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

An Empirical Study of the Impact of Modern Code Review Practices on Software Quality

Shane McIntosh · Yasutaka Kamei · Bram Adams · Ahmed E. Hassan

Author pre-print copy. The final publication is available at Springer via:
`http://dx.doi.org/10.1007/s10664-015-9381-9`

Abstract Software code review, i.e., the practice of having other team members critique changes to a software system, is a well-established best practice in both open source and proprietary software domains. Prior work has shown that formal code inspections tend to improve the quality of delivered software. However, the formal code inspection process mandates strict review criteria (e.g., in-person meetings and reviewer checklists) to ensure a base level of review quality, while the modern, lightweight code reviewing process does not. Although recent work explores the modern code review process, little is known about the relationship between modern code review practices and long-term software quality. Hence, in this paper, we study the relationship between post-release defects (a popular proxy for long-term software quality) and: (1) code review coverage, i.e., the proportion of changes that have been code reviewed, (2) code review participation, i.e., the degree of reviewer involvement in the code review process, and (3) code reviewer expertise, i.e., the level of domain-specific expertise of the code reviewers. Through a case study of the Qt, VTK, and ITK projects, we find that code review coverage, participation, and expertise share a significant link with software quality. Hence, our results empirically confirm the intuition that poorly-reviewed code has a negative impact on software quality in large systems using modern reviewing tools.

Shane McIntosh · Ahmed E. Hassan
Software Analysis and Intelligence Lab (SAIL)
Queen's University, Canada
E-mail: mcintosh@cs.queensu.ca, ahmed@cs.queensu.ca

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

Bram Adams
Lab on Maintenance, Construction, and Intelligence of Software (MCIS)
Polytechnique Montréal, Canada
E-mail: bram.adams@polymtl.ca

Keywords Code review · software quality

1 Introduction

Software code reviews are a well-documented best practice for software projects. In his seminal work, Fagan (1976) notes that formal design and code inspections with in-person meetings reduced the number of errors that are detected during the testing phase in small development teams. Rigby and Bird (2013) find that the modern code review processes that are adopted through a variety of reviewing mediums (e.g., mailing lists or the Gerrit web application[1]) tend to converge on a lightweight variant of the formal code inspections of the past, where the focus has shifted from defect-hunting to collaborative problem-solving. Nonetheless, Bacchelli and Bird (2013) find that one of the main motivations of modern code review still is to improve the quality of a change to the software.

Prior work indicates that formal design and code inspections can be an effective means of identifying defects so that they can be fixed early in the development cycle (Fagan, 1976). Tanaka *et al.* (1995) suggest that code inspections should be applied meticulously to each code change. Kemerer and Paulk (2009) indicate that student submissions tend to improve in quality when design and code inspections are introduced. However, there is little evidence of the long-term impact that modern, lightweight code review processes (which lack the rigid structure of code inspections) have on software quality in large systems.

In particular, to truly improve the quality of a set of proposed changes, reviewers must carefully consider the potential implications of the changes and engage in a discussion with the author. Under the formal code inspection model, time is allocated explicitly for the preparation and execution of in-person meetings, where the reviewers and the author discuss the proposed code changes (Fagan, 1976). Furthermore, reviewers are encouraged to follow a checklist to ensure that a base level of review quality is achieved. However, in the modern reviewing process, such strict reviewing criteria are not mandated (Rigby and Storey, 2011), and hence, reviews may not foster a sufficient amount of discussion between author and reviewers. Indeed, Microsoft developers complain that reviews often focus on minor logic errors rather than discussing deeper design issues (Bacchelli and Bird, 2013).

We hypothesize that a modern code review process that neglects to review a large proportion of code changes, that suffers from low reviewer participation, or that does not involve subject matter experts will likely have a negative impact on software quality. In other words:

> *If a large proportion of the code changes that are integrated during development are either: (1) omitted from the code review process (low review coverage), (2) have lax code review involvement (low review participation), or (3) do not include a subject matter expert (low expertise), then defect-prone code will permeate through to the released software product.*

[1] `https://code.google.com/p/gerrit/`

Tools that support the modern code reviewing process, such as Gerrit, explicitly link changes to a software system recorded in a Version Control System (VCS) to their respective code review. In this paper, we leverage these links to calculate code review coverage, participation, and expertise metrics, and add them to statistical regression models that are built to explain the incidence of *post-release defects* (i.e., defects in official releases of a software product), which is a popular proxy for software quality (Bird *et al.*, 2011; Hassan, 2009; Kamei *et al.*, 2013; Mockus and Weiss, 2000; Nagappan and Ball, 2007). Rather than using these models for defect prediction, we analyze the impact that code review metrics have on the models while controlling for a variety of metrics that are known to be good explainers of software quality. Through a case study of four releases of the large Qt, VTK, and ITK open source systems, we address the following three research questions:

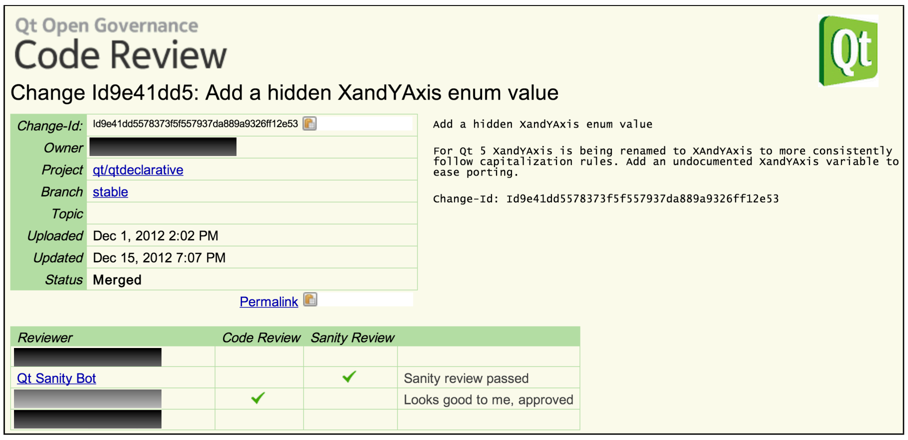
(RQ1) Is there a relationship between code review coverage and post-release defects?
We find that review coverage is negatively associated with the incidence of post-release defects in three of the four studied releases. However, it only provides a significant amount of explanatory power to two of the four studied releases, suggesting that review coverage alone does not guarantee a low incidence rate of post-release defects.

(RQ2) Is there a relationship between code review participation and post-release defects?
We find that the incidence of post-release defects is also associated with developer participation in code review. Review discussion metrics play a statistically significant role in the explanatory power of all of the studied systems.

(RQ3) Is there a relationship between code reviewer expertise and post-release defects?
Our models indicate that components with many changes that do not involve a subject matter expert in the authoring or reviewing process tend to be prone to post-release defects.

This paper is an extended version of our earlier work (McIntosh *et al.*, 2014). We extend the prior work to:

– Use contemporary regression modelling techniques (Harrell Jr., 2002) (Section 3) that:
  1. Relax the requirement of a linear relationship between post-release defect counts and explanatory variables, which enables a more accurate fit of the data.
  2. Filter away *redundant variables*, i.e., explanatory variables that may not be highly correlated with other explanatory variables, but do not provide a signal that is distinct with respect to the other explanatory variables.
  3. Allow us to analyze the stability of our models.
– Study the impact of reviewer expertise on software quality (RQ3).
– Include two additional review participation metrics (RQ2) and one additional review expertise metric (RQ3) that are not threshold-dependent, i.e., discussion speed (normalized by churn), discussion length (normalized by churn), and voter expertise.



Qt Open Governance

Code Review

Change Id9e41dd5: Add a hidden XandYAxis enum value

Change-Id:	Id9e41dd5578373f5f557937da889a9326ff12e53
Owner	[redacted]
Project	qt/qtdeclarative
Branch	stable
Topic	
Uploaded	Dec 1, 2012 2:02 PM
Updated	Dec 15, 2012 7:07 PM
Status	Merged

Permalink

```
Add a hidden XandYAxis enum value

For Qt 5 XandYAxis is being renamed to XAndYAxis to more consistently
follow capitalization rules. Add an undocumented XandYAxis variable to
ease porting.

Change-Id: Id9e41dd5578373f5f557937da889a9326ff12e53
```

Reviewer	Code Review	Sanity Review	
[redacted]			
Qt Sanity Bot		✓	Sanity review passed
[redacted]	✓		Looks good to me, approved
[redacted]			

Fig. 1 An example Gerrit code review.

1.1 Paper Organization

The remainder of the paper is organized as follows. Section 2 describes the Gerrit-driven code review process that is used by the studied systems. Section 3 describes the design of our case study, while Section 4 presents the results of our three research questions. Section 5 discloses the threats to the validity of our study. Section 6 surveys related work. Finally, Section 7 draws conclusions.

2 Gerrit Code Review

Gerrit[1] is a modern code review tool that facilitates a traceable code review process for `git`-based software projects (Bettenburg *et al.*, 2014). It tightly integrates with test automation and code integration tools. Authors upload *patches*, i.e., collections of proposed changes to a software system, to a Gerrit server. The set of reviewers are either: (1) invited by the author, (2) appointed automatically based on their expertise with the modified system components, or (3) self-selected by broadcasting a review request to a mailing list. Figure 1 shows an example code review in Gerrit that was uploaded on December 1st, 2012. Below, we use this figure to illustrate the role that reviewers and verifiers play in a code review.

2.1 Reviewers

Reviewers are responsible for critiquing the changes proposed within the patch by leaving comments for the author to address or discuss. The author can reply to comments or address them by producing a new revision of the patch for the reviewers to consider.

Reviewers can also give the changes proposed by a patch revision a *score*, which indicates: (1) agreement or disagreement with the proposed changes (positive or negative value), and (2) their level of confidence (1 or 2). The second column of the bottom-most table in Figure 1 shows that the change has been reviewed and the reviewer is in agreement with it (+). The text in the fourth column ("Looks good to me, approved") is displayed when the reviewer has a confidence level of two.

2.2 Verifiers

In addition to reviewers, verifiers are also invited to evaluate patches in the Gerrit system. Verifiers execute tests to ensure that patches: (1) truly fix the defect or add the feature that the authors claim to, and (2) do not cause regression of system functionality. Similar to reviewers, verifiers can provide comments to describe verification issues that they have encountered during testing. Furthermore, verifiers can also provide a score of 1 to indicate successful verification, and -1 to indicate failure.

While team personnel can act as verifiers, so too can Continuous Integration (CI) tools that automatically build and test patches. For example, CI build and testing jobs can be automatically launched each time a new review request or patch revision is uploaded to Gerrit. The reports generated by these CI jobs can be automatically appended as a verification report to the code review discussion. The third column of the bottom-most table in Figure 1 shows that the "Qt Sanity Bot" has successfully verified the change.

2.3 Automated integration

Gerrit allows teams to codify code review and verification criteria that must be satisfied before changes are integrated into upstream VCS repositories. For example, a team policy may specify that at least one reviewer and one verifier should provide positive scores prior to integration. Once the criteria are satisfied, patches are automatically integrated into upstream repositories. The "Merged" status shown in the upper-most table of Figure 1 indicates that the proposed changes have been integrated.

3 Case Study Design

In this section, we present our rationale for selecting our research questions, describe the studied systems, and present our data extraction and analysis approaches.

3.1 Research Questions

Broadly speaking, the main goal of this paper is to study whether lax involvement that may creep into the modern, lightweight code review process has a negative impact on software quality. We focus on three aspects of modern code review practices that we believe may have an impact on software quality (i.e., coverage, participation, and

Table 1 Overview of the studied systems. Those to the left of the double line satisfy our criteria for analysis.

	Qt		**VTK**	**ITK**	**Android**	**LibreOffice**
Version	5.0	5.1	5.10	4.3	4.0.4	4.0
Tag name	v5.0.0	v5.1.0	v5.10.0	v4.3.0	4.0.4	4.0.0
Size (LOC)	5,560,317	5,187,788	1,921,850	1,123,614	18,247,796	4,789,039
Components w/ defects	254	187	15	24	-	-
Components total	1,339	1,337	170	218	-	-
Defective rate	19%	14%	9%	11%	-	-
Commits w/ reviews	10,003	6,795	554	344	1,727	1,679
Commits total	10,163	7,106	1,431	352	80,398	11,988
Review rate	98%	96%	39%	98%	2%	14%
# Authors	435	422	55	41	-	-
# Reviewers	358	348	45	37	-	-

expertise). We leave the exploration of other aspects of code review practices to future work.

In order to evaluate our conjecture about lax code review practices, we formulate the following three research questions:

(RQ1) Is there a relationship between code review coverage and post-release defects?
Tanaka *et al.* (1995) suggest that a software team should meticulously review each change to the source code to ensure that quality standards are met. In more recent work, Kemerer and Paulk (2009) find that design and code inspections have a measurable impact on the defect density of student submissions at the Software Engineering Institute (SEI). While these findings suggest that there is a relationship between code review coverage and software quality, the scale of such a relationship has remained largely unexplored in large software systems using modern code reviewing tools.

(RQ2) Is there a relationship between code review participation and post-release defects?
To truly have an impact on software quality, developers must invest in the code reviewing process. In other words, if developers are simply approving code changes without discussing them, the code review process likely provides little value. Hence, we set out to study the relationship between developer participation in code reviews and software quality.

(RQ3) Is there a relationship between code reviewer expertise and post-release defects?
Changes produced by novice developers are more likely to introduce defects than those produced by subject matter experts (Mockus and Weiss, 2000). However, a change produced by a novice can be improved by soliciting feedback from subject matter experts during the code review process. Hence, we set out to study whether changes that are developed by personnel who lack subject matter expertise have an impact on software quality if they are not reviewed by a subject matter expert.

3.2 Studied Systems

In order to address our research questions, we perform a case study on large, successful, and rapidly-evolving open source systems with globally distributed development teams. In selecting the subject systems, we identified two important criteria that needed to be satisfied:

Criterion 1: Reviewing Policy – We want to study systems that have made a serious investment in code reviewing. Hence, we only study systems where a large number of the integrated patches have been reviewed.

Criterion 2: Traceability – The code review process for a subject system must be *traceable*, i.e., it should be reasonably straightforward to connect a large proportion of the integrated patches to the associated code reviews. Without a traceable code review process, review coverage and participation metrics cannot be calculated, and hence, we cannot perform our analysis.

