Quid Pro Quo: An Exploration of Reciprocity in Code Review

Carlos Gavidia-Calderon
The Open University
United Kingdom
carlos.gavidia-calderon@open.ac.uk

DongGyun Han
Singapore Management University
Singapore
dhan@smu.edu.sg

Amel Bennaceur
The Open University
United Kingdom
amel.bennaceur@open.ac.uk

ABSTRACT

We explore the role of reciprocity in code review processes. Reciprocity manifests itself in two ways: 1) reviewing code for others translates to accepted code contributions, and 2) having contributions accepted increases the reviews made for others. We use vector autoregressive (VAR) models to explore the causal relation between reviews performed and accepted contributions. After fitting VAR models for 24 active open-source developers, we found evidence of reciprocity in 6 of them. These results suggest reciprocity does play a role in code review, that can potentially be exploited to increase reviewer participation.

CCS CONCEPTS

• **Software and its engineering → Programming teams**.

KEYWORDS

code review, collaboration, time series analysis

ACM Reference Format:
Carlos Gavidia-Calderon, DongGyun Han, and Amel Bennaceur. 2022. Quid Pro Quo: An Exploration of Reciprocity in Code Review. In *19th International Conference on Mining Software Repositories (MSR '22), May 23–24, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3524842.3528522

1 INTRODUCTION

Code review is a widely adopted practice in software development, that contributes to code quality and knowledge dissemination [1]. However, developers perceive code review as time-consuming, complex, and with low value in job evaluations [10]. Given these challenges, in our MSR 2022 Hackathon project[1] we explore the reasons behind developers reviewing a colleague's work.

Besides contractual reasons —reviewing is a job requirement— we argue *reciprocity* also plays a role. Reviews require devoting time and effort into providing feedback on a contributor's code, to certify its adequacy for production. Our hypothesis is that the reviewer's investment comes with the expectation of reciprocity. They hope for their colleagues to do the same for them when they submit their own code for review. In processes where reciprocity is a factor, we expect a causal relationship between the number of reviews performed, and the number of code contributions accepted. And the other way around: a developer whose contributions are reviewed promptly and efficiently would be motivated to do the same for other developers.

[1] *Team Balloon*'s repository is available at https://github.com/cptanalatriste/code-review-reciprocity

MSR '22, May 23–24, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9303-4/22/05…$15.00
https://doi.org/10.1145/3524842.3528522

Table 1: Data gathered for the reciprocity analysis. The table shows the number of pull-requests per GitHub repository, the number of active developers found, how many of these developers have a valid VAR model , and in how many of these models there is evidence of reciprocity.

Repository	PRs	Devs	Models	Reciproc.
Apache Kafka	10.2K	7	3	1
Eclipse Jetty	2.8K	5	5	0
Deeplearning4j	3.5K	2	1	0
ReactJS	8.4K	5	3	2
GraphQL	2.2K	1	1	0
Kubernetes	47.2K	3	2	2
Visual Studio Code	10.2K	9	5	1
TypeScript	13.7	4	3	0
TensorFlow	17.3K	6	1	0

To check this hypothesis, we use pull request data to estimate vector autoregressive models (VAR) of two time series: 1) monthly pull requests *reviewed* and accepted by a developer (r_t), and 2) monthly pull requests *contributed* by a developer that got accepted (c_t). VAR models are useful for exploring causal relations and dynamic interactions in data of multiple time series [12, 15]. We consider that *unidirectional reciprocity* influences the behaviour of a developer if, in their VAR model of time series $(r_t, c_t)'$, a formal test shows that c *Granger-causes* r or vice versa. Granger causality is based on forecasting ability: c to Granger-cause r implies that past values of c can help predict future values of r. In other words, we identify reciprocity if, for a given developer, the number of code reviews they perform can be predicted by the number of code reviews they receive. *Bidirectional reciprocity* requires Granger-causality in both directions for r and c.

We gathered monthly pull request data from nine GitHub repositories, as shown in Table 1. We selected popular open-source projects from well-known organisations to maximise the chances of finding active codebases. Then, we fitted a VAR model for 24 active developers, meaning they are involved in reviewing and submitting code for at least 12 months. From them, five developers show unidirectional reciprocity and only one developer shows bidirectional reciprocity. They represent the 25% of the models, suggesting that reciprocity does play a role in code review. The presence of reciprocity can be exploited to improve code review. For example, we

can preemptively alert developers that further delays in reviewing could trigger delays in their own contributions.

