these micro commits often address bug fixes, suggesting patches to fix individual name or literal tokens can be challenging with existing program repair approaches. Therefore, it is necessary to investigate these micro commits and propose new program repair approaches for future research.
- We define micro commits based on size metrics. However, micro is a general term, and micro commits can be defined not only by size but also by semantic aspects (e. g., tangled commits or not). Exploring semantic-based micro commits is a potential area for future research.

R3.6 6) The Threats to Validity Section needs to be improved. The paper does not discuss mitigation strategies applied during the empirical study. A section that acknowledges the potential limitations of a study but does not provide details on strategies adopted to mitigate those threats or "plans to do so in future studies" does not make clear to the reader the degree of realism or generalizability of the conclusions drawn.

Response: Thank you for your advice. We have revised the threats to validity section. Specifically, we have updated some threats for which the mitigation strategies or future plans are not sufficient. Furthermore, in this stage of major revisions, we have added new threats to this section. For these new threats, we have also included mitigation strategies or future plans.

Section 8.1 External Validity

We conducted our empirical study on four OSS projects. To mitigate the threats to generalizability, we selected OSS projects that are active, popular, and well-known OSS projects written in two popular programming languages. However, even if we use these OSS projects, our results may not be generalized to all projects. Indeed, these are system software. To remedy this challenge, replication studies in research or practical scenarios (e.g., actual projects in the industry) are necessary. Hence, we provide a replication package[a]. Also, the key tool `cregit` is an OSS tool; thus, researchers and practitioners easily convert their Git repositories into token-based ones.

[a] https://github.com/MKmknd/EMSE2024-micro-commits-replication

Section 8.2 Construct Validity

...
Hence, we believe our definition can be acceptable. However, our definition may not be the best; thus, future studies are necessary to find a better definition than our first one. For example, future studies can investigate different thresholds for the number of tokens. Also, studying different thresholds for added and deleted tokens (e. g., 3 added tokens and 5 deleted tokens) can be beneficial. Additionally, they can consider changes to source code comments. This definition would encompass not only maintenance activities related to code logic, but also various other maintenance activities.
...
There are several factors that can influence commits. For instance, the way developers write commits can vary depending on the developer and the project. To reduce these influences, we chose and analyzed four projects that involve a large number of developers. As a result, we expect the impact of such influences has been minimized.

Section 8.3 Internal Validity

To remove comment lines from source code files, we use regular expressions. This process is not perfect and may overlook comment lines. However, our manual analysis in RQ2 observes that our regular expressions usually work well because we do not find false-positive micro commits. Therefore, we can reduce the risks associated with using regular expressions.

Also, another threat exists in our manual analysis (RQ2). In this analysis, we performed manual labeling to micro commits according to our coding guide. Because this process is performed manually by the first author, the result may have false-positive and false-negative results. Therefore, we have made all labels publicly available to facilitate the validation of future studies. Also, to construct the coding guide, the first three authors independently inspected 20 micro commits three times. This process may also include errors. However, our agreement rate achieved substantial agreement in two consecutive iterations. Hence, we believe the coding guide is reliable. An alternative solution is to use an automatic classification approach rather than manual analysis. We developed a heuristic-based method to classify micro commits into their corresponding targets automatically. However, this method does not yield perfect results. To facilitate replication of this approach, we have included it in our replication package. Finally, we randomly sampled 400 micro commits from all projects. Therefore, our sampled micro commits may be biased by the size of the original projects. To mitigate this risk, we manually inspected additional micro commits from each project. We do not observe significant differences across the projects.

In the discussion, we use keywords to identify the commits related to the corrective maintenance activity as defined by Swanson [40]. While the keyword identification is widely used to categorize commits [18,22,27,28,35,37], it is not perfect [3,25]. To mitigate this threat, we manually review identified commits and estimate their accuracy. Also, there are other sets of maintenance activities that can be used to classify commits, such as the IEEE standard [1]. While we believe the maintenance activities defined by Swanson are acceptable, future studies are necessary to use other sets. Also, if commit messages do not contain any keywords, we exclude those commits from the analysis. However, it is possible that these commits are related to maintenance activities. Using more precise methods would enhance the validity of this analysis.

The tool "cregit" used to tokenize the source code files utilizes srcML. Therefore, our analysis can only be applied to specific versions of Java (Java SE8 Edition) and C (up to C11) that are supported by srcML. We can find the supported versions on the official homepage[a]. To extend our analysis to different versions of Java and C, it is necessary to update srcML and apply our analysis to those versions.

...

[a] `https://www.srcml.org/#home`

References

[1] A. Alali, H. Kagdi, and J. I. Maletic. What's a typical commit? a characterization of open source software repositories. In *Proceedings of the 16th IEEE International Conference on Program Comprehension (ICPC)*, pages 182–191. IEEE, 2008.

[2] H. Hata, C. Treude, R. G. Kula, and T. Ishio. 9.6 million links in source code comments: purpose, evolution, and decay. In *Proceedings of the 41st International Conference on Software Engineering (ICSE2019)*, pages 1211–1221, 2019.

[3] H. Hata, N. Novielli, S. Baltes, R. G. Kula, and C. Treude. Github discussions: An exploratory study of early adoption. *Empirical Software Engineering*, 27:1–32, 2022.

[4] L. P. Hattori and M. Lanza. On the nature of commits. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, page III–63–III–71, 2008.

[5] R. Purushothaman and D. E. Perry. Toward understanding the rhetoric of small source code changes. *IEEE Transactions on Software Engineering*, 31(6):511–526, 2005.

[6] E. B. Swanson. The dimensions of maintenance. In *Proceedings of the 2nd International Conference on Software Engineering (ICSE)*, pages 492–497, 1976.

[7] D. Wang, T. Xiao, P. Thongtanunam, R. G. Kula, and K. Matsumoto. Understanding shared links and their intentions to meet information needs in modern code review. *Empirical Software Engineering*, 26, 2021.

[8] D. Wang, M. Kondo, Y. Kamei, R. G. Kula, and N. Ubayashi. When conversations turn into work: a taxonomy of converted discussions and issues in github. *Empirical Software Engineering*, 28(6): 138, 2023.

Kind regards,
The Authors

Title Page containing ALL Author Contact Info.

Dear Editor of the Empirical Software Engineering Journal,

On behalf of Daniel M. German, Yasutaka Kamei, Naoyasu Ubayashi, Osamu Mizuno and myself, I would like to submit our paper titled "An Empirical Study of Token-based Micro Commits" to the journal first paper track of the Empirical Software Engineering Journal.

Contact Information

The contact information of the authors is as follows:

Masanari Kondo, Principles of Software engineering and programming Languages Lab. (POSL) Kyushu University, Japan
E-mail: kondo@ait.kyushu-u.ac.jp

Daniel M. German, Department of Computer Science, University of Victoria, Canada
E-mail: dmg@uvic.ca

Yasutaka Kamei, Principles of Software engineering and programming Languages Lab. (POSL) Kyushu University, Japan
E-mail: kamei@ait.kyushu-u.ac.jp

Naoyasu Ubayashi, Principles of Software engineering and programming Languages Lab. (POSL) Kyushu University, Japan
E-mail: ubayashi@ait.kyushu-u.ac.jp

Osamu Mizuno, Software Engineering Laboratory (SEL) Kyoto Institute of Technology, Japan
E-mail: o-mizuno@kit.ac.jp

Please let me know if more information is needed for this submission.

Best Regards,

Masanari Kondo
April 25, 2023

Noname manuscript No.
(will be inserted by the editor)

An Empirical Study of Token-based Micro Commits

Masanari Kondo · Daniel M. German · Yasutaka Kamei · Naoyasu Ubayashi · Osamu Mizuno

the date of receipt and acceptance should be inserted later

Abstract In software development, developers frequently apply maintenance activities to the source code that change a few lines by a single commit. A good understanding of the characteristics of such small changes can support quality assurance approaches (e.g., automated program repair), as it is likely that small changes are addressing deficiencies in other changes; thus, understanding the reasons for creating small changes can help understand the types of errors introduced. Eventually, these reasons and the types of errors can be used to enhance quality assurance approaches for improving code quality. While prior studies used code churns to characterize and investigate the small changes, such a definition has a critical limitation. Specifically, it loses the information of changed tokens in a line. For example, this definition fails to distinguish the following two one-line changes: (1) changing a string literal to fix a displayed message and (2) changing a function call and adding a new parameter.

Masanari Kondo, Naoyasu Ubayashi
Principles of Software Languages group (POSL)
Kyushu University, Japan
E-mail: {kondo, ubayashi}@ait.kyushu-u.ac.jp

Daniel M. German
Department of Computer Science, University of Victoria,
Victoria, BC, Canada
E-mail: dmg@uvic.ca

Yasutaka Kamei
Principles of Software Languages group (POSL)
Kyushu University, Japan, and
Inamori Research Institute for Science, Kyoto, Japan
E-mail: kamei@ait.kyushu-u.ac.jp

Osamu Mizuno
Software Engineering Laboratory (SEL)
Kyoto Institute of Technology, Japan
E-mail: o-mizuno@kit.ac.jp

These are definitely maintenance activities, but we deduce that researchers and practitioners are interested in supporting the latter change. To address this limitation, in this paper, we define *micro commits*, a type of small change based on changed tokens. Our goal is to quantify small changes using changed tokens. Changed tokens allow us to identify small changes more precisely. In fact, this token-level definition can distinguish the above example. We investigate defined micro commits in four OSS projects and understand their characteristics as the first empirical study on token-based micro commits. We find that micro commits mainly replace a single name or literal token, and micro commits are more likely used to fix bugs. Additionally, we propose the use of token-based information to support software engineering approaches in which very small changes significantly affect their effectiveness.

Keywords Empirical Study; Micro Commits; Tokens; Mining Software Repositories

1 Introduction

Commits changing a few lines of code are common in software development. Purushothaman et al. defined *small commits* as those modifying less than 10 lines in their study [37]. They found that 50% of changes in the examined systems were small commits. They also reported that 10% of all commits were *one-line commits* (modified at most one line). In a recent study, Alali et al. reported that in the GCC project, 19.9% of commits were *extra-small*, adding at most 5 lines of code [2]. Our research found that in the projects we studied, between 6 and 8% of all commits were one-line commits (see Section 2).

Improving code quality is an ultimate goal for software engineering researchers, and several quality assurance approaches have been widely studied so far, such as automated program repair (APR), defect prediction, and fault localization. A good understanding of the characteristics of very small changes can support such approaches, as it is likely that such changes are addressing deficiencies in the system [37]; thus, understanding the characteristics of creating small changes can help understand the types of errors that other changes introduce and potentially help with program repair. Eventually, the information can be used to enhance quality assurance approaches for improving code quality.