To satisfy the traceability criterion, we focus on software systems that use the Gerrit code review tool. We began our study with five subject systems, however after preprocessing the data, we found that only 2% of *Android* and 14% of *LibreOffice* changes could be linked to reviews, so both systems had to be removed from our analysis (Criterion 1).

Table 1 shows that the *Qt*, *VTK*, and *ITK* systems satisfied our criteria for analysis. *Qt* is a cross-platform application framework whose development is supported by the Digia corporation, however welcomes contributions from the community-at-large.[2] The *Visualization ToolKit* (VTK) is used to generate 3D computer graphics and process images.[3] The *Insight segmentation and registration ToolKit* (ITK) provides a suite of tools for in-depth image analysis.[4]

3.3 Data Extraction

In order to evaluate the impact that code review coverage, participation, and expertise have on software quality, we extract code review data from the Gerrit review databases of the studied systems, and link the review data to the integrated patches recorded in the corresponding VCSs.

Figure 2 shows that our data extraction approach is broken down into three steps: (1) extract review data from the Gerrit review database, (2) extract Gerrit change IDs from the VCS commits, and (3) calculate version control metrics. We briefly describe each step of our approach below.

3.3.1 Extract reviews

Our analysis is based on the Qt code reviews dataset collected by Hamasaki *et al.* (2013). The dataset describes each review, the involved personnel, and the details

[2]`http://qt.digia.com/`

[3]`http://vtk.org/`

[4]`http://itk.org/`

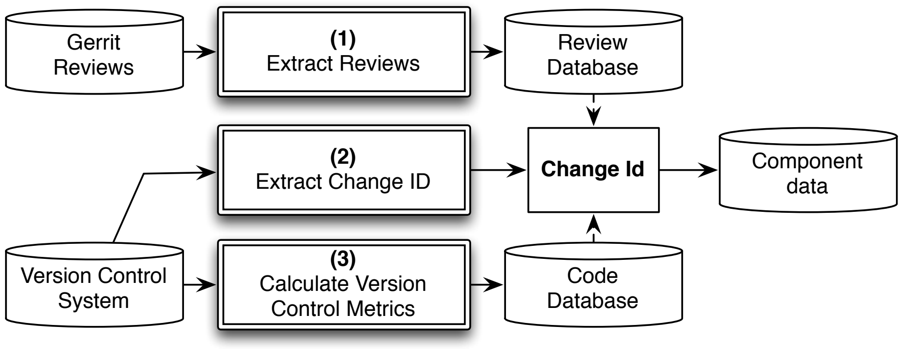


Fig. 2 Overview of our data extraction approach.

of the review discussions. We expand the dataset to include the reviews from the VTK and ITK systems, as well as those reviews that occurred during the more recent development cycle of Qt 5.1. To do so, we use a modified version of the GerritMiner scripts provided by Mukadam *et al.* (2013).

3.3.2 Extract change ID

Each review in a Gerrit database is uniquely identified by an alpha-numeric hash code called a *change ID*. When a review has satisfied project-specific criteria, it is automatically integrated into the upstream VCS (*cf.* Section 2). For traceability purposes, the commit message of the automatically integrated patch contains the change ID. We extract the change ID from commit messages in order to automatically connect patches in the VCS with the associated code review process data. To facilitate future work, we have made the code and review databases available online.[5]

3.3.3 Calculate version control metrics

Prior work has found that several types of metrics have a relationship with defect-proneness. Since we aim to investigate the impact that code reviewing has on defect-proneness, we control for the three most common families of metrics that are known to have a relationship with defect-proneness (Shihab *et al.*, 2011; Hassan, 2009; Bird *et al.*, 2011). Table 2 provides a brief description and the motivating rationale for each of the studied baseline metrics. Similar to prior work (Nagappan *et al.*, 2006; Bird *et al.*, 2011), we measure each metric at the component (i.e., directory) level.

We focus our analysis on the development activity on the `release` branches of each studied system, i.e., activity that: (1) occurred on the main development branch before the `release` branch was cut, (2) occurred on the `release` branch itself, and (3) originated on other branches, but has been merged into the `release` branch. Prior to a release, the integration of changes on a `release` branch is more strictly

[5] `http://sailhome.cs.queensu.ca/replication/reviewing_quality_ext/`

Table 2 A taxonomy of the considered baseline component metrics.

	Metric	Description	Rationale
Product	Size	Number of lines of code.	Large components are more likely to be defect-prone (Koru *et al.*, 2009).
	Complexity	The McCabe cyclomatic complexity.	More complex components are likely more defect-prone (Menzies *et al.*, 2002).
Process	Prior defects	Number of defects fixed prior to release.	Defects may linger in components that were recently defective (Graves *et al.*, 2000).
	Churn	Sum of added and removed lines of code.	Components that have undergone a lot of change are likely defect-prone (Nagappan and Ball, 2005, 2007).
	Change entropy	A measure of the distribution of changes among files.	Components where changes are spread among several files are likely defect-prone (Hassan, 2009).
Human Factors	Total authors	Number of unique authors.	Components with many unique authors likely lack strong ownership, which in turn may lead to more defects (Bird *et al.*, 2011; Graves *et al.*, 2000).
	Minor authors	Number of unique authors who have contributed less than 5% of the changes.	Developers who make few changes to a component may lack the expertise required to perform the change in a defect-free manner (Bird *et al.*, 2011). Hence, components with many minor contributors are likely more defect-prone.
	Major authors	Number of unique authors who have contributed at least 5% of the changes.	Similarly, components with a large number of major contributors, i.e., those with component-specific expertise, are less likely to be defect-prone (Bird *et al.*, 2011).
	Author ownership	The proportion of changes contributed by the author who made the most changes.	Components with a highly active component owner are less likely to be defect-prone (Bird *et al.*, 2011).

controlled than a typical development branch to ensure that only the appropriately-triaged changes will appear in the upcoming release. Moreover, changes that land on a release branch after a release are also strictly controlled to ensure that only high-priority fixes land in maintenance releases. In other words, the changes that we study correspond to the development and maintenance of official software releases.

To determine whether a change fixes a defect, we search the VCS commit messages for co-occurrences of defect identifiers with keywords like “bug”, “fix”, “defect”, or “patch”. A similar approach was used to determine defect-fixing and defect-inducing changes in other work (Mockus and Votta, 2000; Hassan, 2008; Kim *et al.*, 2008; Kamei *et al.*, 2013). Similar to prior work (Kamei *et al.*, 2010), we define post-release defects as those with fixes recorded in the six-month period after the release date.

Product metrics. Product metrics measure the source code of a system at the time of a release. It is common practice to preserve the released versions of the source code of a software system in the VCS using tags. In order to calculate product metrics for the studied releases, we first extract the released versions of the source code by "checking out" those tags from the VCS.

We measure the size and complexity of each component (i.e., directory) as described below. We measure the size of a component by aggregating the number of lines of code in each of its files. We use McCabe's cyclomatic complexity (McCabe, 1976) (calculated using Scitools Understand[6]) to measure the complexity of a file. To measure the complexity of a component, we take the sum of the complexity of each file within it. Finally, since complexity measures are often highly correlated with size, we divide the complexity of each component by its size to reduce the influence of size on complexity measures. A similar approach was used in prior work (Kamei *et al.*, 2010).

Process metrics. Process metrics measure the change activity that occurred during the development of a new release. Process metrics must be calculated with respect to a time period and a development branch. Again, similar to prior work (Kamei *et al.*, 2010), we measure process metrics using the six-month period prior to each release date on the `release` branch.

We use prior defects, churn, and change entropy to measure the change process. We count the number of defects fixed in a component prior to a release by using the same pattern-based approach that we use to identify post-release defects. Churn measures the total number of lines added to and removed from a component prior to release. As suggested by Nagappan and Ball (2005), we divide the churn of each component by its size. Change entropy measures how the complexity of a change process is distributed across files (Hassan, 2009). We measure the change entropy of each component individually, i.e., we calculate how evenly spread out the changes to a component's files before the release are for each component in isolation. Similar to Hassan (2009), we use the Shannon (1948) entropy normalized by the maximum entropy for a component c as described below:

$$H(c) = \frac{-\sum_{k=1}^{n}(p_k * log_2 p_k)}{log_2 n}, \tag{1}$$

where n is the number of files in component c, and p_k is the proportion of the changes to c that occur in file k. Such a normalized entropy allows one to compare the $H(c)$ between components of different sizes with different numbers of files.

Human factors. Human factor metrics measure developer expertise and code ownership. Similar to process metrics, human factor metrics must also be calculated with respect to a time period. We again adopt the six-month period prior to each release date as the window for metric calculation.

Table 2 shows that we adopt the suite of ownership metrics proposed by Bird *et al.* (2011). Total authors is the number of authors that contribute to a component. Minor authors is the number of authors that contribute fewer than 5% of the commits to a component. Major authors is the number of authors that contribute at least 5% of the commits to a component. Author ownership is the proportion of commits that the most active contributor to a component has made.

[6] `http://www.scitools.com/documents/metricsList.php?#Cyclomatic`

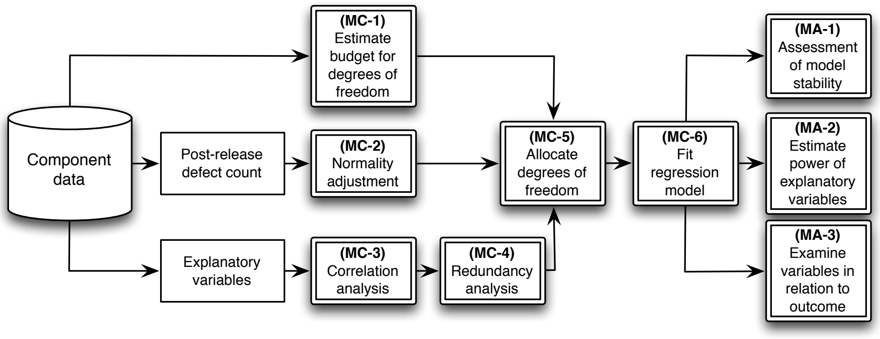

(MC-1) Estimate budget for degrees of freedom
Component data
Post-release defect count
(MC-2) Normality adjustment
Explanatory variables
(MC-3) Correlation analysis
(MC-4) Redundancy analysis
(MC-5) Allocate degrees of freedom
(MC-6) Fit regression model
(MA-1) Assessment of model stability
(MA-2) Estimate power of explanatory variables
(MA-3) Examine variables in relation to outcome

Fig. 3 Overview of our Model Construction (MC) and Model Analysis (MA) approach.

3.4 Model Construction

We build regression models to explain the incidence of post-release defects detected in the components of the studied systems. A regression model fits a curve of the form $y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_n x_n$ to the data, where y is the dependent variable and each x_i is an explanatory variable. In our models, the dependent variable is the post-release defect count and the explanatory variables are the set of metrics outlined in Tables 2 and 3.

We adopt the model construction and analysis approach of Harrell Jr. (2002). These techniques relax linearity assumptions between explanatory and dependent variables, allowing nonlinear relationships to be modelled more accurately, while being mindful of the potential for *overfitting*, i.e., constructing a model that is too specialized for the dataset on which the model was trained that it would not apply to other datasets. We use the R implementation (R Core Team, 2013) of the techniques described by Harrell Jr. (2002) provided by the `rms` package (Harrell Jr., 2014).

Figure 3 provides an overview of the six steps in our model construction approach. Figure 14 in Appendix A provides the R code for each of these steps. We describe each step in the approach below.

(MC-1) Estimate Budget for Degrees of Freedom

When constructing explanatory or predictive models, a critical concern is that of overfitting. An overfit model will exaggerate or dismiss relationships between the dependent and explanatory variables based on characteristics of the dataset from which it was built.

Overfitting may creep into models that use more degrees of freedom (e.g., explanatory variables) than a dataset can support. Hence, it is pragmatic to calculate a budget of degrees of freedom that a dataset can support before attempting to fit a model. As suggested by Harrell Jr. *et al.* (1984, 1985), we budget $\frac{n}{15}$ degrees of freedom for our defect models, where n is the number of rows (i.e., components) in the dataset.

(MC-2) Normality Adjustment

We fit our regression models using the Ordinary Least Squares (OLS) technique using the `ols` function provided by the `rms` package. OLS expects that the dependent variable is normally distributed. Since software engineering data is often skewed, we analyze the distribution of post-release defect counts in each studied system prior to fitting our models. If we find that the distribution is skewed, we apply a log transformation $[ln(x+1)]$ to lessen the skew, and better fit the assumptions of the OLS technique.

(MC-3) Correlation Analysis

Prior to building our models, we check for explanatory variables that are highly correlated with one another using Spearman rank correlation tests (ρ). We choose a rank correlation instead of other types of correlation (e.g., Pearson) because rank correlation is resilient to data that is not normally distributed.

We use a variable clustering analysis to construct a hierarchical overview of the correlation among the explanatory variables (Sarle, 1990). For sub-hierarchies of explanatory variables with correlation $|\rho| > 0.7$, we select only one variable from the sub-hierarchy for inclusion in our models.

(MC-4) Redundancy Analysis

Correlation analysis reduces collinearity among the explanatory variables, but it may not detect all of the *redundant variables*, i.e., variables that do not have a unique signal from the other explanatory variables. Redundant variables in an explanatory model will interfere with each other, distorting the modelled relationship between the explanatory and dependent variables. We, therefore, remove redundant variables prior to constructing our defect models.

In order to detect redundant variables, we fit preliminary models that explain each explanatory variable using the other explanatory variables. We use the R^2 value of the preliminary models to measure how well each explanatory variable is explained by the other explanatory variables.

We use the implementation of this approach provided by the `redun` function in the `rms` package. The function builds preliminary models for each explanatory variable. The explanatory variable that is most well-explained by the other explanatory variables is iteratively dropped until either: (1) no preliminary model achieves an R^2 above a cutoff threshold (for this paper, we use the default threshold of 0.9), or (2) removing a variable would make a previously dropped variable no longer explainable, i.e., its preliminary model will no longer achieve an R^2 exceeding the threshold.

