2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)

Analyzing Comment-induced Updates on Stack Overflow

Abhishek Soni
Independent Researcher, India
abhisheksoni2720@gmail.com

Sarah Nadi
University of Alberta, Edmonton, Canada
nadi@ualberta.ca

***Abstract*—Stack Overflow is home to a large number of technical questions and answers. These answers also include comments from the community and other users about the answer's validity. Such comments may point to flaws in the posted answer or may indicate deprecated code that is no longer valid due to API changes. In this paper, we explore how comments affect answer updates on Stack Overflow, using the SOTorrent dataset. Our results show that a large number of answers on Stack Overflow are not updated, even when they receive comments that warrant an update. Our results can be used to build recommender systems that automatically identify answers that require updating, or even automatically update answers as needed.**

I. Introduction

Stack Overflow (SO) is currently an essential resource for software developers. Answers to already solved questions as well as the ability to post your own question and get feedback from developers all around the world makes it a popular platform that facilitates software development. Up-to-date information is important for any developer that comes to Stack Overflow looking for an answer to the problem they are facing. Answer posters can update their posted answers over time for a variety of reasons, such as API changes [1], flaws discovered in the code snippets later [2], or a revelation of a better way of solving the same problem. At the same time, users may leave comments on posted answers to point out, for example, code deprecation or reasons why the code snippet posted in the answer does not work.

The relationship between comments and updates is currently not clear. Are there comments that result in answer updates? If so, what kind of updates are typically requested? Do the answer posters typically update their answers based on the feedback? To the best of our knowledge, previous work that looked at Stack Overflow data only considered comments to determine user expertise or developer interactions (e.g., [3]–[5]), but did not consider the relationship between comments and answer updates. Understanding these relationships can help in developing automated recommender systems that either prompt answer posters to update their answers or warn SO users from using answers that still require updating.

In this paper, we use the SOTorrent dataset [6] to analyze how comments affect answer updates on Stack Overflow. We analyze answer updates and comments from SO questions tagged with with the top five tags: `java`, `javascript`, `python`, `php`, `android`. Using this data, we answer the following research questions (RQs).

RQ1. How often do comments induce answer updates? We find that, on average across five languages we analyzed, only 4.6% of the comments we studied result in an answer getting updated. About 38% of the comments do not warrant an update and 8.7% of the comments contain only text and/or other discussion.

RQ2. How often are comments ignored, even though they warrant an answer update? On average across the five tags we analyzed, 27.5% of the comments warrant an update but are ignored by the original posters of the answers. This suggests a need for techniques that can automatically update these answers, or at least tag them as *requires update* so SO users are aware before using them.

II. Considered Data

The SOTorrent dataset [6] contains 42M Posts from Stack Overflow, along with their fine grained edit histories. For the purposes of our work, we consider only a subset of this data. Specifically, we consider the top five tags on SO, `java`, `javascript`, `python`, `php`, `android`, and collect their relevant comment and answer update data as follows.

We use the following subset of SOTorrent tables for our analysis: `Posts`, `Comments`, `PostBlockVersion`, `PostHistory`. Our starting point for data collection is a given SO tag we are interested in analyzing. Given a tag, we follow the steps shown in Algorithm II. We store the data we collect in our own temporary database for faster analysis.

Algorithm 1 Data Acquisition from SOTorrent

```
1: Select all questions with given TAG
2: Select all answers to questions selected in Step 1.
3: Select all Comments posted to these answers
4: Select all edits made to these answers, provided there is
   a code block inside the answer and that it has changed.
```

First, we select IDs of all questions that have a particular tag (e.g., java, python, etc.) Corresponding to the questions, we select all the AnswersIDs associated with them. We then query the Comments table and collect all comments corresponding to the AnswerIDs we previously collected.