While prior studies [2, 18, 37] use churn (number of lines added and removed) to identify small changes (e. g., small commits and one-line commits [37], or extra-small commits [2]), it has one significant limitation: they consider the line to be the finest-grained entity of changed source code. More specifically, such a definition overlooks the details of what has changed in a line [11, 34, 38]. For instance, when several lines have a small change (such as an identifier being renamed in a few places), these modifications might appear as one line added and one line removed for each change, rather than a single identifier change.

Another problem is that splitting or joining a line of code that is being modified can result in noise. For example, splitting a line into two would be reflected as a change to multiple lines in version control systems (e. g., Git), and this type of change can add noise to the analysis of the history of the development process.

These limitations sometimes cause researchers to fail in accurately quantifying small changes. For example, Listing 2 shows a commit in the Linux repository that changes a few lines (i. e., three added and two deleted lines). While this commit corresponds with a multiple-line change and may not correspond to a one-line commit, it only adds a token "static". This is similar to Listing 1 corresponding with a one-line commit that only adds a token "static". Studying the actual changed tokens instead of the lines can provide a better understanding of the characteristics of the small changes.

In this paper, we define a new class of commits: *micro commits*. **Micro commits are commits that add at most five tokens and remove at most five tokens of source code**. We aim to quantify small changes using the token-level definition (i. e., micro commits) rather than relying on the line-level definition (i. e., one-line commits). This token-level definition allows us to identify small changes more accurately, and use token information to characterize them. We conducted an empirical study on four large, mature open-source projects to: a) demonstrate that micro commits are common, accounting for between 7.45 and 17.95% of all studied commits in the studied projects, b) understand their qualitative and quantitative characteristics, and c) show our definition of micro commits (a threshold of 5 added and removed tokens) includes approximately 90% of all one-line commits, yet only approximately 40–50% of micro commits are one-line commits.

Specifically, we answer the following research questions (RQs). We have also provided a summary of the key findings for each RQ. The detailed results are described in Section 4, Section 5, and Section 6.

RQ1: **What are the characteristics of micro commits?**
Motivation: This research question aims to explain their quantitative characteristics: how frequent they are, and the types of tokens they delete and add.
Results: Most micro commits replace a single token with one of the same types, and this token type is mostly name (e. g., identifier names) or literal (e. g., numbers). Java and C differ on the most frequent tokens in micro commits.

RQ2: **What are the types of changes that micro commits perform?**
Motivation: We intend to understand the purpose of micro commits (e. g., changing control flow, replacing the name of a variable, and modifying an expression) and whether a micro commit performs one or more activities. Specifically, we manually inspected the changes applied to the source code to understand the purpose behind the micro commit and the occurrence of activities.
Results: More than 85% of micro commits apply a single operation to a single target. The four most common types of these micro commits are replacing an existing expression, identifier, constant, or declaration. Multi-operation micro commits usually change the order of statements.

RQ3: **How do micro commits compare to one-line commits?**
Motivation: This research question aims to explore the extent of differences between one-line commits and micro commits. Extracting micro commits requires syntactic parsing of the source code, which is more costly than extracting one-line commits. If they are identical, micro commits may be redundant.
Results: Most one-line commits are micro commits (approximately 89–93%).

Table 1: The proportion of one-line commits in the studied projects

Project	#total commits	#studied commits	#one-line commits	Proportion(%)
Camel	60,911	38,458	2,405	6.25
Hadoop	69,997	53,796	2,302	4.28
Linux	1,048,688	802,726	65,858	8.20
Zephyr	40,883	25,542	1,979	7.75

In contrast, only about 40–50% of micro commits are one-line commits. Indeed, 30–40% of micro commits include two or more hunks (one-line commits only have one hunk).

The main contributions of this paper are as follows:

- We propose the concept of micro commits as commits that add at most five tokens and remove at most five tokens, and demonstrate that these types of changes are common.
- We empirically investigate micro commits and understand their quantitative and qualitative characteristics. We especially shed light on the differences in micro commits between programing languages through our manual inspection.
- We propose the use of token-level information to support software engineering approaches that use extremely small changes (e. g., programing repair).
- We provide the replication package of this study that contains a set of micro commits that have been manually labeled according to their purpose.

The organization of our paper is as follows: Section 2 introduces motivating examples. Section 3 explains our studied dataset. Section 4, Section 5, and Section 6 present the experiments and results based on our RQs. Section 7 proposes the use of token-level information. Section 8 describes the threats to the validity of our case study. Section 9 introduces related work. Section 10 presents the conclusion.

2 Motivating Example

In this section, we provide an example of a one-line commit. Also, we demonstrate that they account for a non-negligible proportion of commits. Finally, we highlight the drawback of using lines of code to study extremely small changes, and we discuss how micro commits can address this drawback.

We first show the frequency of one-line commits in four OSS projects used in this study and confirm that it is consistent with [37]. As in [37], we use the diffs generated by Git to identify one-line commits. Table 1 shows the proportion of one-line commits. The proportion was computed by using the "#studied commits" column. It only shows the commits that have made changes to the source code. Our analysis is conducted based on these commits. The detailed procedure for extracting commits is explained in Section 3.2 and Section 3.3[1]. We observe 4.28–8.20% of one-line com-

[1] Because of the differences in source code management tools, one-line changes in the prior study [37] and our one-line commits are slightly different.

Listing 1: An example micro commit in Linux retrieved from: 092734b4bb227faddf241b116af14357645d963c

```
@@ -385 +385 @@ EXPORT_SYMBOL(bt878_device_control);
-struct cards card_list[] __devinitdata = {
+static struct cards card_list[] __devinitdata = {
```

Listing 2: An example micro commit with multiple changed lines in Linux retrieved from: 0ce6e62bd6591777bd92873e2db93fdbc5228122

```
@@ -1143,2 +1143,3 @@ int path_lookup_open(const char *name, unsigned int lookup_flags,
-int path_lookup_create(const char *name, unsigned int lookup_flags,
-                struct nameidata *nd, int open_flags, int create_mode)
+static int path_lookup_create(const char *name, unsigned int lookup_flags,
+                              struct nameidata *nd, int open_flags,
+                              int create_mode)
```

mits. Specifically, the proportion is more than 7% in the Linux and Zephyr projects. Hence, one-line commits account for a non-negligible proportion of all commits.

Listing 1 shows an example of a one-line commit, also known as a micro commit. This commit adds a static modifier into a struct definition, and this is not adding a functionality but fixing the code.

However, some extremely small changes are often obscured by splitting or joining lines of code, making them appear more complex than they are. For example, Listing 2 shows an example of a micro commit that is not a one-line commit. This commit semantically adds a static modifier only; however, this commit includes multiple changed lines because of changing the format of the definition of the variable. Listings 1 and 2 are semantically identical, but one-line commits cannot include Listing 2 because it modifies multiple lines. Because we used Git, we deduced that the diff algorithms could address this limitation. Git has four algorithms to compute diffs, and they exhibit different results [36]. Hence, we investigated four algorithms: *patience*, *minimal*, *histogram*, and *myers* described in the Git manual page[2]. However, all algorithms generate the same diff. Hence, one-line commits may overlook such commits. If these diffs are analyzed with finer-grained source code entities (e. g., AST), it is easy to realize these commits have the same intention (i. e. perform the same change). However, AST analysis is expensive, particularly in repositories such as Linux that has more than one million commits and more than 60k source code files.

Therefore, to address this limitation, we define *micro commits* based on tokens. Because tokens are the semantically finest-grained source code entity, micro commits based on tokens can cover ones overlooked by one-line commits. Indeed, Listings 1 and 2 change one token only; thus, they both perform the same change in two different lines of code.

[2] https://git-scm.com/docs/git-diff

3 Dataset Preparation

3.1 Studied Datasets

To answer our RQs, we conducted an empirical study on four notable large OSS projects written in Java and C: Camel[3], Hadoop[4], Linux[5], and Zephyr[6]. The Camel project is an integration framework that provides a routing engine to integrate systems. The Hadoop project is a distributed computing framework. The Linux project (a.k.a. the Linux Kernel) is one of the most popular open-source operating system kernels. The Zephyr project is a real-time operating system supporting several architectures. Hence, these include four software systems: an integration framework, a distributed computing framework, an operating system kernel, and an operating system. We selected these four projects because of three reasons: (1) they are written in popular programming languages (i.e., Java and C), (2) they are well-known popular OSS projects, and (3) they have a long development history.

3.2 One-line Commits and Micro Commits

Our research aims to accurately quantify small changes using a token-level definition (i. e., micro commits). Additionally, to highlight the differences in accuracy between token-level and line-level definitions, we should compare micro commits with one-line commits. Therefore, we detail the process of extracting one-line commits and micro commits from software development histories below.

Git is language agnostic. The changes performed in a commit are displayed as a diff, comparing the code before and after the commit. These changes are grouped into *hunks*. A hunk is a set of contiguous lines that are added/removed/modified together, along with metadata that indicates its context—where the change occurred. Each hunk can include *context* lines (i. e., lines that were not modified but are used to help interpret the change). The default number of context lines is three, but for the purpose of this paper, we have set it to zero; thus, we ignore context lines in the hunk. Git's diff does not present lines that have been modified. Instead, it simply records lines that have been removed (prefixed with "-") and lines that have been added (prefixed with "+"); thus, a modified line is represented by a removed line and its corresponding added line. If several continuous lines are modified simulataneously, Git presents first all removed lines, and thereafter the added lines.

We extracted one-line commits based on the hunks provided by Git. Specifically, one-line commits correspond to commits that have a diff with exactly one removed and one added line **in the same hunk**. Listing 1 is an example of such a commit.

To be able to perform token-level analysis, we processed the repository history using `cregit` [11]. This uses srcML [7] to generate an equivalent commit history where the differences are displayed as changes to sequences of tokens instead of

[3] https://camel.apache.org/
[4] https://hadoop.apache.org/
[5] https://www.linux.org/
[6] https://www.zephyrproject.org/

Listing 3: An example commit in a line repository

```
@@ -10 +10 @@ test();
-int flg = 10;
+static int flag = 10;
```

Listing 4: An example commit in a token repository

```
@@ -100,2 +100,3 @@ test();
+specifier|static
 name|int
-name|flg
+name|flag
```

lines (see [11] for a detailed description). Effectively, we track tokens removed and/or added during a commit and can easily identify commits that have added and/or removed a certain number of tokens. Similarly to the way we can identify modified lines, we can identify modified tokens if one token is added and another is removed in the same hunk. For example, the commit from Listing 3 is shown in its equivalent token version in Listing 4.

We extracted micro commits based on the hunks provided by Git repositories processed by `cregit`. Micro commits refer to commits that include a maximum of five added tokens and five deleted tokens across all hunks. This number was chosen for the following reasons.