(MC-5) Allocate Degrees of Freedom

After removing highly correlated and redundant variables, we must decide how to spend our budgeted degrees of freedom most effectively. Specifically, we are most concerned with identifying the explanatory variables that would benefit most from

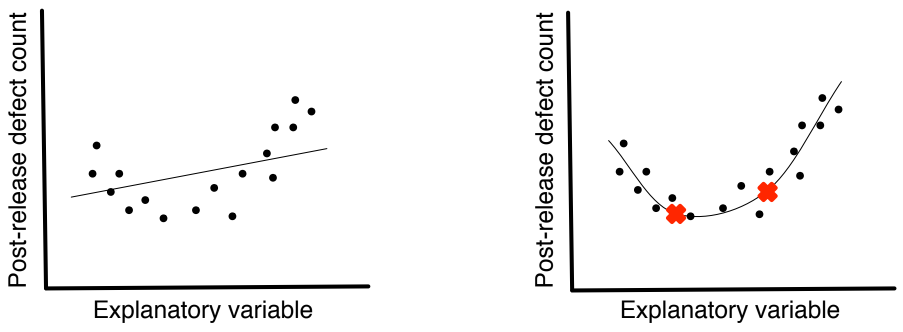


Fig. 4 An example of a non-monotonic relationship that would benefit from knots. The plot on the left shows that a linear fit would not fit the relationship well, whereas the plot on the right has been allocated two knots (red 'X' shapes), and allows each region between the knots to have a different fit to improve the accuracy of the overall fit.

knots, i.e., changes in the direction of the relationship between explanatory and dependent variables. Figure 4 shows an example of a non-monotonic relationship between an explanatory variable and the dependent one. The plot on the left shows that a linear fit would under-represent many of the points in the base and tails of the curve. Conversely, the plot on the right shows that by allocating two knots (red 'X' shapes), and fitting curves through them, we can more appropriately approximate the relationship.

Unfortunately, knots cannot be allocated without careful planning. Each knot costs an additional degree of freedom because it introduces a new term in the model formula. For example, Figure 4 would require three model terms to define the shape of the curve: (1) one to the left of the leftmost knot, (2) one between the two knots, and (3) one to the right of the rightmost knot. We would therefore like to allocate knots to the explanatory variables with the most potential for sharing non-monotonic relationship with the dependent variable.

To measure the potential for non-monotonicity in the relationship between dependent and explanatory variables, we calculate the Spearman multiple ρ^2 between the dependent variable y and linear and quadratic forms of each explanatory variable (x_i, x_i^2). A large Spearman multiple ρ^2 score indicates that there is a strong nonlinear or non-monotonic relationship between x_i and y that would benefit from being allocated additional degrees of freedom.

We use the `spearman2` function in the `rms` package to calculate the Spearman multiple ρ^2 between the dependent and explanatory variables. If we decided that we must log transform the dependent variable in the MC-2 step, we do so prior to measuring the Spearman multiple ρ^2. All of the explanatory variables that survive our correlation (MC-3) and redundancy analyses (MC-4) are allocated at least one degree of freedom (a linear fit). Taking the budgeted degrees of freedom into account, we allocate the degrees of freedom to explanatory variables according to their Spearman multiple ρ^2 values, i.e., variables with larger ρ^2 values are allocated more degrees of freedom than variables with smaller ρ^2 values.

(MC-6) Fit Regression Model

Finally, after selecting appropriate explanatory variables, log transforming the dependent variable (if necessary according to MC-2), and allocating budgeted degrees of freedom to the explanatory variables that will yield the most benefit from them, we fit our regression models to the data. We use *restricted cubic splines* to fit the budgeted number of knots to the explanatory variables that we allocated additional degrees of freedom to. Cubic splines fit cubic forms of an explanatory variable in order to join the different model terms. However, in an unrestricted form, cubic splines tend to fit poorly in the tails, i.e., before the first knot and after the last one, due to the curling nature of a cubic curve. Restricted cubic splines force the tails of the relationship to be linear, and tend to better fit nonlinear relationships (Harrell Jr., 2002).

3.5 Model Analysis

After building regression models, we evaluate the goodness of fit using the *Adjusted* R^2 (Hastie *et al.*, 2009). Unlike the unadjusted R^2, the adjusted R^2 accounts for the bias of introducing additional degrees of freedom by penalizing models for each degree of freedom spent.

As shown in Figure 3, we perform three model analysis steps in order to: (1) study the stability of our models, (2) estimate the impact of each explanatory variable on model performance, and (3) study the relationship between each explanatory variable while controlling for the others. Figure 15 in Appendix A provides the R code for each of these steps. We describe each model analysis step below.

(MA-1) Assessment of Model Stability

While the adjusted R^2 of the model gives an impression of how well the model has fit the dataset, it may overestimate the performance of the model if it is overfit. We take performance overestimation into account by subtracting the bootstrap-calculated *optimism* (Efron, 1986) from initial adjusted R^2 estimates. The optimism of the adjusted R^2 is calculated as follows:

1. From the original dataset with n components, select a *bootstrap sample*, i.e., a new sample of n components with replacement.
2. In the bootstrap sample, fit a model using the same allocation of degrees of freedom as was used in the original dataset.
3. Apply the model built from the bootstrap sample on the bootstrap and original datasets, calculating the adjusted R^2 in each.
4. The optimism is the difference in the adjusted R^2 of the bootstrap sample and the original sample.

The above process is repeated 1,000 times and the average (mean) optimism is calculated. Finally, we obtain the optimism-reduced adjusted R^2 by subtracting the average optimism from the original adjusted R^2. The smaller the optimism values, the more stable that the original model fit is.

Unlike k-fold cross-validation, the model fit that is validated using the above bootstrap-derived technique is the one fit using the entire dataset. k-fold cross-validation splits the data into k equal parts, using $k-1$ parts for fitting the model, setting aside 1 fold for testing. The process is repeated k times, using a different part for testing each time. Notice, however, that models are fit using $k-1$ folds (i.e., a subset) of the dataset. Models fit using the full dataset are not directly tested when using k-fold cross-validation.

(MA-2) Estimate Power of Explanatory Variables

We would like to estimate the impact that each explanatory variable has on our model performance. In our prior work (McIntosh *et al.*, 2014), we evaluated the impact of each explanatory variable using the χ^2 maximum likelihood tests of a "drop one" approach (Chambers and Hastie, 1992). This test measures the impact of an explanatory variable on a model by measuring the difference in the performance of models built using: (1) all explanatory variables (the full model), and (2) all explanatory variables except for the one under test (the dropped model). A χ^2 test is applied to the resulting values to detect whether each explanatory variable improves model performance to a statistically significant degree.

However, in this paper, explanatory variables that have been allocated several degrees of freedom are represented with several model terms instead of just one. To control for the effect of multiple terms, we jointly test the set of explanatory variable model terms for each variable using Wald χ^2 maximum likelihood (a.k.a., "chunk") tests. The larger the Wald χ^2 value, the larger the impact that a particular explanatory variable has on a model's performance. We report both the raw Wald χ^2 value and its significance level according to its p-value.

(MA-3) Examine Explanatory Variables in Relation to the Outcome

Finally, we would like to study the relationship that each modelled reviewing metric shares with the post-release defect count. While the coefficients of the model terms in linear models can give a general impression of the impact that an explanatory variable has on the outcome, each explanatory variable in our models may be represented by several model terms. In order to account for the impact of all of the model terms associated with an explanatory variable, we plot the change in the estimated number of post-release defects against the change in each reviewing metric while holding the other explanatory variables constant at their median values using the `Predict` function in the `rms` package (Harrell Jr., 2014). The plot will follow the relationship as it changes directions at knot locations (*cf.* MC-6). Furthermore, change in estimated value approximates the impact on software quality that the accompanying change in the reviewing metric will have. The plots also show the 95% confidence intervals calculated based on the 1,000 previously executed bootstrap iterations (*cf.* MA-1).

Table 3 A taxonomy of the considered code review metrics.

	Metric	Description	Rationale
Coverage (RQ1)	Proportion of reviewed changes	The proportion of changes that have been reviewed in the past.	Since code review will likely catch defects, components where changes are most often reviewed are less likely to contain defects.
	Proportion of reviewed churn	The proportion of churn that has been reviewed in the past.	Despite the defect-inducing nature of code churn, code review should have a preventative impact on defect-proneness. Hence, we expect that the larger the proportion of code churn that has been reviewed, the less defect prone a module will be.
Participation (RQ2)	Number of self-approved changes	The proportion of changes to a component that are only approved for integration by the original author.	By submitting a review request, the original author already believes that the code is ready for integration. Hence, changes that are only approved by the original author have essentially not been reviewed.
	Number of hastily-reviewed changes	The proportion of changes that are approved for integration at a rate that is faster than 200 lines per hour.	Prior work has shown that when developers review more than 200 lines of code per hour, they are more likely to let lower quality source code slip through (Kemerer and Paulk, 2009). Hence, components with many changes that are approved at a rate faster than 200 lines per hour are more likely to be defect-prone.
	Number of changes without discussion	The proportion of changes to a component that are not discussed.	Components with many changes that are approved for integration without critical discussion are likely to be defect-prone.
	Typical review window*	The length of time between the creation of a review request and its final approval for integration, normalized by the size of the change (churn).	Components with shorter review windows may not be spending enough time carefully analyzing the implications of a change, and hence may be more defect-prone.
	Typical discussion length*	The length of the review discussion (i.e., # non-automated comments), normalized by the size of the change (churn).	Components with many short review discussions may not be deriving value from the review process, and hence may be more defect-prone.
Expertise (RQ3)	Number of changes that do not involve a subject matter expert*	The proportion of changes to a component that are not authored by nor reviewed by a subject matter expert, i.e., a major author.	Components with many changes that do not incorporate a subject matter expert are more likely to be defect-prone.
	Typical voter expertise*	The percentage of prior changes to a component that each voter has either authored or voted on.	Components with a high degree of voter expertise are likely less defect-prone.

*New metric that did not appear in the earlier version of this paper (McIntosh *et al.*, 2014).

4 Case Study Results

In this section, we present the results of our case study with respect to our three research questions. For each question, we discuss: (a) the metrics that we use to measure the reviewing property, (b) our model construction procedure, and (c) the model analysis results.

Table 4 Descriptive statistics of the studied review coverage metrics.

		Qt		**VTK**	**ITK**
		5.0	5.1	5.10	4.3
Rev'd changes	Minimum	0.86	0.00	0.00	0.00
	1st Quartile.	1.00	1.00	0.00	1.00
	Median	1.00	1.00	0.00	1.00
	Mean	0.99	0.98	0.27	0.99
	3rd Quartile	1.00	1.00	0.50	1.00
	Maximum	1.00	1.00	1.00	1.00
Rev'd churn	Minimum	0.50	0.00	0.00	0.00
	1st Quartile.	1.00	1.00	0.00	1.00
	Median	1.00	1.00	0.00	1.00
	Mean	0.99	0.98	0.15	0.99
	3rd Quartile	1.00	1.00	0.06	1.00
	Maximum	1.00	1.00	1.00	1.00

(RQ1) Is there a relationship between code review coverage and post-release defects?

Intuitively, one would hope that higher rates of code review coverage will lead to fewer incidences of post-release defects. To investigate this intuition, we use the code review coverage metrics described in Table 3 in regression models with the baseline metrics of Table 2.

(RQ1-a) Coverage metrics

Table 4 provides descriptive statistics of the studied review coverage metrics. The *proportion of reviewed changes* is the proportion of changes committed to a component that are associated with code reviews. Similarly, *proportion of reviewed churn* is the proportion of the churn of a component that is associated with code reviews. For this research question, we set both the proportion of reviewed changes and the proportion of reviewed churn to 1 for components that have not changed during the pre-release time period.

(RQ1-b) Model construction

In this section, we describe the outcome of the model construction steps outlined in Figure 3.

(MC-1) Estimate budget of degrees of freedom. Table 5 shows that our data can support between 11 ($\frac{170}{15}$ in VTK) and 89 ($\frac{1,339}{15}$ in Qt 5.0) degrees of freedom. We can therefore apply knots more liberally to the explanatory variables in the larger Qt datasets than in the smaller VTK and ITK ones.

(MC-2) Normality adjustment. Analysis of the post-release defect counts of the studied systems reveals that the values are right-skewed in the larger Qt datasets. To counter the skew, we log-transform the post-release defect counts in the Qt datasets.

Table 5 Review coverage model statistics (RQ1).

		Qt				**VTK**		**ITK**	
		5.0		5.1		5.10		4.3	
Adjusted R^2		0.64		0.67		0.39		0.44	
Optimism-reduced adjusted R^2		0.62		0.65		0.20		0.22	
Wald χ^2		$2{,}360^{***}$		$2{,}715^{***}$		118^{***}		177^{***}	
Budgeted Degrees of Freedom		89		89		11		14	
Degrees of Freedom Spent		17		24		9		9	
		Overall	Nonlinear	Overall	Nonlinear	Overall	Nonlinear	Overall	Nonlinear
Size	D.F.	4	3	4	3	1	-	1	-
	χ^2	85^{***}	57^{***}	78^{***}	73^{***}	4^{*}		15^{***}	
Complexity	D.F.	2	1	2	1	1	-	1	-
	χ^2	7^{*}	5^{*}	7^{*}	7^{*}	1°		$< 1^{\circ}$	
Prior defects	D.F.	2	1	2	1	2	1	3	2
	χ^2	48^{***}	32^{***}	81^{***}	13^{***}	106^{***}	84^{***}	51^{***}	26^{***}
Churn	D.F.	1	-	1	-	1	-	1	-
	χ^2	$< 1^{\circ}$		$< 1^{\circ}$		$< 1^{\circ}$		$< 1^{\circ}$	
Change entropy	D.F.	2	1	4	3	1	-	†	
	χ^2	11^{**}	6^{*}	34^{***}	30^{***}	$< 1^{\circ}$			
Total authors	D.F.	3	2	†		1	-	†	
	χ^2	64^{***}	13^{***}			52^{***}			
Minor authors	D.F.	‡		4	3	‡		1	-
	χ^2			55^{***}	40^{***}			25^{***}	
Major authors	D.F.	†		†		†		†	
	χ^2								
Author ownership	D.F.	2	1	3	2	1	-	1	-
	χ^2	4°	2°	4°	4°	3°		$< 1^{\circ}$	
Reviewed changes	D.F.	1	-	4	3	1	-	1	-
	χ^2	1°		84^{***}	83^{***}	12^{***}		$< 1^{\circ}$	
Reviewed churn	D.F.	†		†		†		†	
	χ^2								

Discarded during:
† Variable clustering analysis ($|\rho| \geq 0.7$)
‡ Redundant variable analysis ($R^2 \geq 0.9$)
Statistical significance of explanatory power according to Wald χ^2 likelihood ratio test:
$\circ$ $p \geq 0.05$; * $p < 0.05$; ** $p < 0.01$; *** $p < 0.001$
- Nonlinear degrees of freedom not allocated

On the other hand, the skew is not large enough to be of concern in the smaller VTK and ITK datasets, so we do not apply a transformation to those projects.

