

Does Code Review Promote Conformance?

A Study of OpenStack Patches

Panyawut Sri-iesaranusorn, Raula Gaikovina Kula, Takashi Ishio
 Nara Institute of Science and Technology, Nara, Japan
 Email: {sri-iesaranusorn.panyawut.sg0, raula-k, ishio}@is.naist.jp

Abstract—Code Review plays a crucial role in software quality, by allowing reviewers to discuss and critique any new patches before they can be successfully integrated into the project code. Yet, it is unsure the extent to which coding pattern changes (i.e., repetitive code) from when a patch is first submitted and when the decision is made (i.e., during the review process). In this study, we revisit coding patterns in code reviews, aiming to analyze whether or not the coding pattern changes during the review process. Comparing prior submitted patches, we measure differences in coding pattern between *pre-review* (i.e., patch before the review) and *post-review* (i.e., patch after a review) from 27,736 reviewed OpenStack patches. Results show that patches after review, tend to conform to similar coding patterns of accepted patches, compared to when they were first submitted. We also find that accepted patches do have similar coding patterns to prior accepted patches. Our study reveals insights into the review process, supporting the potential for automated tool support for newcomers and lays the groundwork for work into understanding conformance and how it makes for an efficient code review process.

I. INTRODUCTION

Code Review plays a crucial role in software quality by allowing reviewers to discuss and critique new patches before they can be successfully integrated into the project code. As a code quality assurance activity, code review also functions as knowledge transfer, team building, and coordination mechanism within software teams [2]. It has approval from industry giants like Microsoft and Google, on how ‘Code Reviews at Microsoft are an integral part of the development process that thousands of engineers perceive it as a great best practice and most high-performing teams spend a lot of time doing’. Although light-weight variations of code review streamline the process, reviews still suffer from being ineffective and less efficient. This can detract future contributions, especially for Open Source Software projects that need to attract and retain newcomer contributions.

Prior work has shown that similar coding patterns (i.e., such as the way developers edit day-to-day code that tends to be repetitive, often using existing code elements) is related to the acceptance of a patch. Hellendoorn et al. [5] shows that reviewers may consider conformance to the project’s code style as an indicator of acceptance. Comparing the submitted code with code in the project by using language models [12], they found that rejected changesets contain code significantly less similar to the project. Thus, it is not known the extent to which coding pattern changes during the review process, from when a patch is first submitted to when it is accepted.

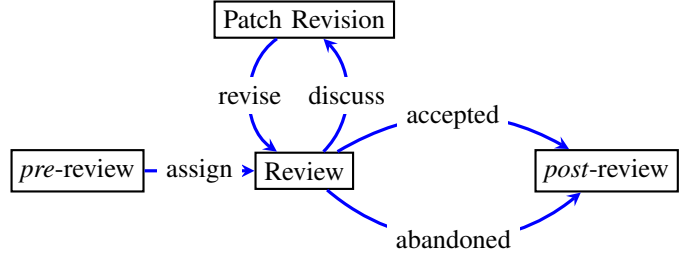


Fig. 1. The Code Review process shows key activities between the *pre-review* and *post-review* of a submitted patch.

To fill this gap, in this study, we revisit coding patterns in code reviews, aiming to analyze whether or not the coding pattern changes during the review process. We define the conformance as “similarly repetitive written code patterns that appear in prior accepted patches for a project”. We reuse language models to compare coding patterns or prior submitted patches before the review (i.e., *pre-review*) with a revised patch after the review (i.e., *post-review*), as shown in Figure 1. Also different to [5], we study patches, which are atomic, modular, and updated by design as opposed to pull requests, that are more a collection of commits, changed files, and the differences (or “diff”) between files in branches [10].

Our large-scale empirical study consisted of two parts. First, we conduct a preliminary study of 27,736 reviewed patches taken from the OpenStack projects. The preliminary results show that the programming files tend to contain more churn and are likely to contain coding pattern changes when compared to configuration or documentation types of files. Focusing on the programming files (i.e., JavaScript, Bash Shell, and Python), we then form two research questions to guide our study.