– In the languages being studied (C and Java), it is highly unlikely to add a new statement with only five tokens, suggesting that such commits carry out minor modifications. For example, within five tokens, developers can only add a function call with one parameter and an ending semicolon: `name(parm);` includes two identifiers, two parentheses, and one semicolon.
– In the systems we studied, between 7.45 and 17.95% of all studied commits add at most 5 tokens and remove at most 5 tokens.

This number serves as a parameter for micro commits. For example, we use the same number for both added and deleted tokens while different numbers could be used. Its potential threats are discussed in Section 8.2.

Source code comments are important for source code and making changes to comments are also maintenance activities. However, in this paper, we exclude comments and execute our analysis. The reason is to prioritize maintenance activities for code logic. As mentioned in Section 1, our intention is to support various software engineering approaches (e. g., defect prediction), which typically prioritize code logic over comments. Indeed, defect prediction studies typically do not take into account comment issues when identifying target defects [19, 24, 33]. While we acknowledge the importance of changes made to comments for maintenance purposes, this perspective is beyond the scope of our paper.

3.3 Data Collection

We preprocessed the commits in the studied repositories and constructed a database with its diffs (both line-based and token-based) using the following steps. From this database, we extracted one-line commits and micro commits.

Step 1: For each commit, extract the line-based diff of its modified source code ignoring any changes to non-source code:
- Remove changes to non-source files[7].
- Remove changes to comments and white space using regular expressions (e. g., "//.*").
- Remove commits that do not have any changes after the aforementioned processes.

Step 2: Using `cregit`, for each commit, extract the token-based diff of its modified source code ignoring any changes to non-source code:
- Remove changes to comments.
- For each source code token, keep its type and its value. `cregit` tokenizes the source code using srcML[8]. Thus, the types of tokens are those created by srcML. For example, `int i;` will be converted to the sequence of *type|value*: `name|int, name|i, decl_stml |;`

Step 3: Create a database in SQLite with these commits (line and token-based) including:
- Identify and store each hunk and its metadata (such as the file where it occurred and the number of lines/tokens added and removed).
- Added and removed lines or tokens in each hunk
- Commit messages
- Metadata (e. g., index)

In summary, we record for each line-based diff: its commit id and its set of hunks (for each hunk, its location, number of lines added, number of lines removed, and its contents as a sequence of added/removed lines). We record the same for token-based diffs (replacing lines with tokens–including their types). Note that when obtaining diffs with Git, we use the *myers* algorithm, which is the default algorithm. Also, we record commit messages. More details can be found in our replication package (see Section 8.1)

Table 2 displays the number of extracted micro commits and one-line commits. We used these micro commits and one-line commits in this study. We found that micro commits can cover approximately 90% of one-line commits. In contract, only approximately 40% (for Linux and Zephyr) or 50% (for Camel and Hadoop) of micro commits can be covered by one-line commits.

As shown in Table 3, between 7.45% and 17.95% of all studied commits are micro commits, and approximately 1 in 3 or 4 micro commits are one-token commits in all projects. Hence, micro commits constitute a non-negligible portion of all studied

[7] We extract files with the extension of "java" in Camel and Hadoop and with the extension of "c" in Linux and Zephyr.

[8] `https://www.srcml.org/`

Table 2: The number and proportion of the intersection between one-line commits and micro commits in each commit type (i. e., one-line or micro). The column of "#intersects" indicates the intersection; the columns of "#One-line" and "#Micro" indicate the number of one-line commits and micro commits; the column of "%One-line" and "%Micro" indicate the proportion of intersection in each commit type (i. e., one-line commits and micro commits).

Project	#intersects	#One-line	#Micro	%One-line	%Micro
Camel	2,131	2,405	4,230	88.6	50.4
Hadoop	2,069	2,302	4,010	89.9	51.6
Linux	59,836	65,858	138,142	90.9	43.3
Zephyr	1,849	1,979	4,585	93.4	40.3

Table 3: Number of micro commits and one-token commits and their proportion with respect to all source-code commits.

Project	Micro commits	Prop (%)	One-token commits	Prop (%)
Camel	4,230	11.00	1,319	3.43
Hadoop	4,010	7.45	1,288	2.39
Linux	138,142	17.21	32,973	4.11
Zephyr	4,585	17.95	1,247	4.88

commits. As expected, most of these commits modify a few lines: between 52.80% and 58.70% modify add or remove at most one line, and between 59.56% and 67.48% add-or-remove two lines.

4 RQ1: What are the characteristics of micro commits?

4.1 Approach

The goal of RQ1 is to understand the characteristics of micro commits. More specifically, we investigated the modified tokens.

In this RQ, we investigated micro commits from two perspectives: (1) *most frequently modified tokens and token types by micro commits* and (2) *modification patterns for each micro commit*. We first count added and removed tokens and their token types from all micro commits and provide researchers with tokens and token types frequently modified by micro commits. Second, we investigate the set of added and removed tokens for every micro commit and show the common *modification patterns* adopted by a single micro commit. Note that we used the set rather than the sequence of tokens. Hence, we characterized modification patterns based on modified tokens in micro commits rather than the sequences of modified tokens.

We use srcML classification for the types of tokens. For example, tokens of type names correspond to names of types and variables (including language predefined ones); literals are constant values; operators are operators to perform mathematical operations; argument_list corresponds to either () (empty parameter list), or each of

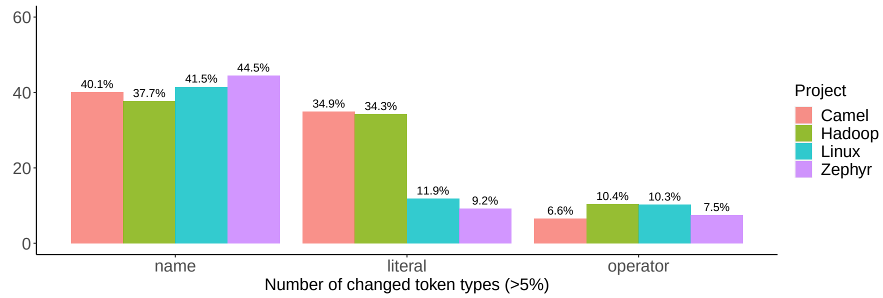


Fig. 1: Proportions of changed token types (> 5%)

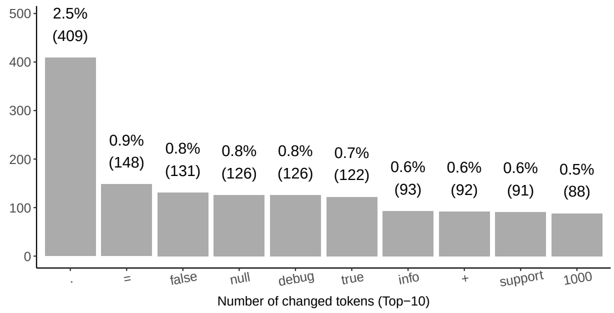
the parenthesis around parameters or the comma that separates them; expr_stmt is the semicolon at the end of the statement; block is a { or }; file is a filename; specifier is a C storage specifier (e. g., `static`); directive a C preprocessor directive; and annotation corresponds to Java annotations. The right-hand side of C macro definitions is not further parsed by srcML and is considered a single token of type value (i. e., the value the macro expands to).

4.2 Results

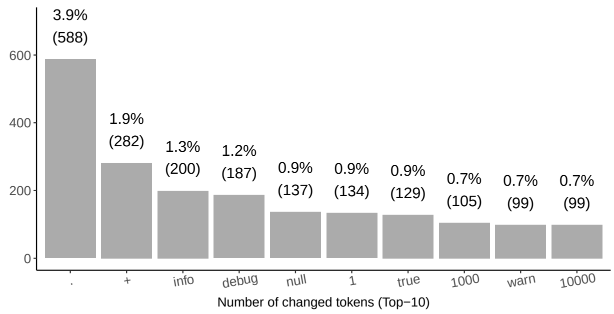
(1) Most frequently modified tokens and token types by micro commits

The top-3 most frequently touched token types in micro commits are generally the name, literal, and operator token types. Figure 1 shows the frequently added/removed token types by micro commits that account for more than 5% in all projects. We found three token types, which we refer to as the top-3 most frequently touched token types. The token type most frequently included in micro commits is the name token (e.g., the name of a variable or function), the second one is the literal token (e.g., 123, 'a', "test"), and the third one is the operator token (such as +). Also, the proportion of the literal token is significantly different between Java and C. Specifically, while the proportion of name tokens is more than three times larger than that of literal tokens in the projects written in C, the difference is less than two times in the projects written in Java. While the proportion of the operator tokens is relatively small, these are also included in the top-3 most frequently touched token types. Hence, micro commits usually modify name, literal, and operator tokens in most cases, but their proportions may differ between programming languages and their token types.

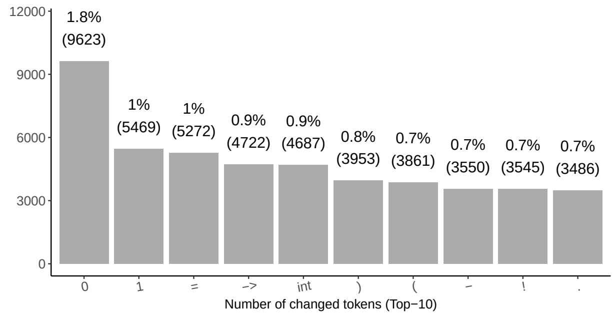
While the tokens corresponding to the top-3 token types differ between Java and C, we observe similar tokens within the same language. Figure 2 shows the top-10 most frequently occurring tokens for the top-3 token types. In Java, boolean literals (e. g., true/false, null), and numeric literals were the most commonly observed, while in C, they were the tokens for 0/1, parentheses and names for types (e. g., int, u32_t, and u8_t).



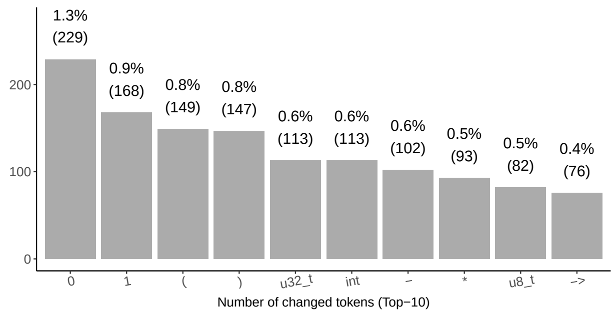
(a) Camel (Java)



(b) Hadoop (Java)



(c) Linux (C)



(d) Zephyer (C)

Fig. 2: Numbers of changed tokens

Table 4: Top-5 added and removed token types applied to source code by micro commits

Project	Add	Remove	n	Pro
Camel (Java)	literal	literal	833	19.7
	name	name	435	10.3
	literal,literal	literal,literal	312	7.4
	name,name	name,name	269	6.4
	name,name,name	name,name,name	125	3.0
Hadoop (Java)	literal	literal	761	19.0
	name	name	408	10.2
	literal,literal	literal,literal	266	6.6
	name,name	name,name	229	5.7
	specifier	-	126	3.1
Linux (C)	name	name	13693	9.9
	literal	literal	7350	5.3
	value	value	6835	4.9
	name,name	name,name	6141	4.4
	specifier	-	4171	3.0
Zephyr (C)	name	name	487	10.6
	value	value	352	7.7
	name,name	name,name	260	5.7
	name,name,name	name,name,name	170	3.7
	literal	literal	169	3.7

Listing 5: An example single token modification in a token repository

```
@@ -100,1 +100,1 @@ test();
-name|flg
+name|flag
```

In conclusion, the types of tokens most frequently changed are the same in both programming languages, but the actual tokens are different.