(MC-3) Correlation analysis. Figure 5 shows the hierarchically clustered Spearman ρ values in the Qt 5.0 dataset. The solid horizontal lines indicate the correlation value of the two metrics that are connected by the vertical branches that descend from it. The gray dashed line indicates our cutoff value ($|\rho| = 0.7$). Similar correlation values were observed in the other studied systems. To conserve space, we provide online access to the figures for the other studied systems.[7]

Analysis of the clustered variable correlations reveals that the proportion of reviewed churn is too highly correlated with the proportion of reviewed changes to include both metrics. Similarly, the number of major authors is too highly correlated with the total number of authors. We selected the proportion of reviewed changes and the total number of authors for our models because they are the simpler of the metric pairs to compute. For the sake of completeness, we analyzed models that use

[7] `http://sailhome.cs.queensu.ca/replication/reviewing_quality_ext/`

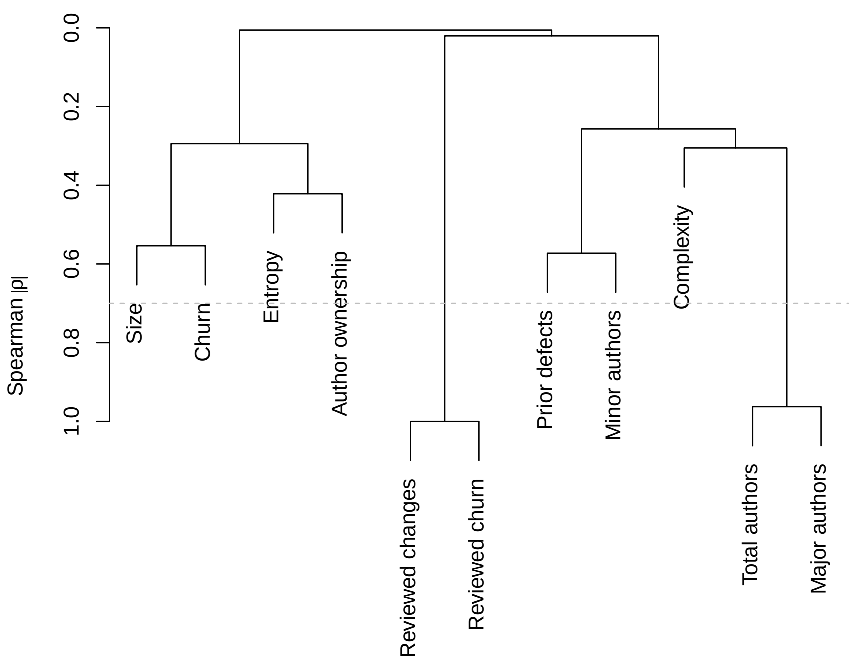


Fig. 5 Hierarchical clustering of variables according to Spearman's $|\rho|$ in Qt 5.0 (RQ1).

the proportion of reviewed churn instead of the proportion of reviewed changes, as well as models that use the number of major authors instead of the total number of authors, and found that neither change of metric had a discernible impact on model performance.

(MC-4) Redundancy analysis. Table 5 shows that the number of minor authors is a redundant variable in the Qt 5.0 and VTK datasets. The number of minor authors is well-explained by the other metrics ($R^2_{Qt5.0} = 0.99$, $R^2_{VTK} = 0.98$). Since this metric is not likely to add additional explanatory power, we exclude it from these models.

(MC-5) Allocate degrees of freedom. Figure 6 shows the Spearman multiple ρ^2 of the post-release defect count with each explanatory variable in Qt 5.0. Variables that show larger Spearman multiple ρ^2 values have more potential for sharing a non-monotonic relationship with the post-release defect count, and hence, would benefit most from additional degrees of freedom. To conserve space, we only show the Spearman multiple ρ^2 figure for the Qt 5.0 system, and provide the figures for the other studied systems online.[8]

By observing the rough clustering of variables according to the Spearman multiple ρ^2 values, we split the explanatory variables of Figure 6 into three groups. The

[8]`http://sailhome.cs.queensu.ca/replication/reviewing_quality_ext/`

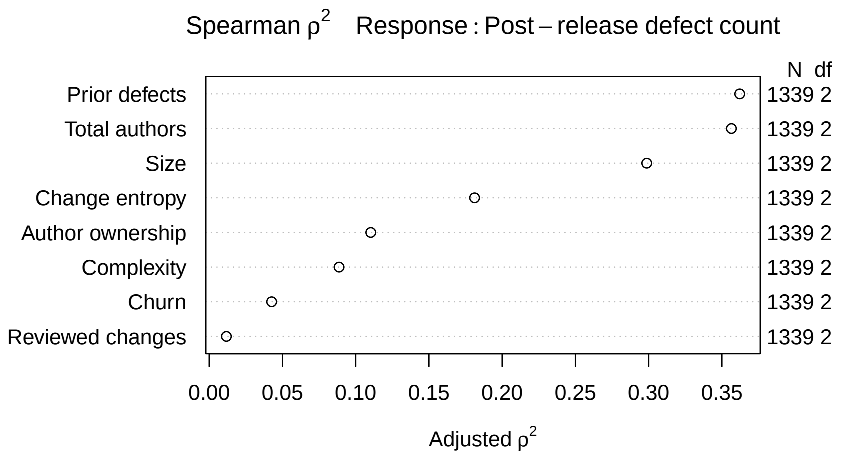


Fig. 6 Dotplot of the Spearman multiple ρ^2 of each explanatory variable and the post-release defect count in Qt 5.0. Larger values indicate a more potential for non-monotonic relationship (RQ1).

first group contains the number of prior defects, the total number of authors, and the component size, and has the largest potential for non-monotonicity. A second group contains the change entropy, author ownership, and complexity metrics, which also have some potential for non-monotonicity. The last group contains the churn and the proportion of reviewed changes.

We allocate a maximum of five knots to one explanatory variable to avoid overfitting its relationship with the post-release defect count (Harrell Jr., 2002). Since we have a large budget of degrees of freedom for the Qt 5.0 dataset, we allocate five knots to the variables in the first group, three to the variables in the second group, and no knots to the variables in the last group (i.e., fit a linear relationship). A similar process was used to allocate the degrees of freedom in the other Qt release.

In the studied VTK and ITK releases, we need allocate degrees of freedom more stringently in order to avoid exceeding the budget. Thus, we only provide additional degrees of freedom to the explanatory variable with the largest Spearman multiple ρ^2 – the number of prior bugs.

(RQ1-c) Model Analysis

In this section, we describe the outcome of our model analysis outlined in Figure 3.
(MA-1) Assessment of model stability. Table 5 shows that our defect models achieve an adjusted R^2 between 0.39 (VTK) and 0.67 (Qt 5.1). However, since these values are calculated using the same data that the models were fit with, they are inherently optimistic (Efron, 1986). Hence, we use the bootstrap technique with 1,000 iterations to calculate the optimism-reduced adjusted R^2.

Our results show that the fit of the Qt models is very stable, only having an adjusted R^2 optimism of 0.02. On the other hand, our VTK and ITK models are less stable, with adjusted R^2 optimism values of 0.19 and 0.22 respectively. While the difference in optimism is noteworthy, it does not invalidate our VTK and ITK models. The difference is partially due to the difference in sample size. Qt is composed of roughly six to eight times more components than the VTK and ITK systems. Nonetheless, the optimism values suggest that the internal validity of findings of the VTK and ITK systems should be scrutinized more carefully.

(MA-2) Estimate power of explanatory variables. Table 5 shows how much power each explanatory variable contributes to the fit of our model. For each studied release, the results are shown in two columns:

1. The *Overall* column that shows the contribution of all of the degrees of freedom that have been allocated to an explanatory variable.
2. The *Nonlinear* column that shows the contribution of only the nonlinear degrees of freedom that have been allocated to an explanatory variable. If no nonlinear degrees of freedom have been allocated to an explanatory variable, a dash (-) symbol is shown in the nonlinear column.

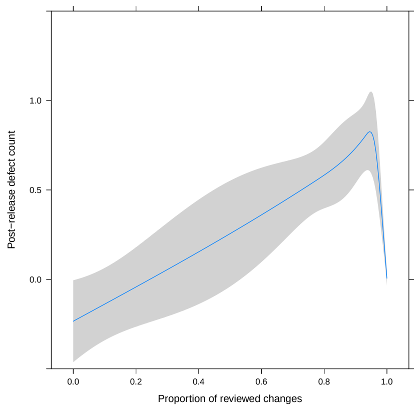
Table 5 shows that of the 15 explanatory variables that were allocated nonlinear degrees of freedom in our models, only 2 did not contribute a significant amount of explanatory power – the author ownership explanatory variable in the Qt 5.0 and 5.1 releases. However, the author ownership was entirely insignificant in Qt releases, indicating that even the linear fit of the author ownership variable did not provide significant explanatory power. The nonlinear style of modelling that we have applied to our datasets is providing significant amounts of explanatory power, leading to more accurately fitting models than our prior work (McIntosh *et al.*, 2014).

Table 5 also shows that the proportion of reviewed changes has a statistically significant impact on the defect models of Qt 5.1 and VTK 5.10. Even in Qt 5.0, where the proportion of reviewed changes does not have a statistically significant impact, we find a weak negative Spearman correlation between the post-release defect count and the proportion of reviewed changes ($\rho = -0.10$, $p < 0.001$). Reviewed churn does not play a role in the model, since it was discarded before building the model.

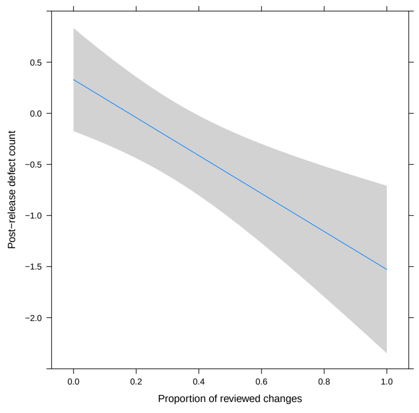
(MA-3) Examine explanatory variables in relation to the post-release defect count. Figure 7 shows the estimated post-release defect counts for Qt 5.1 and VTK of a component with a varying proportion of reviewed changes, while controlling for the other explanatory variables (i.e., holding them at their median values). We omit Qt 5.0 and ITK from Figure 7 because the proportion of reviewed changes did not have a statistically significant impact on our models.

As expected, Figure 7b shows that there is a decreasing trend of defect-proneness for VTK as the proportion of reviewed changes increases. Although this finding agrees with intuition, it should not be taken at face value, since Table 5 shows that our VTK model is not as stable as our Qt ones. Figure 7a provides support for our VTK finding, showing a steep decline in defect-proneness for increases in the proportion of reviewed changes from 0.9 to 1.0 in Qt 5.1 components.

On the other hand, our results suggest that other code review properties may provide additional explanatory power. While we found evidence suggesting that there



(a) Qt 5.1



(b) VTK

Fig. 7 The estimated count of post-release defects in a typical component for various proportions of reviewed changes. The blue line indicates the values of our model fit on the original data, while the grey area shows the 95% confidence interval based on models fit to 1,000 bootstrap samples.

is a decreasing trend in defect-proneness as the proportion of reviewed changes approaches 1.0 in Qt 5.1, Figure 7a also indicates that there is an increasing trend in defect-proneness as Qt 5.1 components increase the proportion of reviewed changes from 0 to 0.9. This range of values is accompanied by a broadening of the confidence interval, suggesting that there is less data supporting this area of the curve. We find that indeed, of the 1,337 investigated Qt 5.1 components, only 41 (3%) of them have a proportion of reviewed changes below 0.9. Nonetheless, the increasing trend in defect-proneness is suggestive of a more complex relationship between reviewing quality and defect-proneness than the proportion of reviewed changes alone can capture.

In addition, while large proportions of reviewed changes are associated with components of higher software quality in two of the four studied releases, the metric does not provide a statistically significant amount of explanatory power in the other two studied releases. To gain a richer perspective about the relationship between code review coverage and software quality, we manually inspect the ten Qt 5.0 components with the most post-release defects.

We find that the Qt 5.0 components with many post-release defects indeed tend to have lower proportions of reviewed changes. This is especially true for the collection of nine components that make up the `QtSerialPort` subsystem, where the proportion of reviewed changes does not exceed 0.1. Initial development of the `QtSerialPort` subsystem began during Qt 4.x, prior to the introduction of Gerrit to the Qt development process. Many foundational features of the subsystem were introduced in an incubation area of the Qt development tree, where reviewing policies are more lax. Hence, much of the `QtSerialPort` code was likely not code reviewed, which may have lead to the inflation in post-release defect counts.

Yet, there are components with a proportion of reviewed changes of 1 that still have post-release defects. Although only 7% of the VTK components with post-release defects (1/15) have a proportion of reviewed changes of 1, 87% (222/254), 70% (131/187), and 83% (20/24) of Qt 5.0, Qt 5.1, and ITK components respectively have a proportion of reviewed changes of 1. We further investigate with one-tailed Mann-Whitney U tests ($\alpha = 0.05$) comparing the incidence of post-release defects in components with a proportion of reviewed changes of 1 to those components with proportions of reviewed change below 1. Test results indicate that Qt 5.1 is the only studied release where the incidence of post-release defects in components with proportions of reviewed changes of 1 is significantly less than the incidence of post-release defects in components with proportions lower than 1 ($p < 2.2 \times 10^{-16}$). In the other systems, the difference is not significant ($p > 0.05$), suggesting that there is more to the relationship between code review and software quality than coverage alone can explain.

Although high values of review coverage are negatively associated with software quality in two of the four defect models, several defect-prone components have high coverage rates, suggesting that other properties of the code review process are at play.