- *RQ₁ Review Process: Does a patch coding pattern conform to the project after it has been reviewed?*
- *RQ₂ Patch Decision: Are there coding pattern differences between an accepted and an abandoned patch?*

For RQ₁, results show that patches after being reviewed, tend to conform to similar coding patterns of accepted patches, compared to when they were first submitted. The results of RQ₂ also show that the conformance of accepted patches is higher than patches that were abandoned, which confirms prior findings. Our full replication package can be found at <https://zenodo.org/record/4537542#.YCaArhMza3I>

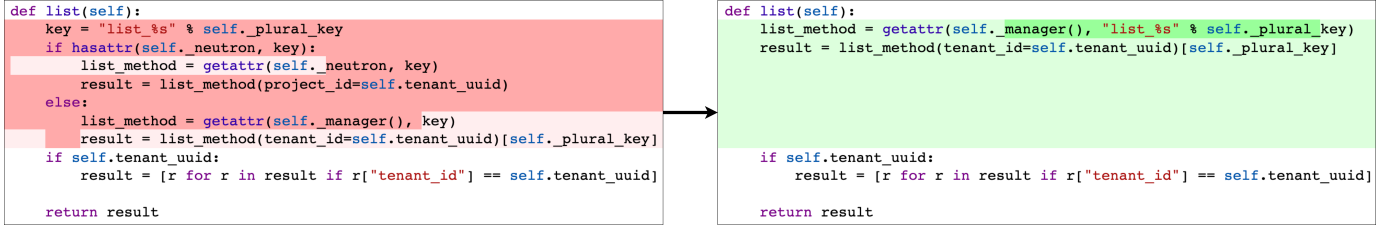


Fig. 2. An example of coding pattern changes between *pre-review* and *post-review*. We can see that *post-review* code has changed its coding pattern to change the key variable into the `list_method` variable. The code excluding `if-else` statement is consistent with other patches that have been accepted.

II. MOTIVATING EXAMPLE: IDENTIFYING SIMILAR CODING PATTERNS USING A LANGUAGE MODEL.

Figure 2 shows a review example [9] on how we compare the coding patterns that appear in the *post-review* compared to the *pre-review*. In this example, we see that the patch purpose is to add a helper function that “*is a way to unify the way of the usage of the Neutron API and remove code duplication*”. This patch has been through 50 patch revisions and consists of 14 python files. In over fifty patch revisions of the patch, we see that evidence of coding pattern changes with the variable `key` being changed into `list_method` variable. If the `list_method` variable is consistent in other prior accepted patches, we regard this as conformance to such repetitive coding patterns in the patch.

The statistical counts infer the probability of conformance to the coding pattern as the entropy. Low cross-entropy indicates the high similar style or conformance.

III. DATA PREPARATION

For our dataset collection, we acquired OpenStack patches from Ueda et al. [13], which is patch source code originally mined from the Gerrit API. The detail of each step is explained in the following steps: *Step1: Label pre and post reviews* - Our data was initially in JSON format, which is annotated as added, removed, or unchanged lines. From this data we create *pre-review* and *post-review*, containing the appropriate lines – added lines to only *post-review*, removed to only *pre-review*, and unchanged to both. *Step2: Remove comments* - To remove the comments, we use the NCDSearch tool [6]. The tool applies grammar from the lexer files generated by

ANTLR4 parser generator [1]. *Step3: Tokenize source code* - In addition to removing comments, we also used NCDSearch as our tokenization tool as it supports several languages such as Python, JavaScript, and plain text like .txt and .html.

For training a language model, we use the MITLM toolkit implemented in [8]. To understand how style changes due to the parameter n of the n -gram models, we measure the model created from 3-grams to 9-grams.

IV. CODING PATTERNS OF FILE TYPES

To address this gap, we perform a file-level analysis focused on the following two Preliminary Questions (PQs):

- *PQ₁ What kinds of files churn during a review?* We want to study the number of patch revisions of a typical review to better understand the magnitude of patches submitted.
- *PQ₂ Which kinds of reviewed files are likely to conform after the review process?* There is no prior work that quantifies which type of files tend to confirm after review.

Table I shows statistics of the collected corpus that belongs to the OpenStack project. This dataset will be used for all the preliminary questions and contains 27,736 revision patches over 177,826 files. Both the *pre-review* and *post-review* combined contain 900M tokens with 0.70M unique tokens.

To evaluate the kinds of files, we use the same groupings employed by McIntosh et al. [7]. Hence, we classify each unique file that ever existed in the analyzed time span as either configuration, programming, or documentation files. The list below shows the file extensions that we manually investigated and classified:

- *Configuration files (conf.):* The file extensions include .yaml, .json, .xml, .pp, and .yaml.
- *Programming files (prog.):* The file extensions include .sh, .py, and .js.
- *Documentation (doc.):* The file extensions include .rst, .php, .html, and .txt.