(2) Modification patterns of micro commits

The single token modification is the most frequently observed pattern in the studied micro commits. Table 4 shows the top-5 most frequently appearing sets of removed and added tokens in micro commits. Each row indicates a set of token types modified by a single micro commit and their frequency and proportion (i. e., # of micro commits). The “n” column indicates the frequency, while “Pro” indicates the proportion. In this paper, we use the same column name in the other tables. In all projects, the most frequently observed micro commits consist of an added and removed token. For example, in the Linux project, micro commits adding and removing a name token are the most frequently observed. This type of single addition and removal usually represents a single token being replaced (e. g., Listing 5) and is the most frequently observed type of micro commit in all projects.

Similar to the results from (1), the modified tokens differ between Java and C. Modifications of literals are the most frequent pattern in Java, accounting for approximately 20% of all micro commits. Modifications of names are the most common pattern in C, accounting for about 10% of all micro commits.

Summary of RQ1

Most micro commits modify a single token, and this token type is either a name, a literal, or an operator. The distribution of micro commits of each of these types is different in C and Java. The operators being replaced are also significantly different across languages.

5 RQ2: What are the types of changes that micro commits perform?

5.1 Approach

This RQ aims to understand the details of the activities performed by micro commits. Specifically, we manually inspect a large set of micro commits to understand what types of change were performed from a source code perspective, considering removed and/or added tokens. Such an understanding gives us insight into whether understanding micro commits can support several approaches in software engineering (see the details in Section 7).

Our manual inspection consisted of two phases: (1) *constructing a coding guide* and (2) *manual classification*. Constructing a coding guide for manual classification/annotation is a common practice in the field of mining software repositories [6, 13, 14, 42, 43, 46]. To create the coding guide, we referred to previous studies [6, 13, 14, 42, 43, 46] and followed the process detailed below.

The initial coding guide was first discussed by the first and second authors. Since this is the first study to classify micro commits, we examined both micro commits and other types of commits to develop the initial coding guide. After constructing the initial coding guide, we aimed to reach a consensus among the first three authors for this guide and refine the guide. Specifically, we independently annotated 20 micro commits from a subset of all micro commits. This subset consists of micro commits that only change less than or equal to five tokens in main files (.c or .java files) in the Linux, Hadoop, and Zephyr projects to investigate a single operation commit for refining the coding guide. We computed the agreement rate for these 20 micro commits using Fleiss' Kappa [9] that is used to demonstrate inter-rater agreement when there are more than two raters. It is also frequently applied in the field of mining software repositories [6, 13]. The Kappa coefficient is commonly interpreted using the following scale [41]: Slight agreement ($0.01 \leq k \leq 0.20$), Fair agreement ($0.21 \leq k \leq 0.40$), Moderate agreement ($0.41 \leq k \leq 0.60$), Substantial agreement ($0.61 \leq k \leq 0.80$), Almost perfect agreement ($0.81 \leq k \leq 0.99$). Then we discussed the coding guide along with any inconsistencies in categorization to reach a consensus. We repeated this process until our categorization substantially matched, indicating that our coding guide was successfully constructed. We, therefore, repeated this process three times (i. e., independently classifying 60 commits). Finally, our agreement rate achieved substan-

Table 5: Fleiss' Kappa scores for each repetition

Criteria	First Time	Second Time	Third Time
Operations	0.686	0.669	0.832
Targets	0.425	0.671	0.754

tial agreement in two consecutive iterations. Table 5 shows all agreement rates across three authors for each repetition.

Through this process, we identified two perspectives: *operation* and *target*. The operation indicates what kind of operations are applied, such as adding a new statement or changing an expression; the target indicates source code entities where the operation is applied, such as expressions. The details of the coding guide are described below.

We utilized the following coding guide to categorize micro commits in terms of the operations.

- *add:* This refers to operations that add a completely new entity.
- *replace:* This refers to operations that modify an entity.
- *remove:* This refers to operations that completely remove an entity.
- *multi:* This code indicates that multiple operations are applied.
- *no:* This code indicates that no functional change is applied.

We utilized the following coding guide to categorize micro commits in terms of the targets.

- *identifier:* This refers to commits that only modify identifiers, such as variable names. If other entities, such as parentheses, are included, it would not be labelled as an identifier but would be considered an expression.
- *statement:* This refers to commits that modify a complete statement, including the semicolon (;), such as an entire function call with its semicolon. C's #include preprocessor statement is also regarded as a statement.
- *constant:* This refers to commits that only modify literals, such as strings or numbers. If other entities, such as parentheses, are included, they would not be identified as a constant but would be considered an expression.
- *declaration:* This refers to commits that modify declarations, such as variable declarations. However, if the commit can be classified as "identifier" or "constant", it should be categorized under these two categories rather than "declaration".
- *control flow:* This refers to commits that modify the control flow of execution, such as adding a new "else" statement.
- *expression:* If a commit does not match other categories and involves modifying a part of a statement, it would be classified into this category. Additionally, this category includes transformations from constants to variables or vice versa, as well as conversions from a variable to a pointer and vice versa.
- *multi:* This code suggests that operations are performed on multiple targets.
- *no:* This code indicates that no functional change is applied.

Table 6: The description of each candidate in our manual inspection

Criteria	Candidate	Description	Example Commits and Their Diffs in Linux
Operations	add	Add a new entity	`122503683169b21d9cdb90380a20caad7ba994b6` Diff: Listing 11
	replace	Replace an existing entity	`b7a90e8043e7ab1922126e1c1c5c004b470f9e2a` Diff: Listing 12
	remove	Remove a completely existing entity	`b95b4e1ed92a203f4bdfc55f53d6e9c2773e3b6d` Diff: Listing 13
	multi	Operations on multiple targets	`8df0cfe6c6c4a9355989baa8de9f166b2bc51f76` Diff: Listing 8
	no	Non-functional modification	`a092532483e3200a53c8b1170b3988cc668c07ef` Diff: Listing 14
Targets	declaration	Change in a type signature	`36f062042b0fd9f8e41b97a472f52139886ca26f` Diff: Listing 15
	constant	A constant (e. g., literal)	`1db76c14d215c8b26024dd532de3dcaf66ea30f7` Diff: Listing 16
	identifier	An identifier (e. g., function calls)	`70e8b40176c75d3544024e7c934720b11a8a11bf` Diff: Listing 17
	control flow	Modifies the control flow	`415a1975923722f729211a9efca550c60c519bf3` Diff: Listing 18
	statement	A the majority of a statement (delimited by semicolon)	`b95b4e1ed92a203f4bdfc55f53d6e9c2773e3b6d` Diff: Listing 13
	expression	A part of a statement and not classified into other categories	`40cc394be1aa18848b8757e03bd8ed23281f572e` Diff: Listing 19
	multi	Multiple targets are altered	`8df0cfe6c6c4a9355989baa8de9f166b2bc51f76` Diff: Listing 8
	no	Non-functional modification	`a092532483e3200a53c8b1170b3988cc668c07ef` Diff: Listing 14

Table 6 shows the summary and examples. This represents different types of activities performed by micro commits. Let us describe two example commits. Listing 6 shows an example commit. This commit changes a function call and its argument. More specifically, the identifiers of the function call and the argument value are replaced so that we classify this commit as operations=replace, and targets=identifier. The commit of Listing 7 replaces an expression "++" into "−−". Hence, we classify this commit as operations=replace, and targets=expression.

Because our agreement rates for operation and target based on our coding guide achieved almost perfect and substantial agreement respectively (Table 5) and we

Listing 6: Example "replace identifier" commit diff retrieved from `f72e6c3e17be568138d8e4855ac2734d251a6913` in Linux.

```
-        strlcpy(drvinfo->bus_info, pci_name(mdev->pdev),
+        strlcpy(drvinfo->bus_info, dev_name(mdev->device),
```

Listing 7: Example "replace expression" commit diff retrieved from `8b58f261113c442717b9d205ab187e51c3823597` in Linux.

```
-        dqm->total_queue_count++;
+        dqm->total_queue_count--;
```

Table 7: Proportion of micro commits having multi activities

Single	Multi
85.75%(343)	14.25%(57)

made an internal consensus of the coding guide, only the first author manually classified the 400 micro commits, similar to previous studies [13, 14, 42, 43]. All manual categorizations are available in our sheet[9]. The sample size in manual inspection was determined as a statistical representative with a confidence level of 95% and a confidence interval of 5% for 150,967 micro commits from all studied projects.[10] The confidence interval, also known as the margin of error, indicates the potential percentage difference between the characteristics obtained from the sampled micro commits and those obtained from the population. The minimum sample size with this confidence level and this confidence interval is 383. For safety, we also inspect 17 additional micro commits. Therefore, we classify a total of 400 micro commits.