This exploratory study brings attention to reciprocity analysis in software development. We invite the research community to join us, and explore the role of reciprocity in other software development practices.

2 METHODOLOGY

Our focus is on developer's reviewing activity: the reviews they perform for their colleagues, and their code submissions reviewed by others. A developer's activity shows reciprocity if the VAR model of their reviewing and contributing time series shows Granger-causality. Unidirectional reciprocity requires that r Granger-causes c *or* c Granger-causes r, while bidirectional reciprocity requires that r Granger-causes c *and* c Granger-causes r. We divide our analysis process in three stages:

2.1 Time-Series Extraction

Due to the exploratory nature of this work, we use convenience sampling [9] to gather pull-request data available on GitHub.

We extracted time series data for monthly pull-requests merged by each developer. Most pull-based development projects perform code reviews before merging [6], so this time series becomes r for a developer's VAR model. For c, we extract monthly contributions by developer that were merged by *other* developers in the team. Given our focus on the causal relation between reviews and accepted contributions, we want to explicitly exclude contributions authored and merged by the same person.

We only considered developers engaged in *both* reviewing code and submitting code for external review, to ensure data for both c_t and r_t. Since we used open-source project data, we also want to exclude occasional and rare developers due to their limited impact [2, 13]. Our heuristic for inclusion are developers with at least 12 months activity as reviewers *and* code contributors. They are listed under the column "Devs" in Table 1.

2.2 VAR Model Estimation

A developer's VAR model shows the evolution over time of the reviews they perform r_t and their accepted code contributions c_t. It contains equations for r_t and c_t, depending on earlier values (also called *lagged* values) of r and c. The p-lag vector autoregressive (VAR) model for a developer's reviewing activity has the form:

$$\begin{cases} r_t = c^r + \pi^r_{r1} r_{t-1} + \pi^r_{c1} c_{t-1} + \ldots + \pi^r_{rp} r_{t-p} + \pi^r_{cp} c_{t-p} + \varepsilon^r_t \\ c_t = c^c + \pi^c_{r1} r_{t-1} + \pi^c_{c1} c_{t-1} + \ldots + \pi^c_{rp} r_{t-p} + \pi^c_{cp} c_{t-p} + \varepsilon^c_t \end{cases}$$

We represent p-lagged values of these time series with r_{t-p} and c_{t-p}. In the r_t equation, r_{t-p} values have coefficients π^r_{rp} and c_{t-p} values have coefficients π^r_{cp}. These coefficients take the form of π^c_{rp} and π^c_{cp} in the c_t equation. Both ε^r_t and ε^c_t are white noise processes. During model fitting, we estimate c^r, c^c and $\pi^i_{ip}, i \in \{r, c\}$.

VAR models require the time series to be stationary. We accomplish this for each developer's r_t and c_t via differencing, and verify it using the Augmented Dickey-Fuller unit root test [14]. Figure 1 shows the time series r_t and c_t after differencing, for the developer showing bidirectional reciprocity. Regarding the lag length p, we chose the value that minimises the Akaike Information Criteria (AIC), as recommended for monthly VAR models [7]. Large models result in forecast errors and small models in estimation bias. By minimising AIC, we ensure an optimal lag length p.

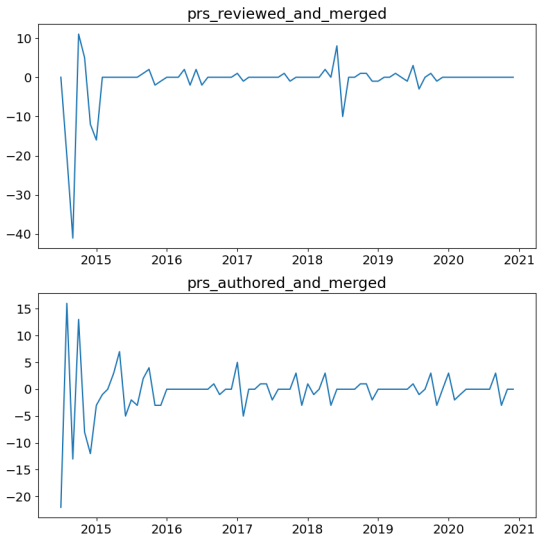

Figure 1: Monthly changes in pull-requests merged r (top) and code contributions accepted c (bottom) for a Kubernetes developer, over a 7-year period. Using changes instead of counts is part of the differencing process, to make both time series stationary.