(RQ2) Is there a relationship between code review participation and post-release defects?

As discussed in RQ1, even components with a proportion of reviewed changes of 1 (i.e., 100% code review coverage) can still be defect-prone. We suggest that a lack of participation in the code review process could be contributing to this. In fact, in thriving open source projects, such as the Linux kernel, insufficient discussion is one of the most frequently cited reasons for the rejection of a patch.[9] In recent work, Jiang *et al.* (2013) found that the amount of reviewing discussion is an important indicator of whether a patch will be accepted for integration into the Linux kernel. To investigate whether code review participation has a measurable impact on software quality, we add the participation metrics described in Table 3 to our defect models.

Since we have observed that review coverage has an impact on post-release defect rates (RQ1), we need to control for the proportion of reviewed changes when addressing RQ2. We do so by only analyzing those components with a proportion of reviewed changes of 1. Unlike RQ1, our RQ2 analysis excludes those components that have not changed, since review participation cannot have an impact on them. Although 90% ($\frac{1,201}{1,339}$) of the Qt 5.0, 88% ($\frac{1,175}{1,337}$) of the Qt 5.1, and 57% ($\frac{125}{218}$) of the ITK components survived the filtering process, only 5% ($\frac{8}{170}$) of the VTK components survive. Since the VTK dataset is no longer large enough for statistical analysis, we omit it from this analysis.

[9] `https://www.kernel.org/doc/Documentation/SubmittingPatches`

Table 6 Descriptive statistics of the studied review participation metrics.

		Qt		**ITK**		**Qt**		**ITK**
		5.0	5.1	4.3		5.0	5.1	4.3
Self-approval	Minimum	0.00	0.00	0.00	Hastily-reviewed	0.00	0.00	0.00
	1st Quartile.	0.00	1.00	0.00		2.00	0.00	1.00
	Median	0.00	1.00	1.00		3.00	0.00	1.00
	Mean	1.65	0.93	1.10		3.88	0.43	1.32
	3rd Quartile	1.00	1.00	2.00		4.00	1.00	2.00
	Maximum	82.00	27.00	9.00		79.00	10.00	4.00
No discussion	Minimum	0.00	0.00	0.00	Review window	0.00	0.08	0.00
	1st Quartile.	0.00	0.00	1.00		0.60	33.74	9.79
	Median	1.00	0.00	2.00		4.66	80.95	103.67
	Mean	1.56	0.34	1.71		69.13	484.52	781.72
	3rd Quartile	2.00	0.00	2.00		35.54	147.31	270.08
	Maximum	104.00	8.00	12.00		4,596.47	88,439.00	25,588.19
Discussion length	Minimum	0.00	1.8×10^{-5}	0.00				
	1st Quartile.	9.9×10^{-5}	9.9×10^{-5}	2.3×10^{-3}				
	Median	5.3×10^{-4}	5.3×10^{-4}	8.9×10^{-3}				
	Mean	1.4×10^{-2}	1.4×10^{-2}	2.9×10^{-2}				
	3rd Quartile	3.1×10^{-3}	3.1×10^{-3}	1.7×10^{-2}				
	Maximum	0.26	1.00	0.50				

(RQ2-a) Participation metrics

We describe the five metrics that we have devised to measure code review participation below. Table 6 provides descriptive statistics of the five studied review participation metrics. The *number of self-approved changes* counts the changes that have only been approved for integration by the original author of the change.

An appropriate amount of time should be allocated in order to sufficiently critique a proposed change. Best practices suggest that code should be not be reviewed at a rate faster than 200 lines per hour (Kemerer and Paulk, 2009). Therefore, if the time window between the creation of a review request and its approval for integration is shorter than this, the review is likely suboptimal. The *number of hastily-reviewed changes* counts the changes that have been reviewed at a rate faster than 200 lines per hour. Since our definition of hastily-reviewed changes assumes that reviewers begin reviewing a change as soon as it is assigned to them, our metric represents a lower bound of the actual proportion. We discuss the further implications of this definition in Section 5.

Reviews without accompanying discussion have not received critical analysis from other members of the development team, and hence may be prone to defects that a more thorough critique could have prevented. The operational definition that we use for a review without discussion is a patch that has been approved for integration, yet does not have any attached comments from other team members. Since our intent is to measure team discussion, we ignore comments generated by automated verifiers (e.g., CI systems) and stock comments generated by voting on a patch, since they do not create a team dialogue. The *number of changes without discussion* counts the number of changes that have been approved for integration without discussion.

While the above metrics count the number of patches that lack sufficient participation, they rely on threshold values. For example, the number of changes without discussion will only flag changes that have zero comments as problematic, while a

Table 7 Review participation model statistics (RQ2).

		Qt				**ITK**	
		5.0		5.1		4.3	
Adjusted R^2		0.69		0.46		0.58	
Optimism-reduced adjusted R^2		0.68		0.40		0.43	
Wald χ^2		2,400		1,017		179	
Budgeted Degrees of Freedom		80		78		8	
Degrees of Freedom Spent		19		18		11	
		Overall	Nonlinear	Overall	Nonlinear	Overall	Nonlinear
Size	D.F.	4	3	2	1	1	-
	χ^2	85^{***}	64^{***}	29^{***}	19^{***}	30^{***}	
Complexity	D.F.	1	-	1	-	1	-
	χ^2	1°		$< 1^{\circ}$		$< 1^{\circ}$	
Prior defects	D.F.	‡		2	1	1	-
	χ^2			11^{**}	$< 1^{\circ}$	8^{**}	
Churn	D.F.	1	-	1	-	1	-
	χ^2	1°		$< 1^{\circ}$		$< 1^{\circ}$	
Change entropy	D.F.	2	1	2	1	1	-
	χ^2	7^{*}	6^{*}	6°	5^{*}	$< 1^{\circ}$	
Total authors	D.F.	3	2	2	1	1	-
	χ^2	152^{***}	5°	52^{***}	15^{***}	7^{**}	
Minor authors	D.F.	‡		1	-	1	-
	χ^2			1°		2°	
Major authors	D.F.	†		†		†	
	χ^2						
Author ownership	D.F.	†		†		†	
	χ^2						
Self-approval	D.F.	2	1	1	-	1	-
	χ^2	3°	3°	2°		1°	
Hastily-reviewed	D.F.	†		2	1	1	-
	χ^2			45^{***}	23^{***}	5^{*}	
No discussion	D.F.	2	1	2	1	1	-
	χ^2	6^{*}	3°	5°	1°	30^{***}	
Typical review window	D.F.	†		†		†	
	χ^2						
Typical discussion length	D.F.	4	3	2	1	1	-
	χ^2	20^{***}	15^{**}	37^{***}	25^{***}	3°	

Discarded during:
† Variable clustering analysis ($|\rho| \geq 0.7$)
‡ Redundant variable analysis ($R^2 \geq 0.9$)
Statistical significance of explanatory power according to Wald χ^2 likelihood ratio test:
$\circ\ p \geq 0.05$; $^{*}\ p < 0.05$; $^{**}\ p < 0.01$; $^{***}\ p < 0.001$
- Nonlinear degrees of freedom not allocated

change with only one comment may not be much different. Hence, we introduce two component metrics that measure: (1) the typical length of the reviewing window, and (2) the typical length of a discussion. We first measure the reviewing window and the length of the discussion in each change to a component, and normalize them by the amount of churn in each change. To arrive at a single value for each component, we

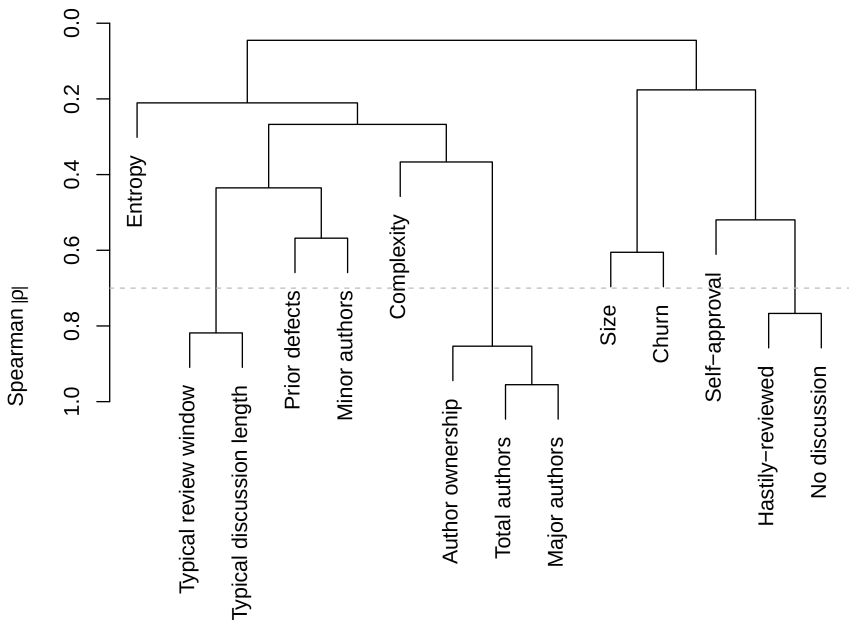


Fig. 8 Hierarchical clustering of variables according to Spearman's $|\rho|$ in Qt 5.0 (RQ2).

then take the median value across all patches of that component. We use the median rather than the mean because the median is more robust to outlier values.

(RQ2-b) Model construction

In this section, we describe the outcome of the model construction steps outlined in Figure 3.

(MC-1) Estimate budget of degrees of freedom. Since we have reduced the number of components in our datasets, we need to recalculate our degrees of freedom budgets. Table 7 shows that the Qt datasets still support 78 ($\frac{1,175}{15}$ in Qt 5.1) to 80 ($\frac{1,201}{15}$ in Qt 5.0) degrees of freedom – many more degrees than our set of explanatory variables can consume. On the other hand, the ITK dataset can only support 8 ($\frac{125}{15}$) degrees of freedom. Hence, we need to stringently allocate degrees of freedom in the ITK dataset.

(MC-2) Normality adjustment. Again, we find that there is right-skew in the post-release defect counts of the Qt datasets, but not in the ITK one. Hence, we only apply the log transformation to the post-release defect counts in the Qt datasets.

(MC-3) Correlation analysis. Figure 8 shows that, similar to RQ1, the number of major authors and the total number of authors are too highly correlated to use in the same model. We also find that author ownership is highly correlated with both the

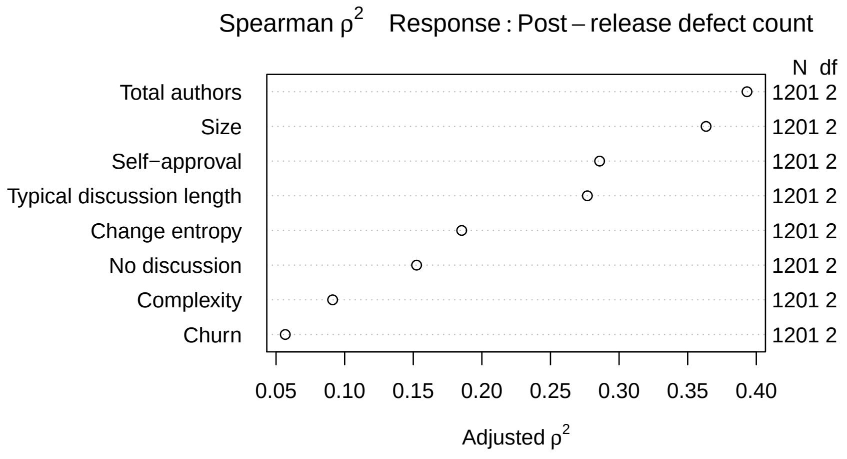


Fig. 9 Dotplot of the Spearman multiple ρ^2 of each explanatory variable and the post-release defect count in Qt 5.0. Larger values indicate a more potential for non-monotonic relationship (RQ2).

number of major authors and the total number of authors. We still select the total number of authors to represent the group.

We also find that the typical review window and the typical discussion length are too highly correlated to include in the same model. We select the typical discussion length to represent the pair, since it is a more conclusive metric, i.e., the review window does not measure the actual time that reviewers spent on the code review, while the discussion length does measure how actively a change was discussed.

In Qt 5.0, we find that the number of hastily-reviewed changes is highly correlated with the number of changes without discussion. Again, we select the number of changes without discussion to represent the pair because it is the more conclusive of the metrics.

(MC-4) Redundancy analysis. Redundancy analysis indicates that the number of minor authors and the number of prior defects in Qt 5.0 components can be accurately estimated using the other explanatory variables ($R^2_{minor} = 0.99$, $R^2_{prior} = 0.91$). Thus, we do not include them in our Qt 5.0 defect model.

(MC-5) Allocate degrees of freedom. Once again, we divide our explanatory variables into three groups according to their propensity for non-monotonicity. For example, we use Figure 9 to first group the total number of authors, component size, number of self-approved changes, and typical discussion length together. We include the change entropy and number of changes without discussion in the second group. The final group is made up of component complexity and churn. Since the Qt 5.0 dataset can support plenty of degrees of freedom, we allocate five knots to the members of the first group, three to the members of the second group, and no knots to the members of the third group. On the other hand, since the budget is more restrictive, we only use linear fits for the explanatory variables in the ITK dataset.

(RQ2-c) Model Analysis

In this section, we describe the outcome of our model analysis outlined in Figure 3.
(MA-1) Assessment of model stability. Table 7 shows that our code review participation models achieve adjusted R^2 values ranging from 0.46 (Qt 5.1) to 0.69 (Qt 5.0). Note that because these models are built using a subset of the system components, they should not be compared directly to the coverage models of RQ1.

Similar to RQ1, we find that our Qt models are more stable than the ITK one. Optimism only reduces the adjusted R^2 of Qt models by 0.01-0.06, while optimism reduces the adjusted R^2 in the ITK model by 0.15. Although we suspect that sample size is playing a major role, we still suggest that the ITK results be scrutinized more carefully.