5.2 Results

Micro commits usually perform a single operation (85.75%). Micro commits with multiple operations account for the remaining 14.25%, and they usually correspond to two operations. Table 7 lists the proportion of micro commits classified into "multi" in the operation and target. Approximately 86% of micro commits are classified into single operations (i. e., non "multi"); thus, micro commits usually modify an extremely small section. We refer to such commits as *single-operation micro commits* (e. g., Listing 5 is a replace identifier). We surprisingly observe that approximately 14% are classified into *multi-operation micro commits* (i. e., "multi"). Hence, multi-operation micro commits account for a non-negligible portion of the

[9] https://docs.google.com/spreadsheets/d/1M6ifKvufH2JV_ZAYcG6j_eEuSeeEjexWavf2eimHyQk/edit?usp=sharing
[10] https://www.surveysystem.com/sscalc.htm

Table 8: The frequency of the combination of operations and targets in single-operation micro commits

Operation	Target	n	Pro
replace	expression	85	24.8
replace	identifier	69	20.1
replace	constant	59	17.2
replace	declaration	57	16.6
add	statement	22	6.4
replace	control flow	12	3.5
no	no	8	2.3
remove	statement	8	2.3
add	expression	7	2.0
remove	declaration	7	2.0
add	control flow	4	1.2
add	identifier	2	0.6
remove	control flow	1	0.3
remove	expression	1	0.3
remove	identifier	1	0.3

Listing 8: Example "multi" micro commit diffs retrieved from `8df0cfe6c6c4a9355989baa8de9f166b2bc51f76` in Linux.

```
@@ -111,0 +112,5 @@
+ * - EXTCON_PROP_USB_SS (SuperSpeed)
+ * @type:          integer (intval)
+ * @value:         0 (USB/USB2) or 1 (USB3)
+ * @default:       0 (USB/USB2)
+ *
@@ -114,0 +120 @@
+#define EXTCON_PROP_USB_SS              2
@@ -117 +123 @@
-#define EXTCON_PROP_USB_MAX             1
+#define EXTCON_PROP_USB_MAX             2
```

micro commits. Listing 8 shows an example of a multi-operation micro commit. This commit has an add declaration and a replace constant. We disregard the first hunk because it only contains comment lines. It should be noted that such multi-operation micro commits usually have two operations only; we observed that only two commits contain more than two operations. Thus, based on our manual inspection, it is rare to find micro commits containing more than two operations.

Approximately 82.2% of single-operation micro commits replace existing tokens. Table 8 summarizes the frequency of the combination of operations and targets in single-operation micro commits. The top-4 combinations include the "replace" operation, accounting for approximately 78.7% (270/343). Also, all the "replace" operation commits account for 82.2% (282/343). This result suggests that many single-operation micro commits modify the existing source code, but do not add or remove the source code.

Table 9: The frequency of the combination of operations and targets in multi-operation micro commits

Operation	Target	n	Pro
replace	identifier	22	19.1
replace	expression	19	16.5
add	statement	16	13.9
remove	statement	16	13.9
replace	constant	15	13.0
add	expression	7	6.1
replace	declaration	5	4.3
remove	expression	4	3.5
add	declaration	3	2.6
replace	control flow	3	2.6
add	control flow	2	1.7
remove	declaration	2	1.7
remove	control flow	1	0.9

Listing 9: Example "multi" commit diff retrieved from `a71bfb4a6aabfe5e6f145883020153103c7fdfba` in Linux.

```
-error_free_data:
-       free(data);
 error_free_buffer_access:
        free(buffer_access);
+error_free_data:
+       free(data);
```

Multi-operation micro commits more frequently add-and-remove statements rather than single-operation ones. Table 9 summarizes the frequency of the combination of operations and targets in multi-operation micro commits, and Table 10 summarizes the frequency of the pair of their combination for each commit. The main difference from single-operation micro commits is that the "add statement" and the "remove statement" are top-3 (Table 9). The reason is that multi-operation micro commits that add and remove statements appear most frequently (Table 10). This type of commit is used to move the statements, and therefore, change the order of execution and potentially the control flow of the program. For example, Listing 9 shows an example micro commit. This commit adds and removes a statement and changes the order of execution of the free statement to fix a bug related to freeing data. This swapping activity is frequently observed in multi-operation micro commits.

Table 10: The frequency of the pair of the combination of operations and targets in multi-operation micro commits

Ope#1	Tar#1	Ope#2	Tar#2	Ope#3	Tar#3	n	Pro
remove	statement	add	statement	-	-	11	19.3
replace	identifier	replace	expression	-	-	6	10.5
replace	identifier	replace	constant	-	-	6	10.5
add	expression	replace	identifier	-	-	4	7.0
replace	expression	replace	constant	-	-	4	7.0
remove	statement	replace	expression	-	-	3	5.3
add	expression	replace	constant	-	-	2	3.5
add	control flow	replace	expression	-	-	2	3.5
replace	identifier	replace	declaration	-	-	2	3.5
replace	expression	replace	control flow	-	-	2	3.5
add	declaration	remove	declaration	-	-	2	3.5
add	statement	remove	expression	replace	identifier	2	3.5
replace	constant	remove	expression	-	-	1	1.8
add	statement	replace	identifier	-	-	1	1.8
remove	statement	replace	control flow	-	-	1	1.8
add	statement	replace	declaration	-	-	1	1.8
replace	declaration	replace	constant	-	-	1	1.8
add	statement	replace	expression	-	-	1	1.8
add	statement	remove	control flow	-	-	1	1.8
add	declaration	replace	constant	-	-	1	1.8
remove	statement	replace	identifier	-	-	1	1.8
add	expression	remove	expression	-	-	1	1.8
replace	declaration	replace	expression	-	-	1	1.8

Summary of RQ2

More than 85% of micro commits apply a single operation to a single target, and they mainly replace existing target. Multi-operation micro commits frequently change the order of statements.

6 RQ3: How do micro commits compare to one-line commits?

6.1 Approach

As discussed in Sections 1 and 2, one-line commits are common in software development and often address deficiencies in the system [37]. However, they have a drawback: they overlook changes within a line. Consequently, two commits with the same number of changed tokens could differ; one might be a one-line commit and the other might not. The concept of micro commits, introduced in this paper, address their drawbacks.

However, the extent of the differences between one-line commits and micro commits is unclear. Extracting micro commits is more costly than one-line commits as it requires syntactic parsing of the source code. Hence, this RQ aims to compare the one-line commits and micro commits.

Our methodology can be summarized as follows: we start by analyzing the changed tokens in one-line commits. We then analyze the modified hunks in micro commits. Finally, we discuss the intersection between one-line commits and micro commits.

6.2 Results

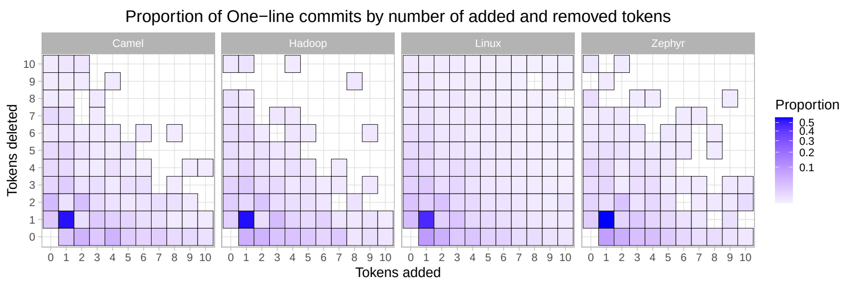


Fig. 3: Proportion of **one-line commits** by the number of tokens added or removed. The x and y-axis show the added and deleted tokens, and each cell indicates the proportion of commits.

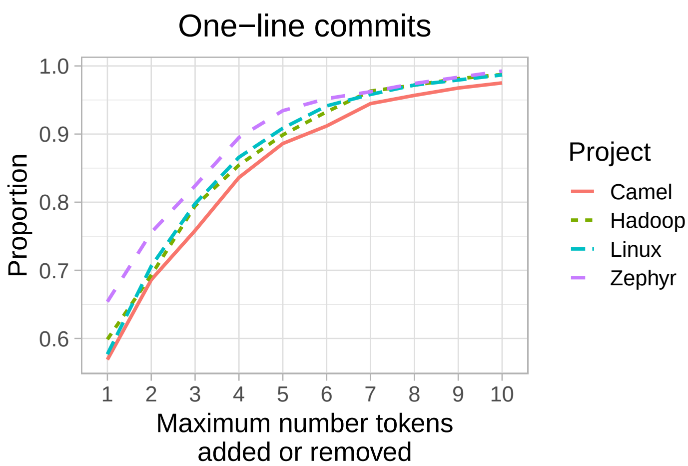


Fig. 4: Accumulated distribution of one-line commits in terms of the maximum number of added or removed tokens

Approximately 90% of one-line commits consist of at most five tokens. Table 1 shows the number of one-line commits. As described in Section 2, there are a non-negligible number of these commits in the studied projects (4.28–8.20%). Figure 3 shows the proportion of one-line commits according to the number of tokens that they have added and removed between 0 and 10. As can be seen, there are a significant number of one-line commits that remove and add exactly one token (between approximately 50 and 63% of all one-line commits). Furthermore, except for the case in the Hadoop project where no commits add or delete five tokens, all cells with five or fewer added and deleted tokens have more than one one-line commit across all projects. This implies that there are no empty cells within five added or deleted tokens except for one cell in the Hadoop project. Also, the distribution of one-line commits, with more than five tokens, varies across the projects. For instance, in the Hadoop and Zephyr projects, there are cells with no one-line commits of more than five deleted tokens and less than or equal to one added token. In contrast, every cell in the Camel and Linux projects has at least one one-line commit. Hence, the majority of one-line commits add or remove at most five tokens, and this finding is generally consistent across all projects.

Figure 4 shows the accumulated distribution of one-line commits according to the maximum number of tokens they add or remove. We use the maximum number of tokens added or removed in this figure. This is because our definition of a micro commit applies the same threshold of five tokens to both the number of added and removed tokens. As can be seen, between approximately 57% and 65% add-or-remove at most one token, between 76% and 82% add-or-remove at most three tokens, and between 89% and 93% add-or-remove at most five tokens. Thus, approximately 90% of one-line commits can be covered by our micro commits.

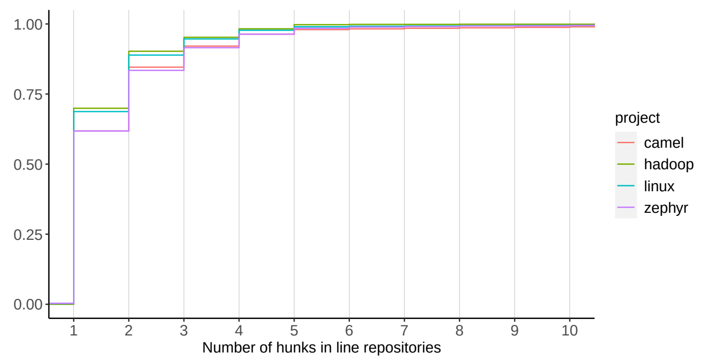


Fig. 5: Accumulated distribution of micro commits ($N = 5$) in terms of the number of hunks included

The number of modified hunks is also a crucial characteristic of commits. By our definition, one-line commits only modify one location in the source code (i. e., one hunk). We define micro commits based on the number of tokens, so even if a commit is spread across multiple locations (i. e., multiple hunks), it it still considered a micro commit if the number of modified tokens is below a certain threshold. This is a significant distinction compared to one-line commits. Therefore, we do not impose any limits on the number of modified hunks.

Figure 5 illustrates the accumulated distribution of the number of hunks included in micro commits to investigate their difference from one-line commits. Approximately 70% (Linux and Hadoop) or 60% (Zephyr and Camel) of micro commits contain a single hunk, while the remaining commits encompass two or more hunks. Hence, while approximately 70% or 60% of micro commits share characteristics with one-line commits, the remaining 30% or 40% represent commits that one-line commits do not detect, even if they modify the same number of tokens.