Figure 3 shows the results that answer to *PQ₁*, confirming that the programming files tend to churn more than other kinds of files. As shown in the figure, the average proportions of programming, configuration, and documentation are 69.97%, 14.15%, and 15.88%. The most common files are the source code files for Bash shell scripts, Python, and JavaScript files. We test the null hypothesis that “*the file churn for all different kinds of files is the same*”. Our result shows that there are

TABLE I
CORPUS SIZE OF TOKENS FOR EACH FILE EXTENSION.

File ext.	#Revisions	#Files	#Unique Token	#Token
.py	18,859	112,566	513,280	816,364,358
.php	20	21,723	11,535	5,330,917
.yaml	4,575	12,295	26,301	79,673,298
.rst	4,560	6,761	38,372	12,251,290
.pp	940	4,974	15,301	4,858,510
.yml	960	3,976	10,811	1,636,665
.sh	2,307	3,946	17,482	8,255,596
.json	1,137	2,977	20,563	6,275,269
.txt	2,105	2,961	2,283	474,107
.js	567	2,542	29,575	11,478,552
.html	512	1,772	6,588	703,987
.xml	253	1,333	4,386	597,631

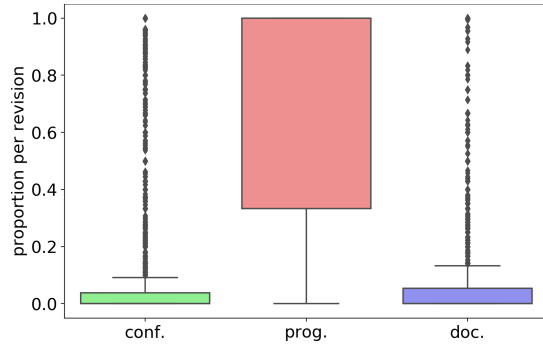


Fig. 3. File churn for each kind of files during a review. Programming file type have more churn compared to configuration and documentation types.

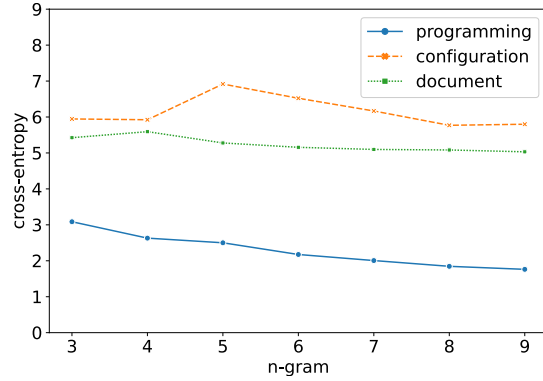


Fig. 4. Analysis of coding pattern changes (as n-grams) between the different types of files during a review.

statistically significant differences between two or more groups in our data ($p < .001$). This preliminary result suggests that during a code review, the intuitive focus is on code inspection, thus programming files will receive more changes.

Figure 4 shows the results relating to PQ_2 , where the programming related files are more likely to be repetitive. The entropy of programming file type is 3.086, and 1.762 for 3-grams and 9-grams, respectively. For configuration and documentation/other groups, the entropy changed from 5.946, and 5.425 to 5.799, and 5.031 for 3-grams and 9-grams. Considering the results of each group, the entropy of the programming decreased when n increases, while the entropy of other groups seems increased or stable. One reason for a higher entropy is that natural language may use different terms that vary across patches that may have different functions and descriptions. As a result, these two groups are less conformant than the programming group, which are constrained by the syntax of programming languages. Therefore, we focused on the programming group for our experiment.

Summary: Programming files (code) tend to contain more churn, and are more likely to conform to the coding pattern compared to configuration or documentation files.

TABLE II
PERCENTAGE OF LANGUAGE SYNTAX TOKEN.