To collect the code block edits made to the answers, we use the `PostBlockVersion` table and `PostHistory` table from SOTorrent. To answer our research questions, we need to relate comments to answer updates. In other words, we are trying to identify when an answer was updated due to a comment being posted. To be able to identify such relationships as precisely as possible, we focus only on answer updates that involve updating a code snippet. We ignore updates where only text has changed. Accordingly, we select only edits made to code blocks present in the answer. Such code block updates can be identified using the `PredEqual` field which indicates if a particular code block has changed or not. Consequently, we select only edits where the `PredEqual` was either `NULL` (when the post was first created) or `FALSE` (the code block changed.)

III. Relating Comments to Answer Updates

Now that we have the full collection of comments and answer updates for all threads of interest, we need to map comments to answer updates. We do so in two steps. The first step identifies *candidate comments* that *may* have caused an update. In this step, we do a simple grouping of comments and answer updates based on the time the comment occurred w.r.t the edit. The second step applies three different heuristics whose goal is to analyze the content of the candidate comments and updates to come up with a more concrete relationship. Note that we first check if the comments contains any code elements. If it does not, we try to find any special keywords/phrase it may have. If the comment does not pass either of those criteria, it is simply discarded.

A. Identify Candidate Comments

The essential premise for the first step of mapping is that a comment could have caused an answer update only if it occurred *before* the update. We also assume that a comment is potentially related to the nearest update that happens after it. For example, assume the following events occurred in order where `up` denotes an answer update and `c` denotes a comment: `up0, c1, up1, c2, c3, up2, up3`. In this case, the candidate comments for `up1` are {`c1`}, the candidate comments for `up2` are {`c2, c3`}, and the candidate comments for `up3` is empty {}. Note that `up0`, the first edit in the list, is actually the creation of the post and no comment can exist before the answer itself is created. We ignore this first update in our processing.

For each answer, we iterate over all its edits and map each comment to an edit, according to the criteria described above. Note that, as the example illustrates, we map each comment to exactly one edit. However, each edit may have multiple comments mapped to it. If no edit was made to the answer after the creation date of the comment, we map the comment to *a dummy edit*. This would allow us to still take these comments into account in later steps. At the end of this step, we have a processed list of comments that *may* have induced an update. Such a mapping allows us to later reason about the relationship between comments and edits.

For each edit, we collect all the code blocks present in the `content` field of the current edit. The `content` field contains the code after the update/edit occurred. To identify the changes that happened in edit $\texttt{up}_{i+1}$, we consider the `content` of $\texttt{up}_{i+1}$ as `curr_content`. We then collect all the code blocks present in the previous edit $\texttt{up}_i$ and call it `prev_content`. We will later look for code elements present in the comment text as well as these two contents (`prev_content` & `curr_content`) which will help us label comments.

B. Categorize Comments

1) Categories of Comments: Before running our analysis, we had manually sampled comments from Stack Overflow to understand what types of comments exist in practice. We came up with four main categories that relate comments to updates.

1) **WARRANT_UPDATE**: A comment that warranted an update (e.g., "You should use `fs` instead of `path` as it is more reliable.") but an edit was ***not*** made to the answer.
2) **UPDATE**: A comment that warranted an update (e.g., "You should use `fs` instead of `path` as it is more reliable.") and an edit **was made** to the answer.
3) **NO_UPDATE**: A comment that did not warrant an update (e.g., "`makeRequest` is indeed the correct method to use in this case. Thanks!")
4) **UNKNOWN**: All other comments that are just text, urls or discussion (e.g., "Thank you so much for this wonderful answer.").

Based on the above categories we had defined, we design three different heuristics that allow us to automatically categorize comments. Note that all these heuristics apply only on candidate comments where the comment contains a code element, happened before the update, and potentially caused the update based on its proximity to the update. We apply the three heuristics in order.

Along with labeling the comments with one of the categories above, we also mark which heuristic resulted in the labeling. For the first heuristic, we mark the labeling cause as `CODE`. For the other two heuristics, we mark the labeling cause as `KEYWORD`.

