Do Explicit Review Strategies Improve Code Review Performance?

Pavlína Wurzel Gonçalves,[1] Enrico Fregnan,[1] Tobias Baum,[2] Kurt Schneider,[2] Alberto Bacchelli[1]
[1]University of Zurich, Switzerland [2]Leibniz Universität Hannover, Germany
[1]<lastname>@ifi.uzh.ch [2]<firstname>.<lastname>@inf.uni-hannover.de

ABSTRACT

Context: Code review is a fundamental, yet expensive part of software engineering. Therefore, research on understanding code review and its efficiency and performance is paramount.
Objective: We aim to test the effect of a guidance approach on review effectiveness and efficiency. This effect is expected to work by lowering the cognitive load of the task; thus, we analyze the mediation relationship as well.
Method: To investigate this effect, we employ an experimental design where professional developers have to perform three code reviews. We use three conditions: no guidance, a checklist, and a checklist-based review strategy. Furthermore, we measure the reviewers' cognitive load.
Limitations: The main limitations of this study concern the specific cohort of participants, the mono-operation bias for the guidance conditions, and the generalizability to other changes and defects.
Full registered report: https://doi.org/10.17605/OSF.IO/5FPTJ;
Materials: https://doi.org/10.6084/m9.figshare.11806656

ACM Reference Format:
Pavlína Wurzel Gonçalves,[1] Enrico Fregnan,[1] Tobias Baum,[2] Kurt Schneider,[2] Alberto Bacchelli[1]. 2020. Do Explicit Review Strategies Improve Code Review Performance?. In *17th International Conference on Mining Software Repositories (MSR '20), October 5–6, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3379597.3387509

1 INTRODUCTION

Code review is a widespread software engineering practice in which one or multiple reviewers inspect a code change written by a peer [1, 23] with the primary goal of improving software quality [3]. Performing a good code review is an expensive and time-consuming task [11]. Therefore, research is investigating how to improve code review efficiency and performance.

With this aim, researchers have developed many reading techniques to guide developers in reviewing code [15]. One of the guidance techniques commonly used in the industry is checklist-based reading [2, 8, 27]. A checklist guides a reviewer in *what* to look for.

Further guidance might be possible by telling a reviewer *how* to review. Providing a specific *strategy* to perform a development task has been proven to be helpful not only with scenario-based reading techniques for code inspection [2], but also in debugging or test-driven development [16, 20].

Checklists and strategies assist developers in performing complex tasks by systematizing their activity, thus lowering the cognitive load of reviewers [17, 20]. The result should be a more effective and efficient review.

In this experiment, we investigate the effect of guidance approaches on reviewing code. We compare three treatment groups – no guidance, checklist, and strategic checklist execution (*strategy*). If we can confirm that a guided approach helps developers identify defects or make review tasks easier, not only "static" checklists, but also explicit reviewing strategies should be incorporated into review tools and used to train reviewers.

2 RESEARCH QUESTIONS

Our main goal is to investigate whether guidance on *how* to perform a review (strategy) provides additional benefits compared to guidance on *what* to look for in the review (checklist). A good review performance not only means finding many of the contained defects (effectiveness) but also finding them quickly (efficiency) [6]. We include not having any guidance as an additional control. Therefore, we ask:

RQ1: Does guidance in review lead to:
- **RQ1.1:** a higher review effectiveness (share of functional defects found)?
- **RQ1.2:** a higher review efficiency (functional defects found over the review time)?

We formalize our research question in the following hypotheses:

$H_{1.1}$: there are significant differences in review effectiveness between checklist, strategy, and no guidance approach.
$H0_{1.1}$: there are **no** significant differences in review effectiveness between checklist, strategy, and no guidance approach.

$H_{1.2}$: there are significant differences in review efficiency between checklist, strategy, and no guidance approach.
$H0_{1.2}$: there are **no** significant differences in review efficiency between checklist, strategy, and no guidance approach.

Both guidance approaches (checklist and strategy) systematize the activity of the reviewers by reducing the amount of information to keep in mind at a given time, thus, supposedly lowering developers' cognitive load [20, 25]. Therefore, we investigate:

RQ2: Is the effect of guidance on code review mediated by a lower cognitive load?

We formalize our research question in the following hypotheses:

H$_{2.1}$: Cognitive load significantly mediates the relationship between the guidance approach and review effectiveness.
H0$_{2.1}$: Cognitive load does not significantly mediate the relationship between the guidance approach and review effectiveness.