Furthermore, we needed to fit our ITK model with more degrees of freedom than the budget suggests, which may lead to overfitting (Harrell Jr., 2002). While the optimism of our ITK model is relatively large, the optimism-reduced adjusted R^2 is 0.43, suggesting that the model still provides a meaningful and robust amount of explanatory power.
(MA-2) Estimate power of explanatory variables. Table 7 shows that many of the variables to which we allocated nonlinear degrees of freedom to provide significant boosts to the explanatory power of the model. For example, much of the explanatory power provided by the component size variable is provided by the nonlinear degrees of freedom in Qt 5.0 and 5.1. On the other hand, Table 7 shows that the nonlinear degrees of freedom that we allocate for prior defects in Qt 5.1 are not contributing a significant amount of explanatory power. Indeed, the total authors variable contributes a large amount of explanatory power in our Qt 5.0 model, but the majority of that power is provided by the linear degree of freedom. Since the adjusted R^2 of our Qt models is only 0.01-0.06 points larger than the optimism-reduced adjusted R^2, our models are likely not overfit, and hence, we are not concerned with the potentially misspent degrees of freedom on these metrics. Yet, it is important to note that allocating additional degrees of freedom will not always improve the fit of a model.

Table 7 also shows that the discussion-related explanatory variables (i.e., the number of changes without discussion and the typical discussion length) survive our model construction steps in all three of the studied releases. Furthermore, they each have a statistically significant impact on two of the three studied releases.

While the number of self-approved changes also survives our model construction steps, it does not have a significant impact on any of the defect models. This suggests that while self-approval is generally a negative quality for code reviews, it has less of an impact on software quality than discussion-related metrics do. This may in part be due to the fact that approval rights in Gerrit are only given to senior Qt team members. These team members will likely be more careful with self-approved patches than novice developers would be. We more thoroughly investigate the impact of expertise in RQ3.

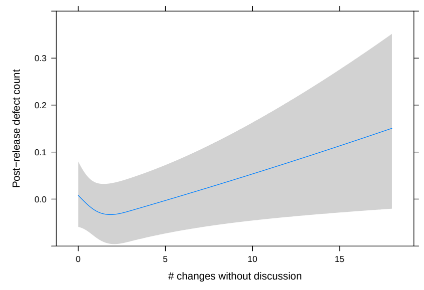
The number of hastily-reviewed changes has a significant impact on the defect models where it survives the correlation analysis. Only in Qt 5.0 was the number of hastily-reviewed changes too highly correlated with the number of changes without discussion to be added to the model. The number of hastily-reviewed changes has an especially large impact on the Qt 5.1 model, where the development team is larger (see Table 1), and would likely benefit from longer review discussions to coordinate and discuss the implications of code changes.

(MA-3) Examine explanatory variables in relation to the post-release defect count. Figure 10 shows the explanatory variables that had a statistically significant impact on our participation models in relation to the estimated post-release defect count. Figures 10a and 10b show that as the number of reviews without discussion increases, the estimated number of post-release defects tends to grow. Manual analysis of the Qt components reveals that the ones that provide backwards compatibility for Qt 4 APIs (e.g., `qt4support`) have many changes that are approved for integration without discussion, and also many post-release defects. Perhaps this is due to a shift in team focus towards newer functionality. However, our results suggest that changes to these components should also be reviewed actively.

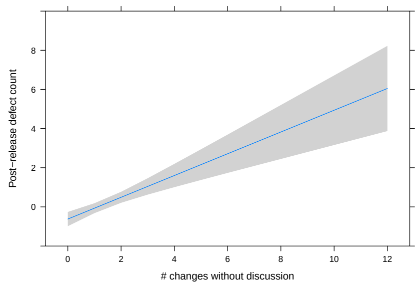
On the other hand, Figures 10c and 10d show that components that are discussed more per line of churn do not share a consistent relationship with the post-release defect count. In both studied Qt releases, there is a sharp initial increase in defect-proneness as typical discussion length increases. Furthermore, Figure 10d shows that defect-proneness slowly increases as discussion lengths continue to increase in Qt 5.1. This suggests that defect-proneness increases as components are more actively discussed. Conversely, Figure 10c shows that defect-proneness slowly decreases as discussion lengths increase in Qt 5.0. Since after the initial increase, there is only a slight slope with a broad confidence interval in both the Qt 5.0 and 5.1 curves, this suggests that changes in typical discussion length do not have a large impact on our post-release defect counts.

Finally, Figures 10e and 10f show that there is again disagreement in the direction of the relationship between the number of hastily-reviewed changes and the post-release defect count. Figure 10e shows that, as intuition would suggest, the post-release defect count in Qt 5.1 increases as the number of hastily-reviewed changes increases. This suggests that there is benefit associated with spending more time on a code review. On the other hand, Figure 10f shows a decreasing slope for ITK, suggesting that more hastily-reviewed changes are accompanied by fewer post-release defects. We suspect that this discrepancy is likely due to an interaction between the changes without discussion and the number of hastily-reviewed changes in ITK. We find that the Spearman correlation between the two explanatory variables is 0.53, suggesting that while there is not enough correlation to remove one of the pair prior to fitting our ITK model, the variables are quite similar. Furthermore, there is a positive Spearman correlation between the number of hastily-reviewed changes in a component and the post-release defect count in ITK ($\rho = 0.2$), indicating that the general trend of the relationship is increasing.

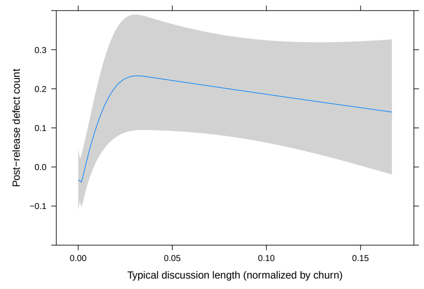
In addition, we find that in half of the ITK components, all of the changes that were flagged as hastily-reviewed were also flagged as having no discussion. Conversely, in half of the Qt 5.0 and 75% of the Qt 5.1 components, there is no overlap between those changes flagged as hastily-reviewed and those flagged as having no discussion. Hence, the decreasing trend that we observe in Figure 10f for ITK is likely to compensate for the relationship between the hastily-reviewed changes and those with no discussion.



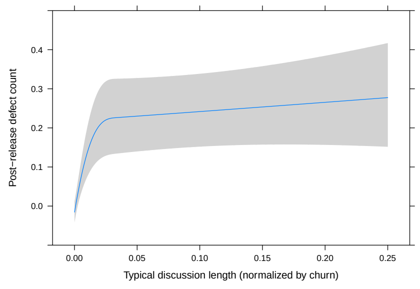
(a) No discussion (Qt 5.0)



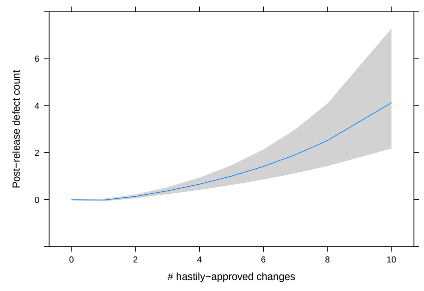
(b) No discussion (ITK 4.3)



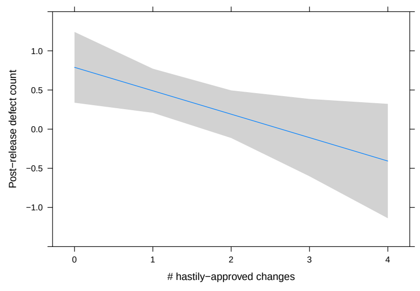
(c) Typical discussion length (Qt 5.0)



(d) Typical discussion length (Qt 5.1)



(e) Hastily-reviewed (Qt 5.1)



(f) Hastily-reviewed (ITK 4.3)

Fig. 10 The estimated count of post-release defects in a component for varying participation metrics. The blue line indicates the values of our model fit on the original data, while the grey area shows the 95% confidence interval based on models fit to 1,000 bootstrap samples.

Lack of participation in code review has a negative impact on software quality. Frequent occurrences of reviews without sufficient discussion are associated with higher post-release defect counts, suggesting that the amount of discussion generated during review should be considered when making integration decisions.

Table 8 Descriptive statistics of the studied review expertise metrics.

		Qt		**ITK**
		5.0	5.1	4.3
Lacking expertise	Minimum	0.00	0.00	0.00
	1st Quartile	0.00	2.00	0.00
	Median	0.00	3.00	1.00
	Mean	3.55	6.11	1.59
	3rd Quartile	0.00	5.00	2.00
	Maximum	343.00	139.00	14.00
Review expertise	Minimum	0.26	0.26	0.00
	1st Quartile	0.73	0.72	0.32
	Median	0.91	0.91	0.33
	Mean	0.86	0.86	0.41
	3rd Quartile	0.98	0.99	0.50
	Maximum	1.00	1.00	1.00

(RQ3) Is there a relationship between code reviewer expertise and post-release defects?

In addition to reviewer participation, reviewer (or author) expertise will likely impact software quality. For example, subject matter experts will likely make fewer mistakes that lead to defects than novices will (Mockus and Weiss, 2000). Furthermore, we hypothesize that involving a subject matter expert in the code review process will also improve the quality of a code change, and thus, reduce the likelihood of future defects. On the other hand, a change that lacks involvement from a subject matter expert may have a higher risk of introducing defects. To investigate this hypothesis, we add the code review expertise metrics to our defect models.

We again control for the impact that code review coverage has on software quality by filtering away those components with a proportion of reviewed changes below 1 (*cf.* RQ2). We further control for the impact that code review participation has on software quality by including the participation metrics in our expertise models.

(RQ3-a) Expertise metrics

Table 8 provides descriptive statistics of the two studied review expertise metrics. The *number of changes that do not involve a subject matter expert* counts the changes to a component that have not been authored nor approved for integration by a subject matter expert. In this study, we identify subject matter experts using the major author definition of Bird *et al.* (2011). This means that any author who is involved with more than 5% of the contributions to a component is considered to be an expert of that component.

We also measure the *typical reviewer expertise* by calculating the number of prior changes to a component that each reviewer who approves a change has authored or approved for integration. Similar to our typical discussion length and typical review

Table 9 Review expertise model statistics (RQ3).

		Qt				**ITK**	
		5.0		5.1		4.3	
Adjusted R^2		0.69		0.47		0.55	
Optimism-reduced adjusted R^2		0.67		0.40		0.40	
Wald χ^2		1,463		1,078		165	
Budgeted Degrees of Freedom		80		78		8	
Degrees of Freedom Spent		22		23		12	
		Overall	Nonlinear	Overall	Nonlinear	Overall	Nonlinear
Size	D.F.	4	3	2	1	1	-
	χ^2	110^{***}	76^{***}	10^{**}	5^{*}	23^{***}	
Complexity	D.F.	1	-	1	-	1	-
	χ^2	1°		$< 1^{\circ}$		$< 1^{\circ}$	
Prior defects	D.F.	‡		2	1	1	-
	χ^2			9^{*}	$< 1^{\circ}$	2°	
Churn	D.F.	1	-	1	-	1	-
	χ^2	$< 1^{\circ}$		1°		$< 1^{\circ}$	
Change entropy	D.F.	2	1	2	1	1	-
	χ^2	8^{*}	7^{**}	6^{*}	6^{*}	1°	
Total authors	D.F.	‡		2	1	‡	
	χ^2			30^{***}	15^{***}		
Minor authors	D.F.	†		1	-	1	-
	χ^2			2°		$< 1^{\circ}$	
Major authors	D.F.	†		†		†	
	χ^2						
Author ownership	D.F.	†		†		†	
	χ^2						
Self-approval	D.F.	2	1	1	-	1	-
	χ^2	22^{***}	1°	$< 1^{\circ}$		1°	
Hastily-reviewed	D.F.	†		2	1	1	-
	χ^2			48^{***}	23^{***}	6^{*}	
No discussion	D.F.	2	1	2	1	1	-
	χ^2	6°	4^{*}	3°	1°	23^{***}	
Typical review window	D.F.	†		†		†	
	χ^2						
Typical discussion length	D.F.	4	3	2	1	1	-
	χ^2	26^{***}	24^{***}	32^{***}	21^{***}	2°	
Lacking subject matter expertise	D.F.	2	1	4	3	1	-
	χ^2	80^{***}	70^{***}	34^{***}	22^{***}	$< 1^{\circ}$	
Typical reviewer expertise	D.F.	4	3	1	-	1	-
	χ^2	12^{*}	11^{*}	$< 1^{\circ}$		2°	

Discarded during:
† Variable clustering analysis ($|\rho| \geq 0.7$)
‡ Redundant variable analysis ($R^2 \geq 0.9$)
Statistical significance of explanatory power according to Wald χ^2 likelihood ratio test:
$\circ$ $p \geq 0.05$; * $p < 0.05$; ** $p < 0.01$; *** $p < 0.001$
- Nonlinear degrees of freedom not allocated

window metrics of RQ2, we take the median of the reviewer expertise values across all of the changes made to a component.

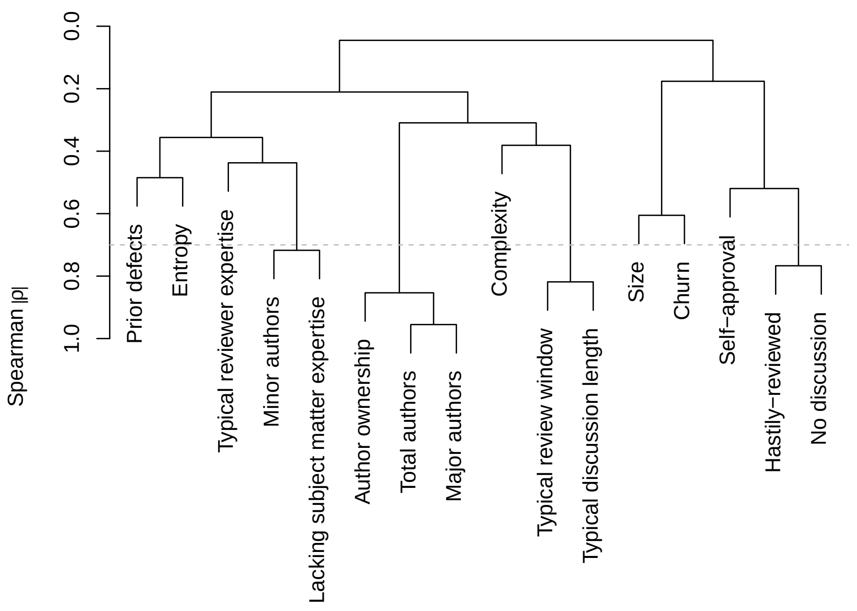


Fig. 11 Hierarchical clustering of variables according to Spearman's $|\rho|$ in Qt 5.0 (RQ3).