	Separators	Operators	Keywords
.py files			
Pre-review	33.88%	5.64%	4.81%
Post-review	33.97%	5.65%	4.79%
.js files			
Pre-review	44.37%	8.40%	6.50%
Post-review	43.83%	8.09%	6.47%
.sh files			
Pre-review	-	5.67%	3.01%
Post-review	-	5.69%	2.99%
Reported Statistics from Rahman et al. [11]			
py	41.98%	6.42%	4.99%
js	47.21%	6.53%	6.87%

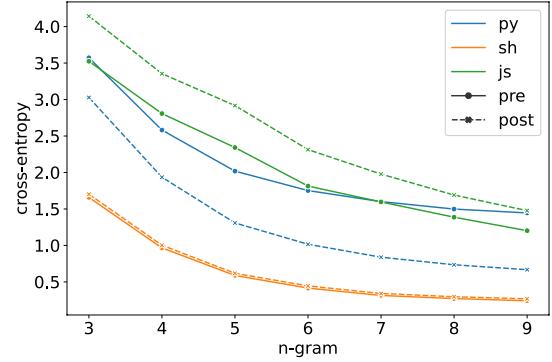


Fig. 5. *pre-review* vs. *post-review* versions based on programming type.

V. RQ₁: REVIEW PROCESS

Figure 5 shows the changed entropy of *post-review* compared to *pre-review*, that answer to RQ_1 , where the patch coding pattern conforms to the project after it has been reviewed. A general trend can be seen that the entropy of the three file types decreases as the length of the n-gram model increases, which follows the typical trend between n-gram entropy and its structural language. Looking at each graph individually in Figure 5, there is the typical natural trend in entropy of the *pre-review* and *post-review* patches of each file type. When we compare the two graphs, we see that *post-review* patches tend to have a lower entropy than *pre-review*. This is further highlighted by the python files, which tend to have lower entropy after the *post-review*. The result suggests that the python programmer concerns the conformance during a code review more than others.

Table II serves as a sanity check to compare our work to prior work in terms of the separators, operators, and keywords proportions. Although the percentages of separators for python files have a 10% difference to the Rahman et al. [11] study, we find that the *pre-review* and *post-review* percentages themselves are fairly consistent. Moreover, as shown in Table III, we conclude that there is no clear coding pattern between the separator, operator, and the keyword during a code review. One possible reason is that only a percentage of these tokens maybe not enough to find the coding pattern.

TABLE III
TOP 3 CHANGED SYNTAX TOKENS (ADDED AND REMOVED) FOR EACH FILE TYPE WHERE THE GREEN AND RED COLOR REPRESENT UNCHANGED AND CHANGED TOKEN, RESPECTIVELY, DURING A CODE REVIEW.

Top	Added Syntax		Operator		Keyword		Removed Syntax		Operator		Keyword	
	Separator token	percent	token	percent	token	percent	Separator token	percent	token	percent	token	percent
.py files												
1	.	20.38%	=	8.33%	def	1.30%	.	20.18%	=	8.47%	def	1.40%
2	,	13.19%	in	0.48%	if	0.84%	,	13.26%	in	0.59%	if	0.86%
3)	12.99%	-	0.31%	return	0.48%)	12.23%	%	0.26%	in	0.59%
.js files												
1	.	15.02%	=	7.34%	this	3.31%	(14.73%	=	6.49%	var	2.66%
2	(13.89%	<	1.80%	var	2.58%	.	14.50%	+	1.00%	this	2.55%
3)	13.47%	>	1.69%	return	1.76%)	14.48%	<	0.91%	return	1.88%
.sh files												
1			-	39.31%	{	2.59%			-	32.57%	then	3.47%
2			/	30.49%	}	2.28%			/	23.62%	{	3.17%
3			=	5.67%	then	2.28%			=	9.87%	if	3.17%

A key takeaway message is patches tend to contain natural coding pattern and more conformance after review, particularly for python files. Potential future work is to manually investigate whether the natural coding pattern correlates with complexity *post-review*. Similar to Campbell et al. [3], a syntax suggestion or highlighter tool during a review is feasible.

Summary: We provide evidence that the review process changes the coding pattern of the patch. Results show that the conformance of a patch after being reviewed, tend to be higher than a patch that was first submitted.

VI. RQ₂: PATCH DECISION

Figure 6 shows the cross-entropy of each programming file type based on whether it was accepted or not. Similar to RQ₁, the entropy trend and structural language are similar. It suggest that there is a difference in the coding pattern between accepted and abandoned patch groups. Confirming the related work of [5], we see that accepted patches are more conformant than the abandoned patches. At the file-level, the entropy of JavaScript file type is lower than Python file type in accepted patches, which is not the case for abandoned patches. This leads us to suspect that JavaScript developers concern the unknown factor which possibly is more important than conformance in the first stage of implementation.