In conclusion, although micro commits can encompass nearly all one-line commits, the reverse is not typically true: one-line commits do not generally cover micro commits. Indeed, Table 2 in Section 3.3 reveals that around 90% of one-line commits can be encapsulated by micro commits. However, only approximately 40% (for Linux and Zephyr) or 50% (for Camel and Hadoop) of micro commits can be encapsulated by one-line commits. Therefore, micro commits provide new insights compared to one-line commits.

Summary of RQ3

Approximately 90% of one-line commits add or remove at most five tokens. Therefore, nearly all one-line commits can be covered by micro commits. In contrast, 30 to 40% of micro commits include two or more hunks that are not covered by one-line commits. In fact, only approximately 40% (for Linux and Zephyr) or 50% (for Camel and Hadoop) of micro commits can be encapsulated by one-line commits. Therefore, the characteristics of micro commits can help us understand the attributes of small changes, including those in one-line commits and commits not identified by one-line commits.

7 Discussion

In this paper, we defined micro commits and shed light on their characteristics. The main motivation is that turning our attention to micro commits would benefit software engineering research. In this section, we describe the implications of micro commits on future research.

7.1 Line-based vs Token-based Complexity Metrics

Line-based complexity metrics (e. g., LA, LD, and LT) [21] are one of the most popular source code complexity metrics in software engineering. However, **there are limitations to using line-based complexity metrics, as they may overlook capturing commits with equivalent complexity in terms of changed tokens.** We have shown examples with different numbers of changed lines but the same number of changed tokens (Listings 1 and 2). We have shown that in the projects under analysis, Listing 10: Example "replace constant" commit diff retrieved from `c143708acfb17e91c5e4fc9bd9b496fc7d2db29c` in Hadoop.

```
@@ -71 +71 @@ protected void render(Block html) {
-       html.h1()._("Invalid log start value: " + $("end"))._();
+       html.h1()._("Invalid log end value: " + $("end"))._();
```

approximately 90% of one-line commits are micro commits, but only approximately 40–50% of micro commits are one-line commits (Table 2).

Thus, **future research should consider tokens (and their types) as an additional metric of the complexity of commits**. Because, for example, prior studies [24,33] in defect prediction reported that current models heavily rely on the added lines, such new metrics would provide new information to identify defective commits accurately.

7.2 Micro Commits Are Non-negligible and Should Be Further Studied

In Section 3.3, we showed that micro commits account for between 7.45 and 17.95% of all studied commits, which is quite high. Furthermore, 1 in 3 or 4 these changes (2.39 and 4.88% of all studied commits) simply change one token. Hence, micro commits, including their finest-grained form, the one-token commit, represent a non-negligible development activity. Thus, **we need to understand how to better support developers, first, by deeply looking at the need for micro commits, and second, by reducing the amount of effort needed to complete these changes**.

7.3 Program Repair

The results of RQ1 showed that micro commits frequently modify a single token, and its token type is name, literal, or operator. **Studying micro commits could help understand how software is modified with such a tiny amount of change, and provide datasets that improve methods that attempt to modify software automatically.** For example, datasets based on micro commits might improve data-driven program repair approaches that have been studied so far [20, 29, 31]. One potential idea involves utilizing our observations of frequently modified token types and tokens in Java and C. Our observations indicate that while the types of frequently modified tokens are similar, the actual tokens differ across languages. This information is important for developing a program repair approach. When dealing with multiple languages, focusing on token types is crucial. However, when focusing on a specific language, actual tokens can also be beneficial.

Also, **the empirical investigation of micro commits would reveal types of micro commits that are difficult to be generated by program repair approaches**. Listing 10 shows a *replace constant* micro commit example. Specifically, this commit changes a string literal token: "start" into "end". Such a change might be difficult to be generated automatically, because it is not obvious why a literal token should be replaced by another one; however, other changes (including micro commits) might have performed this specific replacement somewhere else.

Some large commits might actually be composed of several micro commits (i. e., *tangled commits* [8, 16, 23]). Thus **it is worth also exploring the possibility of untangling micro commits from larger commits**. These untangled micro commits might be very valuable for program repair.

Finally, we present initial analysis results for micro commits regarding their maintenance activities. As stated in Section 1, we hypothesize that small changes are likely intended for maintenance purposes. Therefore, we deduce that exploring micro commits could be beneficial for program repair. To validate this hypothesis, we identify micro commits that fall under the corrective maintenance category as defined by Swanson [40]. Corrective maintenance is performed in response to failures. If corrective maintenance makes up a large proportion of micro commits compared to non-micro commits, it would confirm our hypothesis.

To identify the corrective commits, we followed the methodologies used in prior studies [35, 37], which use keywords in commit messages. More specifically, if at least one of the keywords is included in the commit messages, we classify the commit into the corrective maintenance category. Otherwise, we do not label commits. We used the keyword list defined by Levin et al. [27] as follows: “fix”, “esolv”, “clos”, “handl”, “issue”, “defect”, “bug”, “problem”, “ticket”.

The detailed procedure is as follows.

Step 1: Apply preprocessing to the commit messages using the NLTK package[11] in Python by following the steps below:
- Tokenize the text and convert all words to lowercase.
- Remove stopwords and punctuation.
- Perform stemming on all words.

Step 2: Check if the stemmed commit message contains a keyword.

Step 3: Identify commits that fall under the corrective category.

Micro commits are more likely to be failure-fixing activity than other commits. Figure 6 shows the proportion of corrective micro and non-micro commits. In this figure, we compare the tendency of micro commits (light gray) and non-micro commits (dark gray). Corrective micro commits are larger than non-micro commits. Hence, micro commits distinguishably correspond to the corrective commits. This result shows that micro commits are usually applied to the source code to fix failures.

Also, this finding confirms our initial assumption that micro commits are used more frequently for maintenance purposes than non-micro commits. Interestingly, Hattori and Lanza [15] found similar results, noting that tiny commits are often associated with corrective activities.

It should be noted that the keyword-based approach generally lacks accuracy [3, 25]. To verify the accuracy of the identification, we manually inspect 20 micro commits and 20 non-micro commits identified as corrective, classifying them into three failure types within the corrective category defined by Swanson [40]. If we cannot associate any failure types, those would be considered false positive corrective commits. This allows us to estimate the actual number of corrective micro commits and

[11] `https://www.nltk.org/`

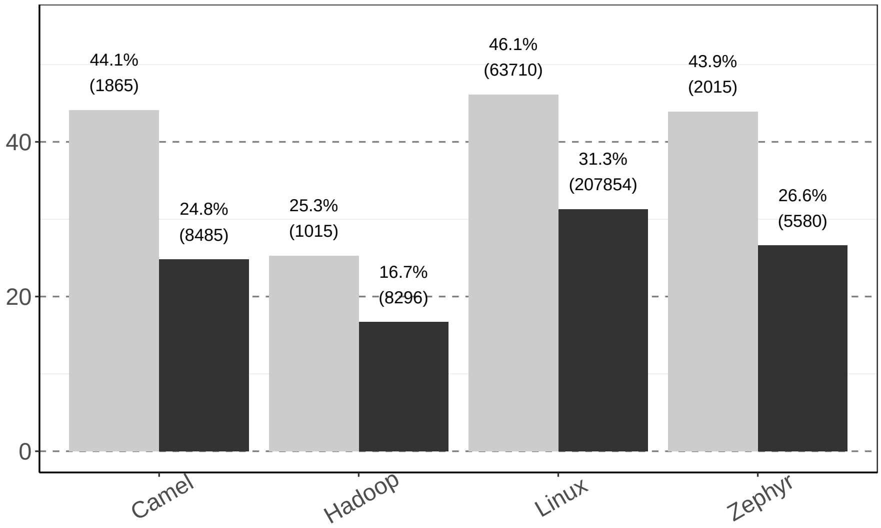


Fig. 6: The proportion of commits for the corrective category. Light gray indicates the proportion in micro commits; dark gray indicates the proportion in non-micro commits.

corrective non-micro commits in the identified commits. We do not examine non-corrective commits to determine the proportion of false-negative corrective commits. This discussion only reports the minimum percentage of corrective commits. Our manual inspection revealed that there were no false-positive corrective micro commits. In contrast, we found 8 out of 20 false positive corrective commits in non-micro commits. This finding suggests that the percentage of corrective micro commits may not change significantly, while the proportion of corrective non-micro commits could decrease. Therefore, our conclusion remains unchanged. In this manual inspection, we inspect only 20 micro and non-micro commits. Future studies could improve the validity of our findings. Our inspection is avaiable in the spreadsheet.[12]

7.4 Size-perspective vs. Semantic-perspective for Defining Micro Commits

In this paper, we define micro commits through size metrics (i. e., the number of tokens). This is because we would like to assist with software engineering research, such as program repair. However, micro is a general term, and micro commits can be defined not only by size but also by semantic aspects. For instance, tangled commits [8, 16, 23] can be considered non-micro, whereas non-tangled commits can be categorized as micro. Additionally, defect-fixing commits can be categorized as micro or non-micro depending on the difficulty of the bug being fixed. We could explore

[12] `https://docs.google.com/spreadsheets/d/17cqps6oSkA86GPuUmin3W1FoRZXslQfIuJAkX3vqH28/edit?usp=sharing`

these aspects using non-source code resources, such as source code comments, issue reports, and mailing lists. Exploring these semantic-based micro commits can also contribute to software engineering research.

8 Threats to Validity

8.1 External Validity

We conducted our empirical study on four OSS projects. To mitigate the threats to generalizability, we selected OSS projects that are active, popular, and well-known OSS projects written in two popular programming languages. However, even if we use these OSS projects, our results may not be generalized to all projects. Indeed, these are system software. To remedy this challenge, replication studies in research or practical scenarios (e.g., actual projects in the industry) are necessary. Hence, we provide a replication package[13]. Also, the key tool `cregit` is an OSS tool; thus, researchers and practitioners easily convert their Git repositories into token-based ones.

8.2 Construct Validity

We define micro commits based on the number of changed tokens. However, micro commits are a general term, and we can make different definitions. The key characteristic of micro commits is that such commits change a small code fragment. Our analysis (RQ1 and 2) shows that our definition is consistent with this characteristic. Hence, we believe our definition can be acceptable. However, our definition may not be the best; thus, future studies are necessary to find a better definition than our first one. For example, future studies can investigate different thresholds for the number of tokens. Also, studying different thresholds for added and deleted tokens (e. g., 3 added tokens and 5 deleted tokens) can be beneficial. Additionally, they can consider changes to source code comments. This definition would encompass not only maintenance activities related to code logic, but also various other maintenance activities.