H$_{2.2}$: Cognitive load significantly mediates the relationship between the guidance approach and review efficiency.
H0$_{2.2}$: Cognitive load does not significantly mediate the relationship between the guidance approach and review efficiency.

3 RESEARCH PROTOCOL

For RQ1, we use a randomized controlled experiment design with three groups. We setup RQ2 as a correlational study; studies investigating mediators in experimental design manipulate the independent variable and the mediator is "only" measured like in observational studies (*measurement-of-mediation* design [29]).

3.1 Variables

Table 1 presents the study's variables. The guidance approach being used is the independent variable for both RQ1 and RQ2. A central dependent variable for RQ1 and RQ2 is the number of functional defects found by the participants. In RQ2, cognitive load is used as the mediator variable. Furthermore, we measure participants' demographic data (e.g., Java experience), reported in Table 1, to control for potential correlation with the review performance. We employ the following treatments *(guidance approaches)*:

No Guidance. The first group of developers does not receive any aid in the review and perform the review as they are used to.

Checklist. The second group is presented with a checklist (see Section 3.2). They are required to identify defects using this checklist, but also any other defects that might appear.

Strategy. Inspired by formerly developed strategies [18, 20], we apply the same principles in our implementation of a checklist-based reviewing strategy. While a developer has the best potential to retrieve complex information and consequently make contextualized decisions, the tool-supported strategy can free their mental capacity to do these tasks by aiding systematic execution of steps and storing and providing relevant information when needed [20]. Our strategy builds on the "static" checklist and enforces a tool-supported process on the reviewers. Furthering the checklist's purpose – to guide the review and take the mental load off developers, the strategy iterates for a developer through individual pieces of the change, displaying the checklist items relevant to the piece at the general, class, and methods levels.
The strategy is implemented as a top bar in the review task interface. It displays the same items as the checklist, grouped by scope. Differently from the checklist, items are not shown all at the same time, but participants are explicitly asked first to check the general items, then the class, then the method ones. We display only the items that are meaningful for the selected code chunk. Furthermore, the strategy highlights the code chunk(s) the user is currently reviewing. The user must explicitly mark the items as checked before being able to proceed to the next item(s) in the review strategy.

3.2 Material

The following section introduces the material we plan to use in this study; this material is publicly available [30].

Experiment UI. We employ a web-based tool that participants use to complete the experiment remotely. We log participants' answers, environment, and UI interactions. The tool was built for our previous work and we modified it according to the new experiment's requirements and past experience [5]. Figure 1 shows a partial view of the checklist implementation in the web-based experiment UI; a complete view is available in our online appendix [30].

Checklist. The checklist is developed based on recommendations in the literature and Microsoft checklists [24]. According to the literature, a good checklist requires a specific answer for each item, separates items by topic, and focuses on relevant issues [9, 12, 17]; Checklists should specify the scope in which items should be checked (*e.g.*, "for each method/class") to prevent developers from memorizing big portions of code and jumping through it [17]. Following these recommendations, we created a tentative version of the checklist.
For each seeded defect, the final checklist contains at least one item that helps to find the issue but does not give obvious clues about the type or location of the defects. To assess its face validity, we contacted three Java developers with experience in code review. Based on their feedback, we improved the items in our checklist. Then, we repeated this process with other three developers.

Cognitive Load Questionnaire. To measure cognitive load, we used a standardized questionnaire (StuMMBE-Q) [19]. It captures the two components of cognitive load (i.e., mental load and mental effort) in two 6-item subscales. Effort and difficulty ratings are reliable measures for the cognitive processing that contributes to cognitive load [13].

System Usability Scale. To measure the usability of the guidance approaches, we adapted the items of the System Usability Scale to fit the purpose of the checklist and strategy evaluation [7].

3.3 Tasks

We ask participants to perform three code reviews clarifying that they only have to look for *functional defects* in the code. The code changes to review are taken from a previous experiment on code review and contain both original and seeded defects [5]. All the changes are from the text editor jEdit, a system that was successfully employed in previous studies [5, 28]. The first short code change contains three defects, while the others contain nine and ten defects, respectively. The two large changes are presented to the participants in a randomized order. A description of the defects is publicly available [4].

Developers enter remarks in the code review UI by writing a comment in a text area that appears on any line the reviewer selects. A remark (*Reported defect*) is a note of the developer in natural language pointing at a place of a potential defect. As done previously [5], we count a remark as referring to a defect, if it is in the right position and can make a reader aware of the defect. One author rates all the remarks, a second author does the same for a subset and, then, the inter-rater agreement is computed.