(RQ3-b) Model construction

In this section, we describe the outcome of the model construction steps outlined in Figure 3. Since the dataset is the same as the one used in RQ2, we omit the discussion of MC-1 and MC-2 because they are the same.

(MC-3) Correlation analysis. Figure 11 shows that, in addition to the high correlations among explanatory variables that we observed in RQ2, we find that the number of minor authors is too highly correlated with the number of changes lacking subject matter expertise to include in the Qt 5.0 model. Hence, we drop the number of minor authors in order to investigate the impact that our expertise metric has on defect-proneness. Furthermore, when we attempt to fit models using the number of minor authors instead of the number of changes lacking subject matter expertise, we find that the number of minor authors is a redundant variable, and hence, does not survive to be fit in our models.

(MC-4) Redundancy analysis. Similar to RQ2, we find that the number of prior defects and the total number of authors can be well-explained using the other explanatory variables in the Qt 5.0 dataset ($R^2_{total} = 0.94$, $R^2_{prior} = 0.91$), and are thus excluded from our model fit.

(MC-5) Allocate degrees of freedom. Again, we produce three groups of explanatory variables using the Spearman multiple ρ^2 plots like the one for Qt 5.0 shown in Figure 12. We allocate five degrees of freedom to the number of changes lacking

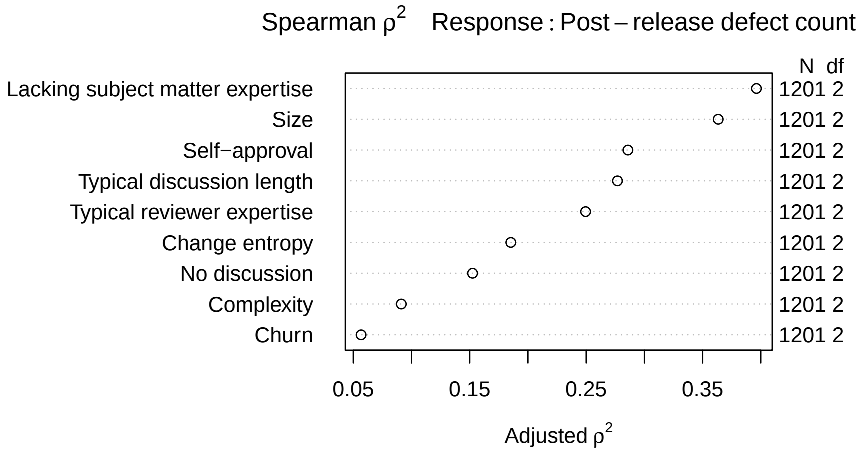


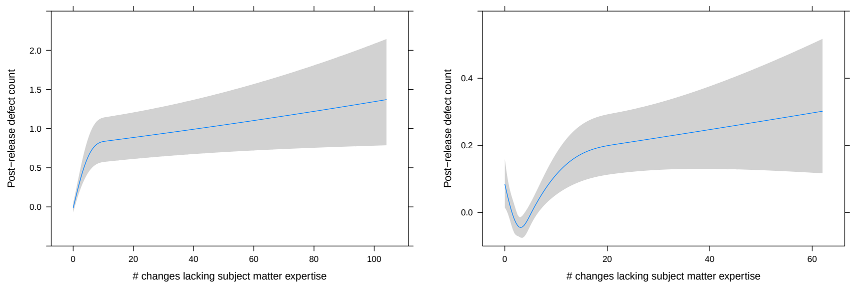
Fig. 12 Dotplot of the Spearman multiple ρ^2 of each explanatory variable and the post-release defect count in Qt 5.0. Larger values indicate a more potential for non-monotonic relationship (RQ3).

subject matter expertise and component size. Three degrees of freedom are allocated to the number of self-approved changes, typical discussion length, typical reviewer expertise, change entropy, and the number of changes without discussion. Only one degree of freedom is provided for component complexity and churn. We use a similar allocation process for the Qt 5.1 release. Due to the restrictive budget, we only allocate one degree of freedom for each explanatory variable in ITK.

(RQ3-c) Model Analysis

In this section, we describe the outcome of our model analysis outlined in Figure 3.
(MA-1) Assessment of model stability. Table 9 shows that we achieve an adjusted R^2 of between 0.47 (Qt 5.1) and 0.69 (Qt 5.0). While we find that our Qt models are once again more stable than the ITK one, it is not surprising due to the difference in sample sizes and that we needed to spend more degrees of freedom than our budget suggests. Nonetheless, the optimism-reduced adjusted R^2 for ITK is 0.4, indicating that our ITK model still provides a reasonably robust fit.

Since these models were built using the same dataset as RQ2, they can be compared. We find that our expertise models of Table 9 do not outperform the participation models of Table 7 in terms of adjusted R^2. In fact, the adjusted R^2 of our expertise model in ITK is lower than that of its discussion model. This indicates that the expertise variables have not improved the unadjusted R^2 by enough to offset the penalty for adding additional explanatory variables.
(MA-2) Estimate power of explanatory variables. Table 9 shows that our two expertise explanatory variables survive our model construction preprocessing to be included in the fit of all of our models. This suggests that expertise captures a dimension of the datasets that is distinct. Moreover, Table 9 also shows that the number of



(a) Lacking subject matter expertise (Qt 5.0) (b) Lacking subject matter expertise (Qt 5.1)

Fig. 13 The estimated count of post-release defects in a component for varying expertise metrics. The blue line indicates the values of our model fit on the original data, while the grey area shows the 95% confidence interval based on models fit to 1,000 bootstrap samples.

changes lacking subject matter expertise has a significant impact on the defect models of both studied Qt releases. This suggests that subject matter expertise is an important dimension to consider in defect models of large systems.

On the other hand, we find that neither expertise metric has a statistically significant impact on the ITK defect model. The difference in behaviour between Qt and ITK is likely due to the difference in team and system size. Table 1 shows that the Qt system is composed of 1,337-1,339 components, with code contributions from 422-435 developers. Conversely, the ITK system only has 218 components, with contributions from 41 developers.

(MA-3) Examine explanatory variables in relation to the post-release defect count. Figure 13 shows that as the number of changes lacking subject matter expertise increases, so too does the estimated post-release defect count. Figure 13a shows an increasing trend throughout the spectrum of observed numbers of changes lacking subject matter expertise, albeit a slowing trend after a count of 8. Interestingly, Figure 13b indicates that in Qt 5.1, there is a counter-intuitive drop in defect-proneness as the number of changes lacking subject matter expertise increases from 0 to 1. Indeed, we find that 25 of the 200 components (13%), that have no changes lacking subject matter expertise, have at least one post-release defect, while 0 of the 87 components with one change lacking subject matter expertise have any post-release defects. There are likely confounding factors that we have not controlled for that would explain why these components that do not lack subject matter expertise are still defect-prone. Nonetheless, the drop in Figure 13b has only a small net impact of -0.09 on the estimated count of post-release defects.

Reviewer expertise has a measurable impact on post-release defect counts in the larger studied releases. Components with many changes that lack subject matter expertise tend to be defect-prone.

5 Threats to Validity

In this section, we discuss the threats to the validity of our case study.

5.1 External validity

We focus our study on three open source systems, due to the low number of systems that satisfied our eligibility criteria for analysis. The proportion of commits that underwent code review through Gerrit presented a major challenge. Nonetheless, additional replication studies are needed.

5.2 Construct validity

Our models assume that each post-release defect is of the same weight, while in reality it may be that some post-release defects are more severe than others. Although modern Issue Tracking Systems (ITS) provide a field for practitioners to denote the priority and severity of a defect, recent work suggests that these fields are rarely accurate. For example, Herraiz *et al.* (2008) argue that the severity levels offered by the Eclipse bug reporting tool do not agree with clusters of defects that form based on the time taken to deliver a fix. Indeed, Mockus *et al.* (2002) find that the recorded priority in Apache and Mozilla projects was not related to the time taken to resolve an issue, largely because the reporters who file the defects had far less experience than the core developers who fix them. Nonetheless, each defect that we consider as a quality-impacting post-release defect was at least severe enough to warrant a fix that was integrated into the strictly controlled `release` branches of the studied systems.

Furthermore, our models are incomplete, i.e., we have not measured all potential dimensions that impact defect proneness. Other metrics that we may have overlooked may better explain defect proneness than code review activity does. However, we feel that our baseline metrics are strong, having been derived from the defect prediction literature (Koru *et al.*, 2009; Menzies *et al.*, 2002; Graves *et al.*, 2000; Nagappan and Ball, 2005, 2007; Hassan, 2009; Bird *et al.*, 2011).

5.3 Internal validity

We assume that a code review has been rushed if the elapsed time between the time that a patch has been uploaded and the time that it has been approved is shorter than the amount of time that should have been spent if the reviewer was digesting 200 lines of code per hour. However, there are likely cases where reviewers do not start reviewing the change immediately, but rush their review on a later date. Unfortunately, since reviewers do not record the time that they actually spent reviewing a patch, we must rely on heuristics to recover this information. On the other hand, our heuristic is highly conservative, i.e., reviews that are flagged as rushed are certainly rushed. Furthermore, setting the reviewing speed threshold to 100 lines per hour had little impact on the observations derived from our models.

Since there is an inherent delay between the code review (and integration) of a change and its appearance in a release, confounding factors could influence our results. However, our conclusions are intuitive, i.e., lax reviewing practices could allow defects to permeate through to the release.

6 Discussion & Related Work

In this section, we discuss the broader implications of our findings and survey the related work with respect to code review and software quality dimensions.

6.1 Discussion

Our results of Section 4 indicate that:

(RQ1) Since review coverage metrics are only significant contributors to two of the four studied releases, there appears to be more to the relationship between code review practices and post-release defects than coverage alone can explain.
(RQ2) Participation is a consistent contributor to our software quality models. For all of the studied systems, metrics that capture the length of discussion and the speed at which the review was performed offer statistically significant amounts of explanatory power to our models.
(RQ3) Expertise provides statistically significant explanatory power to our models of the large Qt system. However, participation metrics tend to have a larger impact on our models than expertise metrics do.

Thus, we conclude that optimizing integration practices for coverage, i.e., enforcing a policy that ensures that a review has taken place for any new code change, may not be sufficient to ensure that high quality code is shipped. Instead, software quality concerns may be best addressed by monitoring code review participation when deciding whether or not to integrate a code change into a codebase, making sure to involve subject matter experts.

Furthermore, our results relating to the relative impact of review participation and expertise are echoed in the recent work of Rigby *et al.* (2014), who (like Porter *et al.* (1998)) build statistical models to study *review efficiency*, i.e., the amount of time that a review was open for discussion, and *review effectiveness*, i.e., the number of defects detected during a review. Rigby *et al.* find that an increase in review participation has a larger impact on review effectiveness and efficiency than a similar increase in experience or expertise does. This is in line with our results for defect proneness.

6.2 Code reviews

Prior work has qualitatively analyzed the modern code review process used by large software systems. Rigby *et al.* (2008) find that the Apache project adopted a broadcast-based style of code review, where frequent reviews of small and independent changes were in juxtaposition to the formal code inspection style prescribed by prior research, yet were still able to deliver a high level of software quality. In more recent work, Rigby and Storey (2011) find that open source developers that adopt the broadcast-based code review style actively avoid discussions in reviews about opinionated and trivial patch characteristics. In our work, we find that active participation in the code review process tends to reduce post-release counts and improve software quality.

The identification of defects is not the sole motivation for modern code review. For example, Rigby and Storey (2011) show that non-technical issues are a frequent motivation for the patch rejection in several open source systems. Indeed, Baysal *et al.* (2013) find that *review positivity*, i.e., the proportion of accepted patches, is also influenced by non-technical factors. Furthermore, a recent qualitative study at Microsoft indicates that sharing knowledge among team members is also considered a very important motivation of modern code review (Bacchelli and Bird, 2013). Inspired by these studies, we empirically analyze the relationship between developer investment in the code review process and software quality.

Recent work has also shown that roughly 75% of the issues that are found (Mäntylä and Lassenius, 2009) and fixed (Beller *et al.*, 2014) during modern code reviews do not alter system behaviour. Rather than pointing out pertinent issues that may lead to defects, this large proportion of identified issues during code review aims to improve the maintainability of the code, with the intent of making future changes easier. Complementing their work, we find that components that are reviewed with active reviewer involvement tend to be less defect-prone, suggesting that these components are indeed more maintainable.

Kemerer and Paulk (2009) show that the introduction of design and code review to student projects at the SEI leads to code that is of higher quality. By studying student projects, Kemerer and Paulk are able to control for several confounding factors like team dynamics. Rather than control for team dynamics, our study aims to complement prior work by examining the impact of participation in the code review process of three large open source systems.

6.3 Software quality

There are many empirical studies that propose software metrics to predict software quality. For example, Hassan (2009) proposes complexity metrics (e.g., the change entropy used in our paper) that are based on the code change process instead of on the code. He shows that the entropy of the code change process is a good indicator of defect-prone source code files. Rahman and Devanbu (2013) built defect prediction models to compare the impact of product and process metrics. They show that product metrics are generally less useful than process metrics for defect prediction. Through a case study of Eclipse, Kamei *et al.* (2010) also find that process metrics tend to outperform product metrics when software quality assurance effort is considered. In this paper, our focus is on explaining the impact that modern code review practices have on software quality, rather than predicting it. Hence, we build models to study whether metrics that measure code review coverage, participation, and expertise add unique information that helps to explain incidence rates of post-release defects.

Recent work studies the relationship between source code ownership and software quality. Bird *et al.* (2011) find that ownership measures have a strong relationship with both pre- and post-release defect-proneness. Matsumoto *et al.* (2010) show that their proposed ownership measures (e.g., the number of developers and the code churn generated by each developer) are also good indicators of defect-prone source code files. Rahman and Devanbu (2011) find that lines of code that are changed to address a defect are more strongly associated with single developer contributions, suggesting that code review is a crucial part of software quality assurance. We find that the code ownership metrics that we adopt in the baseline analysis of the studied systems are very powerful, contributing a statistically significant amount of explanatory power to each of the defect models that we built.