2) Heuristic 1: Code Checks: This check is based on the following hypothesis: A comment can be classified as an UPDATE if it contains a code element that is not present in both `prev_content` or `curr_content`. Our reasoning is that if a candidate comment mentioned a code element that was not present before but then got added to the answer after the update, then it is likely that this change is in response to the comment. Since an edit can add or remove code elements from the answer based on what is suggested in the comment, we check for both directions: (a) a code element mentioned in the comment getting added, and (b) a code element mentioned in the comment getting deleted. If a code element from the comment is found in `curr_content` but not in `prev_content` or vice versa, we label that comment as `UPDATE`. If it is found in both `curr_content` and `prev_content`, we label it as `NO_UPDATE`. Note that this process is repeated for each code element found in the comment. The whole comment is labeled as `UPDATE` if at

TABLE I
TOTAL ANSWERS AND COMMENTS ANALYZED FOR EACH TAG, AS WELL AS THE DISTRIBUTION OF TYPES OF COMMENTS

Tag	Total Answers	Total Comments	Discarded Comments	No Update	Update	Warrant Update	Unknown
java	179,482	330,904	67,102 (20.28%)	123,809 (37.42%)	14,339 (4.33%)	97,077 (29.34%)	28,577 (8.64%)
python	158,778	297,057	62,597 (21.07%)	107,419 (36.16%)	16,045 (5.40%)	82,643 (27.82%)	28,353 (9.54%)
javascript	198,060	393,490	88,264 (22.43%)	149,242 (37.93%)	19,463 (4.95%)	99,996 (25.41%)	36,525 (9.28%)
android	137,400	279,782	54,930 (19.63%)	107,522 (38.43%)	12,579 (4.50%)	80,086 (28.62%)	24,665 (8.82%)
php	96,914	174,740	37,165 (21.27%)	70,673 (40.44%)	7,431 (4.25%)	46,039 (26.35%)	13,432 (7.69%)

least one code element from the candidate comment is updated in the answer. Also note that our search function uses regular expressions to match code elements that are function calls as functions mentioned in comments do not always map exactly to functions in the code snippet since they may not necessarily mention the complete parameter list (e.g., "comment `foo` takes two parameters.." while the code snippet has `foo(x)`).

3) Heuristic 2: Keyword/Word Phrase Checks: If the label from the previous Code Check heuristic is not `UPDATE`, or if the comment does not contain any code elements, we then check for the presence of specific keywords in the comment.

Based on our manual sampling, we had also compiled a list of word phrases/keywords that are related to update suggestions. Some examples of the comments we observed from various answers (in varying forms) are:

- This answer should be updated.
- Please update your answer.
- The code to change the path needs to be changed.

Based on such observations, we create a list of regular expressions to encode the word patterns that explicitly indicate that an update is requested or needed, as follows:

```
word_patterns=['needs(.+?)(update)', 'be(.+?)(renamed|replaced|
    ↪ updated|improved|changed)', 'update(.+?)(answer)', '
    ↪ change(.+?)(to)', 'v\d+.\d+.\d+']
```

The last RegEx is used to find any version number mentions in the comments. The mention of a version number is a strong indicator of API changes being discussed and, hence, is a useful word pattern.

In this second-step heuristic, we search for these word patterns in each comment. The search terminates either if it finds a match, or it exhausts the list of word patterns. If a match is found, we check if `prev_content` is equal to `curr_content`. (i.e., if the content has not changed.) If the content has changed, we label the comment as `UPDATE`. If it has not changed, we label the comment as `WARRANTS_UPDATE`.

4) Heuristic 3: Question Checks: If a comment has still not received a label until this point, we run one additional check on it. During manual sampling, we noticed a class of comments that had the following basic structure: `INTERROGATIVE WORD --- VERB --- CODE`? For example, "Why are you using `makeRequest` here?"