Once we select p, we estimate the parameters π and c for the equations r_t and c_t using ordinary least squares. We then test the model fit via the Portmanteau-test for residual autocorrelation [7, 8, 12]. Significant residual autocorrelation can be a consequence of a low lag length value [8]. Hence, if the test is unsuccessful, we look for a valid fit by increasing p up to a maximum value of 12. We selected this threshold since it is uncommon to have $p > 12$ in VAR models for monthly data [7]. We list the VAR models that pass the test under the column "Models" in Table 1.

2.3 Reciprocity Identification

If c *fails* to Granger-cause r, all the coefficients $\pi^r_{cn}, n \in [1, p]$ are zero in the r_t equation. For r to fail to Granger-cause c, the coefficients $\pi^c_{rn}, n \in [1, p]$ are zero in the c_t equation. On a fitted VAR model, we test these two scenarios using the Wald statistic [12, 15]. The column "Reciproc." in Table 1 shows the developers whose VAR models evidence reciprocity, by passing at least one of the Granger-causality tests.

3 PRELIMINARY RESULTS

We implemented the process described in section 2 in Python, using Perceval [3] for data extraction and Statsmodels [11] for time series analysis.

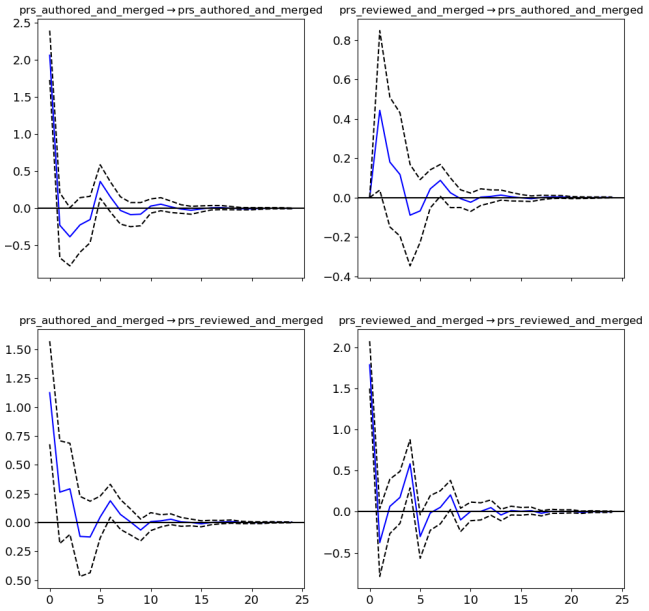

Figure 2: Impulse response functions from a VAR model fit using the pull-request data featured in Figure 1. The top row shows responses of accepted code contributions (c) to shocks in c and r. In the bottom row, we see the responses of code reviews performed (r) to these shocks.

We limited our analysis to the 24 active developers whose model fit pass the residual autocorrelation test. We excluded the other active developers due to the manual effort needed for specifying a valid VAR model. From the 24 VAR models, 6 of them evidence reciprocity, representing the 25%. Five of them show unidirectional reciprocity, and only one Kubernetes developer shows bidirectional reciprocity.

VAR models can be hard to interpret due to their numerous interacting parameters [15]. For example, the VAR(4) model for the developer with bidirectional reciprocity has 18 parameters. To obtain insights on the dynamic properties of a VAR model, we rely on *impulse response analysis*. As an example, Figure 2 shows the impulse response functions for the Kubernetes developer with bidirectional reciprocity. In the bottom-left plot, we observe that a shock in c —i.e., a sudden increase in accepted code contributions— triggers an incremental response in r, that drops to zero after 15 months. In the top-right plot, a shock in reviews performed r produces an increment in accepted code contributions c, peaking after one month to again drop to zero after 15 months.

4 CONCLUSION AND FUTURE WORK

In future work, we plan to explore how to exploit reciprocity for improving code review processes. We envision code review systems that use data to periodically fit reciprocity models of team members. Then, these systems could: 1) inform managers if reciprocity influences the team's review process, 2) show developers the review forecast based on their activity, and 3) suggest corrective measures if needed.