7 Conclusions

Although code reviewing is a broadly endorsed best practice for software development, little work has empirically evaluated the impact that properties of the modern code review process have on software quality in large software systems. With the recent emergence of modern code reviewing tools like Gerrit, high quality data is now becoming available to enable such empirical studies.

The lightweight nature of modern code review processes relaxes the strict criteria of the formal code inspections that were mandated to ensure that a basic level of review participation was achieved (e.g., in-person meetings and reviewer checklists). In this paper, we quantitatively investigate three large software systems using modern code review tools (i.e., Gerrit). We build and analyze regression models that explain the incidence of post-release defects in the components of these systems. Specifically, we evaluate the conjecture that:

> *If a large proportion of the code changes that are integrated during development are either: (1) omitted from the code review process (low review coverage), (2) have lax code review involvement (low review participation), or (3) do not include a subject matter expert (low expertise), then defect-prone code will permeate through to the released software product.*

The results of our case study indicate that:

- Code review coverage metrics only contribute a significant amount of explanatory power to two of the four defect models when we control for several metrics that are known to be good explainers of software quality. This suggests that while code review coverage is an important quality to attain, other properties of the code reviewing process are likely at play.
- Discussion-related code review participation metrics contribute significant amounts of explanatory power to the defect models of each of the studied releases. Metrics related to review participation should be considered when making integration decisions.

– Expertise-related metrics that include data from the code review process also provide significant amounts of explanatory power to defect models of the larger Qt releases. Teams should aim to ensure that if a subject matter expert is not available to fix a component, they should be involved in the code reviewing process.

We believe that our findings provide strong empirical evidence to support the design of modern code integration policies that take code review coverage, participation, and expertise into consideration. Our models suggest that such policies will lead to higher quality, less defect-prone software.

8 Acknowledgments

The authors would like to acknowledge Frank Harrell Jr. for the insightful discussions and assistance with the configuration and debugging of the `rms` R package. The authors would also like to thank Peter C. Rigby and the anonymous reviewers for their fruitful comments on earlier versions of this work.

This research was partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) and JSPS KAKENHI Grant Numbers 24680003 and 25540026.

References

Bacchelli A, Bird C (2013) Expectations, Outcomes, and Challenges of Modern Code Review. In: Proc. of the 35th Int'l Conf. on Software Engineering (ICSE), pp 712–721

Baysal O, Kononenko O, Holmes R, Godfrey MW (2013) The Influence of Non-technical Factors on Code Review. In: Proc. of the 20th Working Conf. on Reverse Engineering (WCRE), pp 122–131

Beller M, Bacchelli A, Zaidman A, Juergens E (2014) Modern Code Reviews in Open-Source Projects: Which Problems Do They Fix? In: Proc. of the 11th Working Conf. on Mining Software Repositories (MSR), pp 202–211

Bettenburg N, Hassan AE, Adams B, German DM (2014) Management of community contributions: A case study on the Android and Linux software ecosystems. Empirical Software Engineering To appear

Bird C, Nagappan N, Murphy B, Gall H, Devanbu P (2011) Don't Touch My Code! Examining the Effects of Ownership on Software Quality. In: Proc. of the 8th joint meeting of the European Software Engineering Conf. and the Symposium on the Foundations of Software Engineering (ESEC/FSE), pp 4–14

Chambers JM, Hastie TJ (eds) (1992) Statistical Models in S, Wadsworth and Brooks/Cole, chap 4

Efron B (1986) How Biased is the Apparent Error Rate of a Prediction Rule? Journal of the American Statistical Association 81(394):461–470

Fagan ME (1976) Design and Code Inspections to Reduce Errors in Program Development. IBM Systems Journal 15(3):182–211

Graves TL, Karr AF, Marron JS, Siy H (2000) Predicting Fault Incidence using Software Change History. Transactions on Software Engineering (TSE) 26(7):653–661

Hamasaki K, Kula RG, Yoshida N, Cruz AEC, Fujiwara K, Iida H (2013) Who Does What during a Code Review? Datasets of OSS Peer Review Repositories. In: Proc. of the 10th Working Conf. on Mining Software Repositories (MSR), pp 49–52

Harrell Jr FE (2002) Regression Modeling Strategies, 1st edn. Springer

Harrell Jr FE (2014) rms: Regression Modeling Strategies. URL `http://biostat.mc.vanderbilt.edu/rms`, r package version 4.2-1

Harrell Jr FE, Lee KL, Califf RM, Pryor DB, Rosati RA (1984) Regression modelling strategies for improved prognostic prediction. Statistics in Medicine 3(2):143–152

Harrell Jr FE, Lee KL, Matchar DB, Reichert TA (1985) Regression models for prognostic prediction: advantages, problems, and suggested solutions. Cancer Treatment Reports 69(10):1071–1077

Hassan AE (2008) Automated Classification of Change Messages in Open Source Projects. In: Proc. of the 23rd Int'l Symposium on Applied Computing (SAC), pp 837–841

Hassan AE (2009) Predicting Faults Using the Complexity of Code Changes. In: Proc. of the 31st Int'l Conf. on Software Engineering (ICSE), pp 78–88

Hastie T, Tibshirani R, Friedman J (2009) Elements of Statistical Learning, 2nd edn. Springer

Herraiz I, German DM, Gonzalez-Barahona JM, Robles G (2008) Towards a Simplification of the Bug Report form in Eclipse. In: Proc. of the 5th Working Conf. on Mining Software Repositories (MSR), pp 145–148

Jiang Y, Adams B, German DM (2013) Will My Patch Make It? And How Fast?: Case Study on the Linux Kernel. In: Proc. of the 10th Working Conf. on Mining Software Repositories (MSR), pp 101–110

Kamei Y, Matsumoto S, Monden A, ichi Matsumoto K, Adams B, Hassan AE (2010) Revisiting Common Bug Prediction Findings Using Effort-Aware Models. In: Proc. of the 26th Int'l Conf. on Software Maintenance (ICSM), pp 1–10

Kamei Y, Shihab E, Adams B, Hassan AE, Mockus A, Sinha A, Ubayashi N (2013) A Large-Scale Empirical Study of Just-in-Time Quality Assurance. Transactions on Software Engineering (TSE) 39(6):757–773

Kemerer CF, Paulk MC (2009) The Impact of Design and Code Reviews on Software Quality: An Empirical Study Based on PSP Data. Transactions on Software Engineering (TSE) 35(4):534–550

Kim S, Whitehead EJ Jr, Zhang Y (2008) Classifying software changes: Clean or buggy? Transactions on Software Engineering (TSE) 34(2):181–196

Koru AG, Zhang D, Emam KE, Liu H (2009) An Investigation into the Functional Form of the Size-Defect Relationship for Software Modules. Transactions on Software Engineering (TSE) 35(2):293–304

Mäntylä MV, Lassenius C (2009) What Types of Defects Are Really Discovered in Code Reviews? Transactions on Software Engineering (TSE) 35(3):430–448

Matsumoto S, Kamei Y, Monden A, ichi Matsumoto K, Nakamura M (2010) An analysis of developer metrics for fault prediction. In: Proc. of the 6th Int'l Conf. on Predictive Models in Software Engineering (PROMISE), pp 18:1–18:9

McCabe TJ (1976) A complexity measure. In: Proc. of the 2nd Int'l Conf. on Software Engineering (ICSE), p 407

McIntosh S, Kamei Y, Adams B, Hassan AE (2014) The Impact of Code Review Coverage and Code Review Participation on Software Quality: A Case Study of the QT, VTK, and ITK Projects. In: Proc. of the 11th Working Conf. on Mining Software Repositories (MSR), pp 192–201

Menzies T, Stefano JSD, Chapman M, McGill K (2002) Metrics That Matter. In: Proc of the 27th Annual NASA Goddard/IEEE Software Engineering Workshop, pp 51–57

Mockus A, Votta LG (2000) Identifying Reasons for Software Changes Using Historic Databases. In: Proc. of the 16th Int'l Conf. on Software Maintenance (ICSM), pp 120–130

Mockus A, Weiss DM (2000) Predicting Risk of Software Changes. Bell Labs Technical Journal 5(2):169–180

Mockus A, Fielding RT, Herbsleb JD (2002) Two Case Studies of Open Source Software Development: Apache and Mozilla. Transactions On Software Engineering and Methodology (TOSEM) 11(3):309–346

Mukadam M, Bird C, Rigby PC (2013) Gerrit Software Code Review Data from Android. In: Proc. of the 10th Working Conf. on Mining Software Repositories (MSR), pp 45–48

Nagappan N, Ball T (2005) Use of relative code churn measures to predict system defect density. In: Proc. of the 27th Int'l Conf. on Software Engineering (ICSE), pp 284–292

Nagappan N, Ball T (2007) Using Software Dependencies and Churn Metrics to Predict Field Failures: An Empirical Case Study. In: Proc. of the 1st Int'l Symposium on Empirical Software Engineering and Measurement (ESEM), pp 364–373

Nagappan N, Ball T, Zeller A (2006) Mining metrics to predict component failures. In: Proc. of the 28th Int'l Conf. on Software Engineering (ICSE), pp 452–461

Porter A, Siy H, Mockus A, Votta L (1998) Understanding the Sources of Variation in Software Inspections. Transactions On Software Engineering and Methodology (TOSEM) 7(1):41–79

R Core Team (2013) R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing, Vienna, Austria, URL `http://www.R-project.org/`

Rahman F, Devanbu P (2011) Ownership, Experience and Defects: A Fine-Grained Study of Authorship. In: Proc. of the 33rd Int'l Conf. on Software Engineering (ICSE), pp 491–500

Rahman F, Devanbu P (2013) How, and why, process metrics are better. In: Proc. of the 35th Int'l Conf. on Software Engineering (ICSE), pp 432–441

Rigby PC, Bird C (2013) Convergent Contemporary Software Peer Review Practices. In: Proc. of the 9th joint meeting of the European Software Engineering Conf. and the Symposium on the Foundations of Software Engineering (ESEC/FSE), pp 202–212

Rigby PC, Storey MA (2011) Understanding Broadcast Based Peer Review on Open Source Software Projects. In: Proc. of the 33rd Int'l Conf. on Software Engineering (ICSE), pp 541–550

Rigby PC, German DM, Storey MA (2008) Open Source Software Peer Review Practices: A Case Study of the Apache Server. In: Proc. of the 30th Int'l Conf. on Software Engineering (ICSE), pp 541–550

Rigby PC, German DM, Cohen L, Storey MA (2014) Peer Review on Open Source Software Projects: Parameters, Statistical Models, and Theory. Transactions On Software Engineering and Methodology (TOSEM) To appear

Sarle WS (1990) The VARCLUS Procedure. In: SAS/STAT User's Guide, 4th edn, SAS Institute, Inc.

Shannon CE (1948) A Mathematical Theory of Communication. The Bell System Technical Journal 27:379–423, 623–656

Shihab E, Mockus A, Kamei Y, Adams B, Hassan AE (2011) High-Impact Defects: A Study of Breakage and Surprise Defects. In: Proc. of the 8th joint meeting of the European Software Engineering Conf. and the Symposium on the Foundations of Software Engineering (ESEC/FSE), pp 300–310

Tanaka T, Sakamoto K, Kusumoto S, ichi Matsumoto K, Kikuno T (1995) Improvement of Software Process by Process Description and Benefit Estimation. In: Proc. of the 17th Int'l Conf. on Software Engineering (ICSE), pp 123–132

A Example Scripts

In this appendix, we include Figures 14 and 15, which show how our model construction and analysis steps were implemented.

```r
require(rms)

# Load data into data frame 'data'

###############################################################################
# (MC-1) Estimate budget for degrees of freedom
###############################################################################

# Since we plan to fit using ordinary least squares, we use the below rule of
# thumb to estimate our budget
budget = floor(data$post_bugs / 15)

###############################################################################
# (MC-2) Normality adjustment
###############################################################################

# Check the skew and kurtosis of the dependent variable.
skewness(data$post_bugs)
kurtosis(data$post_bugs)

###############################################################################
# (MC-3) Correlation analysis
###############################################################################

# Produce hierarchical clusters
vcobj = varclus(~ size + complexity + ...,
                data=data,
                trans="abs")

# Plot with better labels
plot(vcobj, labels=c("Size", "Complexity", ...))

# Draw line at our threshold value
thresh = 0.7
abline(h = 1 - thresh, col="grey", lty=2)

###############################################################################
# (MC-4) Redundancy analysis
###############################################################################

# List only the variables that survive correlation analysis. nk can be set to 0
# to detect only the linearly redundant variables.
redun_obj = redun(~ size + complexity + ...,
                  data=data,
                  nk=5)

# Print the redundant variables
paste(redun_obj$Out, collapse=",␣")

###############################################################################
# (MC-5) Allocate degrees of freedom
###############################################################################

# Calculate spearman rho^2 values. Replace transform() with log1p() if the
# dependent variable is right skewed.
spearman2_obj = spearman2(transform(post_bugs) ~ size + complexity + ...,
                          data=data,
                          p=2)

# Produce dotplot of spearman rho^2 values
plot(spearman2_obj)

###############################################################################
# (MC-6) Fit regression model
###############################################################################

# Fit a regression model using 5 knots for size and 3 knots for complexity
fit = ols(transform(post_bugs) ~ rcs(size,5) + rcs(complexity,3) + ...,
          data=data,
          x=T,
          y=T)
```

Fig. 14 Example R script showing our model construction approach.

```
require(rms)

# Load data into data frame 'data'

# Load model fit into 'fit'

################################################################################
# (MA-1) Assessment of model stability
################################################################################

# Run the bootstrapped optimism calculations
num_iter = 1000
validate(fit, B=num_iter)

################################################################################
# (MA-2) Estimate power of explanatory variables
################################################################################

# Print the Wald chi^2 likelihood ratio test results
anova(fit, test="Chisq")

################################################################################
# (MA-3) Examine variables in relation to outcome
################################################################################

# Store bootstrap results in an object
bootcov_obj = bootcov(fit, B=num_iter)

# Plot the response curve for the 'size' variable. If the dependent variable
# was transformed, undo the transformation here using the 'fun' parameter.
response_curve = Predict(bootcov_obj,
                         size,
                         fun=function(x) return(undo_transform(x)))
plot(response_curve)
```

Fig. 15 Example R script showing our model analysis approach.