In these kind of comments, we observed that the commenter was trying to find out more about how the code works. It was, sometimes, also done to derive more information from the original poster about how the posted code snippet works. These comments can be classified as `WARRANTS_UPDATE` if we can confirm that the commenter is not merely asking more about the code snippet and has actually found a flaw in the code. We only run this third heuristics if the code has not changed in the answer update. We search for the pattern structure described above. If we find such a pattern, we label that comment as `WARRANTS_UPDATE`; otherwise, we label the `NO_UPDATE`.

To validate our heuristics, we randomly select 100 posts and run them through our system. Then, we sample 30 random posts out of the selected ones and manually verify the comment's labels. We find that around 85% of the comments have been correctly identified. We also observe that the most common mismatch case occurs between `WARRANTS_UPDATE` and `NO_UPDATE`. In the future, we plan to investigate better heuristics to differentiate those cases better.

IV. RESULTS

For our analysis, we select the top five language tags on SO: `java, javascript, python, php, android`. For each tag, we select all questions where the score (number of upvotes on SO) is 5 or more. We observe that questions with more upvotes on SO are more likely to attract more users in the community. They in turn have a higher number of answers posted to them, which consequently means a higher chance of comments on the answers. Since the goal of our work is to study the relationship between comments and updates, we need to ensure that we have enough of both in our dataset. We then select all answers from the selected questions for each language tag. For example, in the case of JavaScript, our dataset contains approximately 0.2M answers, with the number of related comments being almost 0.4M.

Table I shows the complete statistics for our dataset. The table also shows the number of different comment types in each language. Figure 1 visualizes the percentage of each comment category against each tag. The system discards about $\sim 23-24\%$ comments because they (a) do not contain any code element and (b) do not contain any special keyword etc. These are different from `UNKNOWN` as the latter is processed by the pipeline but cannot be labelled accurately.

Based on Table I and Figure 1, we can see that the percentage of comments that resulted in an update are quite

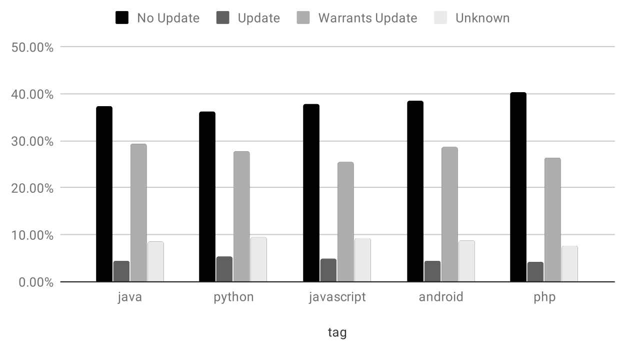

Fig. 1. Labeled Comments

TABLE II
REASON OF COMMENT LABELING

Tag	Code	Keyword
java	253,770	10,032
python	225,899	8,561
javascript	291,108	14,118
android	214,776	10,076
php	132,088	5,487

low (mostly $\sim 4-5\%$ across the five tags). On the other hand, many of the comments ($\sim 36-40\%$) are in the `NO_UPDATE` category. This suggests that these are more general comments or clarifications rather than comments that point out problems with the answer or suggest better code elements to use.

Finding 1: Many of the posted comments do not require an update ($\sim 36-40\%$), while only a few ($\sim 4-5\%$) actually resulted in answer updates.

We now look into the `WARRANT_UPDATE` category. This category is the most interesting as it gives an indication into the extent of outdated or wrong answers on Stack Overflow. We can see that more than quarter of the comments ($\sim 26-29\%$) are in the `WARRANT_UPDATE` category. This means that the comment pointed out something that needs to be updated in the answer, but the answer poster has not updated their answer accordingly. Up-to-date information is vital for any user or developer that comes on the website. Our finding suggests that there is potential for automated systems that automatically detect and/or flag outdated or wrong answers to improve the use of information found on Stack Overflow.