Table 1: The variables of the study.

Name	Description	Scale	Operationalization
Independent variables (design):			
Guidance approach	The guidance provided by the tool to the participant: none, checklist, or strategy	nominal	see Sect. 3.2 and 3.3; randomized.
Dependent variables:			
Number of detected defects	Total detected functional defects, over all reviews by the participant (RQ1)	ratio	see Sec. 3.1
Review time	The needed net review time, i.e., the time needed for all the reviews subtracting pauses (RQ1)	ratio	Automatically measured by tool for each review task. Pauses are toggled by the participant.
Review effectiveness	Ratio of defects found by the participant over the total number of defects in the code change [6]	ratio	Computed at the end using the number of detected defects and the total number of defects.
Review efficiency	Number of defects found per hour spent reviewing [6]	ratio	Computed at the end using the number of detected defects and the review time.
Cognitive load	Load imposed on a person's cognitive system while performing a particular task [25] (RQ 2)	ordinal	see Sect. 3.1 and 3.2
Treated/Measured variables:			
Prof. development experience	Years of experience with professional software development	ordinal	Measured: 6-point scale ("never" …"11 years or more"), questionnaire
Java experience	Years of experience with the Java programming language	ordinal	Measured: 6-point scale ("never" …"11 years or more"), questionnaire
Code review experience	The number of years of experience with code reviews	ordinal	Measured: 6-point scale ("never" …"11 years or more"), questionnaire
Current program. practice	How often the participant currently programs	ordinal	Measured: 5-point scale ("not" …"daily or more often"), questionnaire
Current code review practice	How often the participant currentyl performs code reviews	ordinal	Measured: 5-point scale ("not" …"daily or more often"), questionnaire
Fitness	Perceived tiredness or fitness of the participant before the experiment	ordinal	Measured: 5-point scale ("very tired" …"very fit"), questionnaire
Experience with jEdit	Whether the participant has experience with the jEdit editor, the source of the changes to review	ordinal	Measured: 3-point scale ("none", "used", "contributed"), questionnaire
Code change	Code change under review, including the contained defects	nominal	Design: Each reviewer has to review three different code changes
Change part order	Order of presenting the code changes	nominal	For the two larger changes, the order is randomly chosen from 4 possibilities (see [5])
Usability	Perceived efficiency, effectiveness and satisfaction in the use of an object [7].	ordinal	Measured: 5-point scale ("agree" …"disagree"), questionnaire
Understandability	Participants' understanding of the code. Number of correct answers over all answers per change	ratio	Measured: Multiple choice questions on the reviewed code change (available in [30])

3.4 Participants

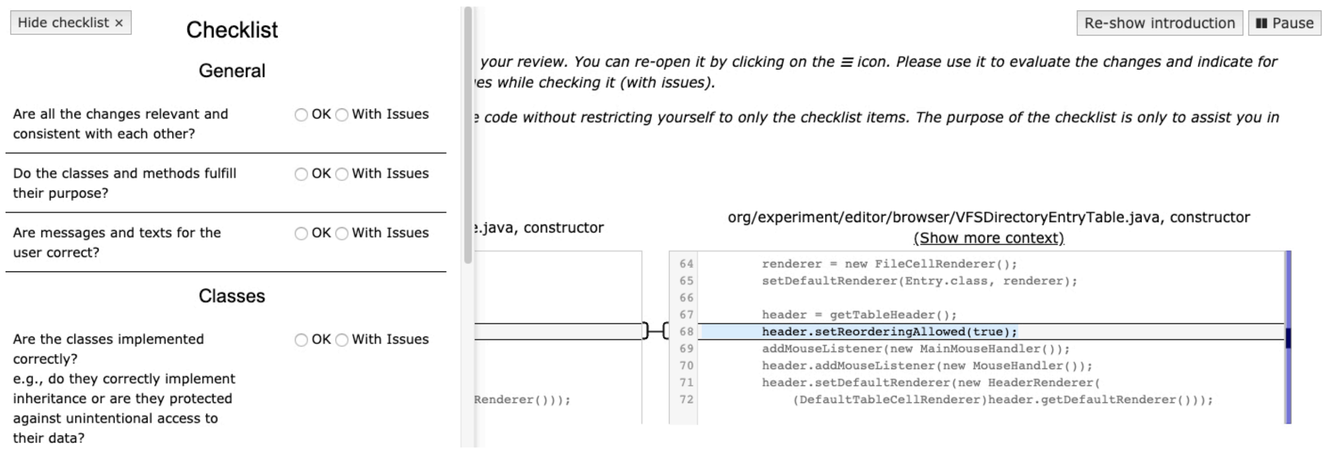
The sample consists of about 100 developers from an Indian software development company normally hired for outsourcing projects. This company provides a wide range of services (*e.g.*, web development, mobile development, and DevOps) and has more than 2,000 employees. We contacted the company via email and agreed to conduct the experiment in multiple iterations. Proceeding this way, we can adjust the experiment setup, if necessary: *e.g.*, asking for participants with longer programming experience.