For future studies, researchers can use ASTs to tokenize the source code instead of `cregit`, which we used in this paper. While ASTs are powerful in analyzing token-level information, `cregit` is designed for Git, a de facto standard version control system. Hence, researchers easily analyze the software development history to support its process when using `cregit` instead of ASTs. Hence, we recommend using `cregit` in future studies.

There are several factors that can influence commits. For instance, the way developers write commits can vary depending on the developer and the project. To reduce these influences, we chose and analyzed four projects that involve a large number of developers. As a result, we expect the impact of such influences has been minimized.

[13] https://github.com/MKmknd/EMSE2024-micro-commits-replication

8.3 Internal Validity

To remove comment lines from source code files, we use regular expressions. This process is not perfect and may overlook comment lines. However, our manual analysis in RQ2 observes that our regular expressions usually work well because we do not find false-positive micro commits. Therefore, we can reduce the risks associated with using regular expressions.

Also, another threat exists in our manual analysis (RQ2). In this analysis, we performed manual labeling to micro commits according to our coding guide. Because this process is performed manually by the first author, the result may have false-positive and false-negative results. Therefore, we have made all labels publicly available to facilitate the validation of future studies. Also, to construct the coding guide, the first three authors independently inspected 20 micro commits three times. This process may also include errors. However, our agreement rate achieved substantial agreement in two consecutive iterations. Hence, we believe the coding guide is reliable. An alternative solution is to use an automatic classification approach rather than manual analysis. We developed a heuristic-based method to classify micro commits into their corresponding targets automatically. However, this method does not yield perfect results. To facilitate replication of this approach, we have included it in our replication package. Finally, we randomly sampled 400 micro commits from all projects. Therefore, our sampled micro commits may be biased by the size of the original projects. To mitigate this risk, we manually inspected additional micro commits from each project. We do not observe significant differences across the projects.

In the discussion, we use keywords to identify the commits related to the corrective maintenance activity as defined by Swanson [40]. While the keyword identification is widely used to categorize commits [18,22,27,28,35,37], it is not perfect [3,25]. To mitigate this threat, we manually review identified commits and estimate their accuracy. Also, there are other sets of maintenance activities that can be used to classify commits, such as the IEEE standard [1]. While we believe the maintenance activities defined by Swanson are acceptable, future studies are necessary to use other sets. Also, if commit messages do not contain any keywords, we exclude those commits from the analysis. However, it is possible that these commits are related to maintenance activities. Using more precise methods would enhance the validity of this analysis.

The tool "cregit" used to tokenize the source code files utilizes srcML. Therefore, our analysis can only be applied to specific versions of Java (Java SE8 Edition) and C (up to C11) that are supported by srcML. We can find the supported versions on the official homepage[14]. To extend our analysis to different versions of Java and C, it is necessary to update srcML and apply our analysis to those versions.

[14] `https://www.srcml.org/#home`

9 Related Work

9.1 Challenges of Mining Git Repositories

Prior studies [4, 5, 11, 39] investigated and intended to address the challenges of mining Git repositories. For example, as described in Section 2, some non-functional changes update the information for each line and make it difficult to track code changes accurately. `cregit` [11] is proposed to address this problem by improving the blame feature in Git. More specifically, `cregit` tokenizes each line and applies the blame feature to the tokenized files. Spacco and Williams [39] proposed a technique SDiff to track changes at the statement level instead of the line level. This technique combines previous line- and structural-based approaches. Specifically, SDiff tokenizes each statement and uses diff between revisions. These techniques tokenize the source code to address this problem. Similarly, we define micro commits based on changed tokens in this paper to track code changes accurately.

9.2 Change Classification

Classifying changes (e. g., commits) into a certain category is a research topic in mining software repositories so far [2, 10, 12, 17, 18, 22, 27, 28, 32, 35, 37, 44, 45]. For example, many prior studies intend to classify changes in terms of the purpose [10, 12, 17, 18, 22, 27, 28, 32, 35, 45]. Levin et al. [27] classified commits into the maintenance activities defined by Swanson [40]. Hindle et al. [17] used machine learning classifiers to classify changes into the extended Swanson categories. Ghadhab et al. [12] used a pre-trained deep learning model known as BERT to classify commits into maintenance categories.

On the contrary, in this paper, we classify commits into micro commits based on their size and empirically investigate their characteristics, and there are several similar prior studies [2, 18, 37]. Purushothaman and Perry [37] classified changes into three categories and studied them: one-line changes, small changes, and all. Specifically, this study used the number of changed lines for this classification. Hindle et al. [18] identified large commits based on the number of changed files and revealed the characteristics of large commits. They also compared their result with the characteristics of the small commits by Purushothaman and Perry [37]. Alali et al. [2] empirically investigated the characteristics of commits in nine OSS projects. They used three size criteria: the number of files, lines, and hunks. For example, they found that approximately 19.9% of commits in the GNU gcc system change at most five lines. However, the finest-grained changed source code entity is a line in these papers, and such an entity loses the information of changed tokens in a line. This limitation makes it difficult to define a certain category of commits based on finer-grained source code changes. Hence, our investigation would provide a new research direction in which researchers and practitioners use token-level changes.

9.3 Knowledge Gap in Previous Studies

Compared to these prior studies, this research is the first to define micro commits at a fine granularity, specifically at the token level, through empirical analysis. Small commits defined at the line level, which previous studies often used, may overlook important information for improving existing software engineering research. Our research addresses this knowledge gap by conducting the analysis at the token level.

For instance, as explained in Section 7.3, the findings of RQ1 in this study have implications for research in program repair. These findings indicate the need to explore approaches for fixing bugs caused by a single name or literal token. This is because existing automated program repair approaches [20, 26, 30] may not be effective in such scenarios due to a lack of information to repair the code. These findings and implications were obtained because the analysis was conducted at the token level. It would have been difficult to obtain such findings and implications using a line-level analysis. The novelty of this study lies in conducting the analysis at the token level and providing these implications. The details of our findings and implications can be found in Sections 5, 6 and 7.

10 Conclusion

In this paper, we defined micro commits (add at most five tokens and remove at most five tokens) and investigated their characteristics. This research is the first to define micro commits at a fine granularity, specifically at the token level. The key novelty of this study lies in conducting the analysis at the token level and providing implications for software engineering research.

Below, we present a summary of the findings from our empirical analysis:

- Our defined micro commits account for between 7.45–17.95% of all studied commits. Approximately 1 in 3 or 4 these changes (2.39–4.88% of all studied commits) involve replacing one token with another. Furthermore, RQ3 demonstrates that approximately 90% of one-line commits are micro commits, but only approximately 40–50% of micro commits are one-line commits. In fact, approximately 30–40% of micro commits include two or more hunks.
- The results of RQ1 show that micro commits primarily affect name token types (37.7–44.5%), literal token types (9.2–34.9%), or operator token types (6.6–10.4%). The most frequently affected tokens vary: the period in Java (2.5% in Camel and 3.9% in Hadoop) and the 0/1 in C (1.8 and 1.0% in Linux and 1.3 and 0.9% in Zephyr). Furthermore, the most frequently observed pattern is the modification of a single token. In Java projects, this modification is typically a single literal token. On the other hand, in C projects, the modification is usually a single name token.
- The results of RQ2 indicate that approximately 86% of micro commits involve a single operation on a single target, with the main focus being the replacement of existing targets. The multi-operation micro commits primarily involve changing the order of statements (19.3%).

In the discussion, we presented the following four implications of micro commits on future research based on the findings:

- Based on RQ3, it is observed that almost all one-line commits are micro commits, whereas only 40–50% of the micro commits are one-line commits. Therefore, token-based complexity metrics offer supplementary information to the commonly used line-based complexity metrics. Designing metrics to measure token-based complexity is a potential area for future research.
- Based on the statistics of micro commits, they account for a non-negligible proportion of all studied commits (7.45–17.95%). Additionally, according to Section 7.3, these commits are more likely used to fix bugs. Therefore, supporting the development of micro commits is an important area for future research.
- Based on RQ1, micro commits frequently modify a single token, with the token type often being either a name or a literal. While these micro commits often address bug fixes, suggesting patches to fix individual name or literal tokens can be challenging with existing program repair approaches. Therefore, it is necessary to investigate these micro commits and propose new program repair approaches for future research.
- We define micro commits based on size metrics. However, micro is a general term, and micro commits can be defined not only by size but also by semantic aspects (e. g., tangled commits or not). Exploring semantic-based micro commits is a potential area for future research.

The key message of this paper is as follows:

The token-level definition could help researchers and practitioners to improve software engineering approaches for software quality assurance activities.

Acknowledgment

This work has been supported by JSPS KAKENHI Japan (Grant Numbers: JP21H04877, JP22K17874, and JP22K18630) and NSERC Canada. Yasutaka Kamei is supported by Inamori Research Institute for Science, Kyoto, Japan.

Declarations

Funding and/or Conflicts of interests/Competing interests

All authors certify that they have no affiliations with or involvement in any organization or entity with any financial interest or non-financial interest in the subject matter or materials discussed in this manuscript.

Data Availability Statements

The replication package that supports the findings of this study is available in our GitHub repository[15].