Finding 2: More than a quarter of the comments we studied across the five tags ($\sim 26-29\%$) require the answer to be updated, but are ignored by answer posters.

To help future work build on and improve our heuristics, we show the heuristic type (code check vs. keyword check) that resulted in the labeling of comments in Table II. As shown, most of the labeling was facilitated by the code checks. Only a small percentage of classifications involved the need for keyword/word phrases. This suggests that focusing on the code check heuristics and making them more precise is promising.

Finding 3: Code check heuristics seem to be more promising in labeling comments.

V. Threats to Validity

We now discuss potential threats to the validity of our results.

False Positives A comment can be mislabeled if the code element in the comment is not correctly identified by our system. The keyword/word phrases list we use is not exhaustive, and it is possible that we may have missed some of them.

Analyzing the entire dataset We run our pipeline only on a subset of threads, based on score, for the language tags we selected. Since questions with a higher score tend to receive more comments, this gave us more comments to work with for this challenge paper. In the future, we plan to run our analysis on the entire data set, while investigating additional heuristics. Additionally, we only focused on comments and answer updates involving code, since the code elements provide a starting point for matching comments to updates. Thus, our reported numbers are a lower bound, since some of the discarded comments could be comments that resulted in an update or warrant an update.

Closely occurring updates In order to get the candidate comments for an update, we assume that the comment must belong to the nearest edit. It is possible that multiple edits are made within a very small frame of time. In that case, it is quite possible that the actual *code correction* took place in the second edit. However, in our current setup, the comments will belong only to the first edit. We plan to investigate how often this edge case occurs and find heuristics to deal with it.

Heuristics All our matching criteria rely on heuristics. We manually sampled the results to verify that they work well. However, this also means that false positives or false negatives may occur. We decided to err on the side of precision: if we cannot precisely identify the relationship between a comment and update, we marked it as unknown.

VI. Conclusion

Comments are an integral part of Stack Overflow. They are a form of communication between users and help correct flaws in posted answers. However, our study shows that a large percentage of comments warrant an update, but are ignored by the original poster of the answers. Our results suggest the need to build automated systems that help Stack Overflow users pick the right answer without going through the comments or testing the code in the answer manually. We can also build systems that automatically notify posters of the need to update their answers, when required. In the future, we will expand our work and look at the updates that are caused by comments on a finer granularity level to check what type of updates are mostly prompted by comments (e.g., API Changes, flaw in code, etc.).

References

[1] C. Ragkhitwetsagul, J. Krinke, and R. Oliveto, “Awareness and experience of developers to outdated and license-violating code on stack overflow: An online survey,” *arXiv preprint arXiv:1806.08149*, 2018.

[2] C. Ragkhitwetsagul, J. Krinke, M. Paixao, G. Bianco, and R. Oliveto, “Toxic code snippets on stack overflow,” *IEEE Transactions on Software Engineering*, pp. 1–1, 2019.

[3] J. Yang, K. Tao, A. Bozzon, and G.-J. Houben, “Sparrows and owls: Characterisation of expert behaviour in stackoverflow,” in *International Conference on User Modeling, Adaptation, and Personalization*. Springer, 2014, pp. 266–277.

[4] D. Movshovitz-Attias, Y. Movshovitz-Attias, P. Steenkiste, and C. Faloutsos, “Analysis of the reputation system and user contributions on a question answering website: Stackoverflow,” in *Proceedings of the 2013 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*. ACM, 2013, pp. 886–893.

[5] S. Wang, D. Lo, and L. Jiang, “An empirical study on developer interactions in stackoverflow,” in *Proceedings of the 28th Annual ACM Symposium on Applied Computing*. ACM, 2013, pp. 1019–1024.

[6] S. Baltes, C. Treude, and S. Diehl, “Sotorrent: Studying the origin, evolution, and usage of stack overflow code snippets,” in *Proceedings of the 16th International Conference on Mining Software Repositories (MSR 2019)*, 2019.