We also believe reciprocity can be incorporated in software process models to reflect team dynamics. In particular, game-theoretic models represent software development as interacting agents [4]. Reciprocity analysis can be used to extend these models with consequences for certain actions. Using bug prioritisation as an example, its game-theoretic model [5] could reflect that quick fixes from developers trigger accurate priorities from testers.

This study is a first step towards identifying the role of reciprocity in software development. Like in code review, we believe it plays a role in other software practices.

ACKNOWLEDGMENTS

This work was supported by the Engineering and Physical Sciences Research Council [grant numbers EP/V026747/1, EP/R013144/1].

REFERENCES

[1] Alberto Bacchelli and Christian Bird. 2013. Expectations, outcomes, and challenges of modern code review. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, David Notkin, Betty H. C. Cheng, and Klaus Pohl (Eds.). IEEE Computer Society, 712–721. https://doi.org/10.1109/ICSE.2013.6606617

[2] Enrico di Bella, Alberto Sillitti, and Giancarlo Succi. 2013. A multivariate classification of open source developers. *Inf. Sci.* 221 (2013), 72–83. https://doi.org/10.1016/j.ins.2012.09.031

[3] Santiago Dueñas, Valerio Cosentino, Gregorio Robles, and Jesús M. González-Barahona. 2018. Perceval: software project data at your will. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 1–4. https://doi.org/10.1145/3183440.3183475

[4] Carlos Gavidia-Calderon, Federica Sarro, Mark Harman, and Earl T. Barr. 2020. Game-theoretic analysis of development practices: Challenges and opportunities. *Journal of Systems and Software* 159 (jan 2020), 110424. https://doi.org/10.1016/j.jss.2019.110424

[5] Carlos Gavidia-Calderon, Federica Sarro, Mark Harman, and Earl T. Barr. 2021. The Assessor's Dilemma: Improving Bug Repair via Empirical Game Theory. *IEEE Transactions on Software Engineering* 47, 10 (oct 2021), 2143–2161. https://doi.org/10.1109/TSE.2019.2944608

[6] Georgios Gousios, Andy Zaidman, Margaret-Anne D. Storey, and Arie van Deursen. 2015. Work Practices and Challenges in Pull-Based Development: The Integrator's Perspective. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, Antonia Bertolino, Gerardo Canfora, and Sebastian G. Elbaum (Eds.). IEEE Computer Society, 358–368. https://doi.org/10.1109/ICSE.2015.55

[7] Ventzislav Ivanov and Lutz Kilian. 2005. A Practitioner's Guide to Lag Order Selection For VAR Impulse Response Analysis. *Studies in Nonlinear Dynamics & Econometrics* 9 (2005).

[8] Katarina Juselius. 2006. *The Cointegrated VAR Model: Methodology and Applications*. Oxford University Press. https://EconPapers.repec.org/RePEc:oxp:obooks:9780199285679

[9] Paul Lavrakas. 2008. *Encyclopedia of Survey Research Methods*. Sage Publications, Inc., 2455 Teller Road, Thousand Oaks California 91320 United States of America. https://doi.org/10.4135/9781412963947

[10] Laura MacLeod, Michaela Greiler, Margaret-Anne D. Storey, Christian Bird, and Jacek Czerwonka. 2018. Code Reviewing in the Trenches: Challenges and Best Practices. *IEEE Softw.* 35, 4 (2018), 34–42. https://doi.org/10.1109/MS.2017.265100500

[11] Wes McKinney, Josef Perktold, and Skipper Seabold. 2011. Time Series Analysis in Python with statsmodels. In *Proceedings of the 10th Python in Science Conference*. 107–113. https://doi.org/10.25080/Majora-ebaa42b7-012

[12] Anders Milhoj. 2016. *Multiple Time Series Modeling Using the SAS VARMAX Procedure*. SAS Institute Inc., USA.

[13] Gustavo Pinto, Igor Steinmacher, and Marco Aurélio Gerosa. 2016. More Common Than You Think: An In-depth Study of Casual Contributors. In *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14-18, 2016 - Volume 1*. IEEE Computer Society, 112–123. https://doi.org/10.1109/SANER.2016.68

[14] Ran Tao and Chris Brooks. 2019. Python Guide to Accompany Introductory Econometrics for Finance. *Available at SSRN 3475303* (2019).

[15] Eric Zivot and Jiahui Wang. 2006. *Modeling Financial Time Series with S-PLUS®*. Springer-Verlag, Berlin, Heidelberg.