The experiment participants are going to be randomly assigned to one of the three conditions (no guidance, checklist, and strategy). The review tasks are performed in fulfillment of an official contract with the company and, therefore, we expect developers to complete them as part of their official working duties. In case we detect irregularities or drop-outs, we will ask the company to provide replacements.

We performed a power analysis to estimate the sample size needed to identify existing differences between the treatment groups. Based on previous studies, we do not expect a large effect size to appear [14]. The sample size is calculated using a standard setting for an ANOVA medium effect size [10]. The estimated total sample size is 66 participants.

The selection of the sample of developers who take part in the experiment is up to the project manager of the company. However, we are going to specify the number of participants in each iteration of the experiment and the required level of experience with Java.

Pilot. We initially contacted the aforementioned company with the aim of expanding our previous experiment on the effects of ordering code changes on review effectiveness and efficiency [5].



Checklist

General

Are all the changes relevant and consistent with each other? ○ OK ○ With Issues

Do the classes and methods fulfill their purpose? ○ OK ○ With Issues

Are messages and texts for the user correct? ○ OK ○ With Issues

Classes

Are the classes implemented correctly?
e.g., do they correctly implement inheritance or are they protected against unintentional access to their data? ○ OK ○ With Issues

your review. You can re-open it by clicking on the ≡ icon. Please use it to evaluate the changes and indicate for
es while checking it (with issues).

code without restricting yourself to only the checklist items. The purpose of the checklist is only to assist you in

.java, constructor

org/experiment/editor/browser/VFSDirectoryEntryTable.java, constructor
(Show more context)

```
Renderer()));
```

```
        renderer = new FileCellRenderer();
        setDefaultRenderer(Entry.class, renderer);

        header = getTableHeader();
        header.setReorderingAllowed(true);
        addMouseListener(new MainMouseHandler());
        header.addMouseListener(new MouseHandler());
        header.setDefaultRenderer(new HeaderRenderer(
            (DefaultTableCellRenderer)header.getDefaultRenderer()));
```

Figure 1: Partial view of the checklist implementation in the web-based experiment UI.

We conducted a pilot with 29 developers who did not receive any guidance in their review. Unexpectedly, the participants were not able to identify many defects.This finding triggered the idea to investigate guidance approaches to help reviewers become more effective. Moreover, it allowed us to test the experiment platform as well as the collaboration setup with the company.

3.5 Execution Plan

Participants access the experiment online via a provided URL. The assignment is rolled out in batches of 10 to 20 developers, to check for problems after each batch. The tool presents the following workflow to each participant:

Welcome. A welcome page, containing general information about the experiment and indications on how it takes place. We ask for informed consent and explicitly request the developers not to share information about the experiment with each other. We explain that participation is voluntary and can be canceled at any time.

Explanation. An explanation of the review tasks. It emphasizes that participants should focus on finding functional issues. We also give a brief explanation of the relevant support technique and an introduction to the software system to review.

Three code review tasks. In addition to the review tasks, the participant is presented with a questionnaire to measure cognitive load at the end of each task.

Usability. If the participant is assigned to the checklist or strategy group, they are presented with a questionnaire adapted from the System Usability Scale [7] to evaluate the usability and quality of developed treatments. Moreover, participants are asked about previous knowledge of jEdit.

Demographics. General demographics questions: to gain a deeper understanding of the participants in each treatment group, we ask them general questions about potential confounders.

Closing. Final optional remarks on the experiment and ending.

3.6 Analysis Plan

Data Cleaning. We remove participants who do not finish all the reviews. We regard participants who spent less than 5 minutes on a review and entered no review remark as 'did not finish'. We also exclude participants who restart the experiment or participate several times (we collect client IPs—hashed to guarantee data anonymization—and cookies). Furthermore, we control the developers' comprehension of the system by asking questions about the change as code comprehension is an important condition for good reviews [1, 26]. We do not plan to remove other outliers unless we find specific reasons to believe the data is not valid.