One important takeaway from the results of RQ₂ is that patches themselves are sufficient to measure the coding pattern of the patch. This is instead of compared to analyzing the code in the project itself, which is performed by prior work. Analysis of the patches themselves should pave the way for potential code support tools and recommendation tools. Similar to RQ₁, a syntax suggestion or highlighter tool during a review is feasible, but for this case, we would like to predict the likelihood that a patch can be accepted or not.

Summary: The results suggest that there are differences in the coding pattern between accepted and abandoned patches groups. This is especially the case for the JavaScript code of the patch.

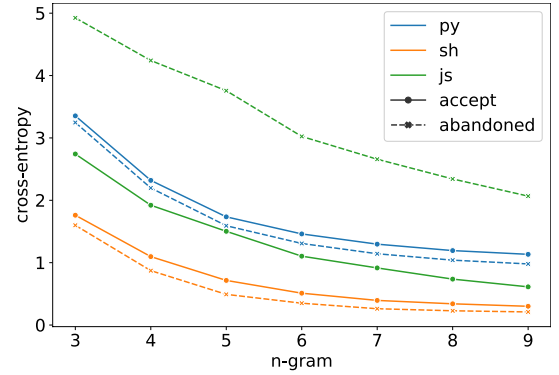


Fig. 6. Accepted vs. Abandoned Patches on programming language file type. The green line represents the .js file type that its accepted patches are more likely to conform when compared to the abandoned patches.

VII. SUMMARY AND FUTURE CHALLENGES

Our preliminary study confirms that reviewing code conforms code to repetitive coding patterns. Researchers and OSS project teams could use our results as motivation for exploring tool support or automatic detection of conformance coding patterns, while newcomers could increase the likelihood for acceptance by looking at prior submitted patches.

This work lays the groundwork to open new avenues such as exploring (1) whether or not there is a universal definition of conformance for source code and (2) whether conformance is related to developer individual experience or skill, and (3) whether or not conformance creates a more efficient review process, to name a few. Furthermore, we plan to explore how our approach contrasts and complements other techniques such as coding style checkers [4].

ACKNOWLEDGEMENT

This work supported by JSPS KAKENHI Grants JP20H05706, JP18H03221, JP20K19774, and JP18H04094.

REFERENCES

- [1] ANTLR, 2020. URL <https://github.com/antlr/antlr4>.
- [2] A. Bacchelli and C. Bird. Expectations, outcomes, and challenges of modern code review. In Proceedings of the 2013 International Conference on Software Engineering, pages 712–721, 2013.
- [3] H. Campbell, A. Hindle, and J. Amaral. Syntax errors just aren’t natural: Improving error reporting with language models. 05 2014.
- [4] D. Han, C. Ragkhitwetsagul, J. Krinke, M. Paixao, and G. Rosa. Does code review really remove coding convention violations? In 2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM), pages 43–53, 2020.
- [5] V. J. Hellendoorn, P. T. Devanbu, and A. Bacchelli. Will they like this? evaluating code contributions with language models. In Proceedings of the 12th Working Conference on Mining Software Repositories, pages 157–167, 2015.
- [6] T. Ishio, N. Maeda, K. Shibuya, and K. Inoue. Cloned buggy code detection in practice using normalized compression distance. In Proceedings of the 2018 IEEE International Conference on Software Maintenance and Evolution, pages 591–594, 2018.
- [7] S. McIntosh, B. Adams, M. Nagappan, and A. E. Hassan. Mining co-change information to understand when build changes are necessary. In 2014 IEEE International Conference on Software Maintenance and Evolution, pages 241–250, 2014.
- [8] MITLM, 2018. URL <https://github.com/mitlm/mitlm>.
- [9] OpenDev, 2020. URL <https://review.opendev.org/#/c/723716/>.
- [10] Reviewing proposed changes in a pull request, 2020. URL <https://help.github.com/en/github/collaborating-with-issues-and-pull-requests/reviewing-proposed-changes-in-a-pull-request>.
- [11] M. Rahman, D. Palani, and Peter C. Rigby. Natural software revisited. In Proceedings of the 41st International Conference on Software Engineering, pages 37–48, 2019.
- [12] R. Robbes and M. Lanza. Improving code completion with program history. Automated Software Engg., 17(2): 181–212, 2010.
- [13] Y. Ueda, A. Ihara, T. Ishio, and K. Matsumoto. Impact of coding style checker on code review - a case study on the openstack projects. In 2018 9th International Workshop on Empirical Software Engineering in Practice, pages 31–36, 12 2018.