References

1. Iso/iec/ieee international standard for software engineering - software life cycle processes - maintenance. ISO/IEC 14764:2006 (E) IEEE Std 14764-2006 Revision of IEEE Std 1219-1998) pp. 1–58 (2006). DOI 10.1109/IEEESTD.2006.235774
2. Alali, A., Kagdi, H., Maletic, J.I.: What's a typical commit? a characterization of open source software repositories. In: Proceedings of the 16th IEEE International Conference on Program Comprehension (ICPC), pp. 182–191. IEEE (2008)
3. AlOmar, E.A., Mkaouer, M.W., Ouni, A.: Toward the automatic classification of self-affirmed refactoring. Journal of Systems and Software **171**, 110,821 (2021)
4. Asaduzzaman, M., Roy, C.K., Schneider, K.A., Penta, M.D.: Lhdiff: A language-independent hybrid approach for tracking source code lines. In: Proceedings of the IEEE International Conference on Software Maintenance (ICSM2013), pp. 230–239. IEEE Computer Society (2013)
5. Bird, C., Rigby, P.C., Barr, E.T., Hamilton, D.J., German, D.M., Devanbu, P.: The promises and perils of mining git. In: Proceedings of the 6th IEEE International Working Conference on Mining Software Repositories (MSR), pp. 1–10 (2009)
6. Chouchen, M., Ouni, A., Kula, R.G., Wang, D., Thongtanunam, P., Mkaouer, M.W., Matsumoto, K.: Anti-patterns in modern code review: Symptoms and prevalence. In: Proceedings of the 28th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER2021), pp. 531–535. IEEE (2021)
7. Collard, M.L., Decker, M.J., Maletic, J.I.: srcml: An infrastructure for the exploration, analysis, and manipulation of source code: A tool demonstration. In: Proceedings of the 2013 IEEE International Conference on Software Maintenance, pp. 516–519. IEEE (2013)
8. Dias, M., Bacchelli, A., Gousios, G., Cassou, D., Ducasse, S.: Untangling fine-grained code changes. In: Proceedings of the 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER), pp. 341–350. IEEE (2015)
9. Fleiss, J.L.: Measuring nominal scale agreement among many raters. Psychological Bulletin **76**, 378–382 (1971)
10. German, D.M.: An empirical study of fine-grained software modifications. Empirical Software Engineering **11**, 369–393 (2006)
11. German, D.M., Adams, B., Stewart, K.: cregit: Token-level blame information in git version control repositories. Empirical Software Engineering **24**(4), 2725–2763 (2019)
12. Ghadhab, L., Jenhani, I., Mkaouer, M.W., Messaoud, M.B.: Augmenting commit classification by using fine-grained source code changes and a pre-trained deep neural language model. Information and Software Technology **135** (2021)
13. Hata, H., Novielli, N., Baltes, S., Kula, R.G., Treude, C.: Github discussions: An exploratory study of early adoption. Empirical Software Engineering **27**, 1–32 (2022)
14. Hata, H., Treude, C., Kula, R.G., Ishio, T.: 9.6 million links in source code comments: purpose, evolution, and decay. In: Proceedings of the 41st International Conference on Software Engineering (ICSE2019), pp. 1211–1221 (2019)
15. Hattori, L.P., Lanza, M.: On the nature of commits. In: Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE), p. III–63–III–71 (2008)
16. Herzig, K., Zeller, A.: The impact of tangled code changes. In: Proceedings of the 10th Working Conference on Mining Software Repositories (MSR), pp. 121–130. IEEE (2013)
17. Hindle, A., German, D.M., Godfrey, M.W., Holt, R.C.: Automatic classication of large changes into maintenance categories. In: Proceedings of the IEEE 17th International Conference on Program Comprehension (ICPC), pp. 30–39. IEEE (2009)

[15] `https://github.com/MKmknd/EMSE2024-micro-commits-replication`

18. Hindle, A., German, D.M., Holt, R.: What do large commits tell us? a taxonomical study of large commits. In: Proceedings of the 2008 International Working Conference on Mining Software Repositories (MSR), pp. 99–108 (2008)
19. Hoang, T., Dam, H.K., Kamei, Y., Lo, D., Ubayashi, N.: Deepjit: an end-to-end deep learning framework for just-in-time defect prediction. In: Proceedings of the 16th International Conference on Mining Software Repositories (MSR), pp. 34–45. IEEE (2019)
20. Jiang, J., Xiong, Y., Zhang, H., Gao, Q., Chen, X.: Shaping program repair space with existing patches and similar code. In: Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 298–309 (2018)
21. Kamei, Y., Shihab, E., Adams, B., Hassan, A.E., Mockus, A., Sinha, A., Ubayashi, N.: A large-scale empirical study of just-in-time quality assurance. IEEE Transactions on Software Engineering **39**(6), 757–773 (2013)
22. Karampatsis, R.M., Sutton, C.: How often do single-statement bugs occur? the manysstubs4j dataset. In: Proceedings of the 17th International Conference on Mining Software Repositories (MSR), pp. 573–577 (2020)
23. Kirinuki, H., Higo, Y., Hotta, K., Kusumoto, S.: Hey! are you committing tangled changes? In: Proceedings of the 22nd International Conference on Program Comprehension (ICPC), pp. 262–265 (2014)
24. Kondo, M., German, D.M., Mizuno, O., Choi, E.H.: The impact of context metrics on just-in-time defect prediction. Empirical Software Engineering **25**(1), 890–939 (2020)
25. Kondo, M., Kashiwa, Y., Kamei, Y., Mizuno, O.: An empirical study of issue-link algorithms: which issue-link algorithms should we use? Empirical Software Engineering **27** (2022)
26. Le Goues, C., Nguyen, T., Forrest, S., Weimer, W.: Genprog: A generic method for automatic software repair. IEEE Transactions on Software Engineering **38**(1), 54–72 (2012)
27. Levin, S., Yehudai, A.: Using temporal and semantic developer-level information to predict maintenance activity profiles. In: Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 463–467. IEEE (2016)
28. Levin, S., Yehudai, A.: Boosting automatic commit classification into maintenance activities by utilizing source code changes. In: Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering, pp. 97–106 (2017)
29. Liu, K., Kim, D., Koyuncu, A., Li, L., Bissyandé, T.F., Le Traon, Y.: A closer look at real-world patches. In: Proceedings of the International Conference on Software Maintenance and Evolution (ICSME), pp. 275–286 (2018)
30. Liu, K., Koyuncu, A., Kim, D., Bissyandé, T.F.: Tbar: Revisiting template-based automated program repair. In: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 31–42 (2019)
31. Martinez, M., Monperrus, M.: Mining software repair models for reasoning on the search space of automated program fixing. Empirical Software Engineering **20**, 176–205 (2015)
32. Mauczka, A., Brosch, F., Schanes, C., Grechenig, T.: Dataset of developer-labeled commit messages. In: Proceedings of the 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories (MSR), pp. 490–493. IEEE (2015)
33. McIntosh, S., Kamei, Y.: Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction. IEEE Transactions on Software Engineering **44**(5), 412–428 (2018)
34. Meng, X., Miller, B.P., Williams, W.R., Bernat, A.R.: Mining software repositories for accurate authorship. In: Proceedings of the 2013 IEEE International Conference on Software Maintenance, pp. 250–259. IEEE (2013)
35. Mockus, A., Votta, L.G.: Identifying reasons for software changes using historic databases. In: Proceedings of the International Conference on Software Maintenance (ICSM), pp. 120–130 (2000)
36. Nugroho, Y.S., Hata, H., Matsumoto, K.: How different are different diff algorithms in git? use–histogram for code changes. Empirical Software Engineering **25**, 790–823 (2020)
37. Purushothaman, R., Perry, D.E.: Toward understanding the rhetoric of small source code changes. IEEE Transactions on Software Engineering **31**(6), 511–526 (2005)
38. Servant, F., Jones, J.A.: Fuzzy fine-grained code-history analysis. In: Proceedings of the IEEE/ACM 39th International Conference on Software Engineering (ICSE), pp. 746–757. IEEE (2017)
39. Spacco, J., Williams, C.C.: Lightweight techniques for tracking unique program statements. In: Proceedings of the IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM2009, pp. 99–108. IEEE Computer Society (2009)
40. Swanson, E.B.: The dimensions of maintenance. In: Proceedings of the 2nd International Conference on Software Engineering (ICSE), pp. 492–497 (1976)

41. Viera, A.J., Garrett, J.M.: Understanding interobserver agreement: the kappa statistic. Family Medicine **37**(5), 360–363 (2005)
42. Wang, D., Kondo, M., Kamei, Y., Kula, R.G., Ubayashi, N.: When conversations turn into work: a taxonomy of converted discussions and issues in github. Empirical Software Engineering **28**(6), 138 (2023)
43. Wang, D., Xiao, T., Thongtanunam, P., Kula, R.G., Matsumoto, K.: Understanding shared links and their intentions to meet information needs in modern code review. Empirical Software Engineering **26** (2021)
44. Williams, B.J., Carver, J.C.: Characterizing software architecture changes: A systematic review. Information and Software Technology **52**(1), 31–51 (2010)
45. Yan, M., Fu, Y., Zhang, X., Yang, D., Xu, L., Kymer, J.D.: Automatically classifying software changes via discriminative topic model: Supporting multi-category and cross-project. Journal of Systems and Software **113**, 296–308 (2016)
46. Zanaty, F.E., Hirao, T., McIntosh, S., Ihara, A., Matsumoto, K.: An empirical study of design discussions in code review. In: Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, pp. 1–10 (2018)

Appendix

A Example micro commits

Listing 11: Example "add statement" commit diff retrieved from `122503683169b21d9cdb90380a20caad7ba994b6` in Linux.

```
@@ -240,0 +241 @@ static int check_partial_mapping(struct drm_i915_gem_object *obj,
+                cond_resched();
```

Listing 12: Example "replace constant" commit diff retrieved from `b7a90e8043e7ab1922126e1c1c5c004b470f9e2a` in Linux.

```
@@ -174 +174 @@ static int hfsplus_sync_fs(struct super_block *sb, int wait)
-        dprint(DBG_SUPER, "hfsplus_write_super\n");
+        dprint(DBG_SUPER, "hfsplus_sync_fs\n");
```

Listing 13: Example "remove statement" commit diff retrieved from `b95b4e1ed92a203f4bdfc55f53d6e9c2773e3b6d` in Linux.

```
@@ -2973 +2972,0 @@ void * __meminit alloc_pages_exact_nid(int nid, size_t size, gfp_t gfp_mask)
-EXPORT_SYMBOL(alloc_pages_exact_nid);
```

Listing 14: Example "no" commit diff retrieved from `a092532483e3200a53c8b1170b3988cc668c07ef` in Linux.

```
@@ -1035 +1035 @@ static int __dwc3_gadget_ep_queue(struct dwc3_ep *dep, struct dwc3_request *req)
-        };
+        }
```

Listing 15: Example "change declaration" commit diff retrieved from `36f062042b0fd9f8e41b97a472f52139886ca26f` in Linux.

```
@@ -382 +382 @@ static ssize_t read_vmcore(struct file *file, char __user *buffer,
-static int mmap_vmcore_fault(struct vm_fault *vmf)
+static vm_fault_t mmap_vmcore_fault(struct vm_fault *vmf)
```

Listing 16: Example "change constant" commit diff retrieved from `1db76c14d215c8b26024dd532de3dcaf66ea30f7` in Linux.

```
@@ -185 +185 @@ struct mthca_cmd_context {
-static int fw_cmd_doorbell = 1;
+static int fw_cmd_doorbell = 0;
```

Listing 17: Example "change identifier" commit diff retrieved from `70e8b40176c75d3544024e7c934720b11a8a11bf` in Linux.

```
@@ -564 +564 @@ static irqreturn_t pcie_isr(int irq, void *dev_id)
-                       return IRQ_HANDLED;
+                       return IRQ_NONE;
```

Listing 18: Example "change control flow" commit diff retrieved from `415a1975923722f729211a9efca550c60c519bf3` in Linux.

```
@@ -530,0 +531 @@ static int ir_probe(struct i2c_adapter *adap)
+               break;
```

Listing 19: Example "change expression" commit diff retrieved from `40cc394be1aa18848b8757e03bd8ed23281f572e` in Linux.

```
@@ -1922 +1922 @@ static int nfs_parse_devname(const char *dev_name,
-                       *comma = 0;
+                       len = comma - dev_name;
```