Descriptive Statistics. We present descriptive statistics for the demographic, dependent, and independent variables per each treatment group by reporting means and standard deviations of respective variables. We present a correlation matrix table to assess potential covariance and relationships between examined variables.

Inferential Statistics. We take a frequentist stance. To verify our hypotheses for RQ1, we will conduct a One-way ANOVA to identify whether a difference between the three treatment groups exists and report the effect size. We will explore these differences further by using Tukey's Range Test to do the post-hoc analysis. We use mediation analysis [29] for RQ2. We assess whether there is a partial or full mediation [21, 22] of the examined relationship by cognitive load using the *mediation* R package.

Validity Threats. The implementation of the treatments might be a validity threat. To check it, we measure the checklist/strategy usability. Furthermore, the validity of the study might be undermined by our choice of changes to review. To control for overly complicated changes, we ask developers about their comprehension of the code. All the participants work in the same company and have, at the best of our knowledge, a very similar technical and cultural background; this limits the generalizability of our findings.

4 ACKNOWLEDGMENTS

A. Bacchelli, P. Wurzel Gonçalves, and E. Fregnan gratefully acknowledge the support of the Swiss National Science Foundation through the SNF Projects No. PP00P2_170529. P. Wurzel Gonçalves also acknowledges the student sponsoring support by CHOOSE, the Swiss Group for Software Engineering.

REFERENCES

[1] Alberto Bacchelli and Christian Bird. 2013. Expectations, outcomes, and challenges of modern code review. In *Proceedings of the 2013 international conference on software engineering*. IEEE Press, 712–721.

[2] Tobias Baum. 2019. *Cognitive-support code review tools: improved efficiency of change-based code review by guiding and assisting reviewers*. Ph.D. Dissertation. Hannover: Institutionelles Repositorium der Universität Hannover.

[3] Tobias Baum, Hendrik Leßmann, and Kurt Schneider. 2017. The Choice of Code Review Process: A Survey on the State of the Practice. In *Product-Focused Software Process Improvement*. Springer International Publishing, Cham, 111–127.

[4] Tobias Baum, Kurt Schneider, and Alberto Bacchelli. 2018. Online Material for "Associating Working Memory Capacity and Code Change Ordering with Code Review Performance". https://doi.org/10.6084/m9.figshare.5808609

[5] Tobias Baum, Kurt Schneider, and Alberto Bacchelli. 2019. Associating working memory capacity and code change ordering with code review performance. *Empirical Software Engineering* 24, 4 (01 Aug 2019), 1762–1798.

[6] Stefan Biffl. 2000. Analysis of the impact of reading technique and inspector capability on individual inspection performance. In *Software Engineering Conference, 2000. APSEC 2000. Proceedings. Seventh Asia-Pacific*. IEEE, 136–145.

[7] John Brooke et al. 1996. SUS-A quick and dirty usability scale. *Usability evaluation in industry* 189, 194 (1996), 4–7.

[8] Jeffrey C Carver. 2003. *The impact of educational background and experience on software inspections*. Ph.D. Dissertation. University of Maryland.

[9] Yuri Chernak. 1996. A statistical approach to the inspection checklist formal synthesis and improvement. *IEEE Transactions on Software Engineering* 22, 12 (1996), 866–874.

[10] Jacob Cohen. 1992. Statistical power analysis. *Current directions in psychological science* 1, 3 (1992), 98–101.

[11] Jason Cohen. 2010. Modern Code Review. In *Making Software*, Andy Oram and Greg Wilson (Eds.). O'Reilly, Chapter 18, 329–338.

[12] Asaf Degani and Earl L Wiener. 1991. Human factors of flight-deck checklists: the normal checklist. (1991).

[13] Krista E DeLeeuw and Richard E Mayer. 2008. A comparison of three measures of cognitive load: Evidence for separable measures of intrinsic, extraneous, and germane load. *Journal of educational psychology* 100, 1 (2008), 223.

[14] Alastair Dunsmore, Marc Roper, and Murray Wood. 2001. Systematic object-oriented inspection—an empirical study. In *Proceedings of the 23rd International Conference on Software Engineering*. IEEE Computer Society, 135–144.

[15] Shouki A Ebad. 2017. Inspection reading techniques applied to software artifacts-a systematic review. *Comput. Syst. Sci. Eng* 32, 3 (2017), 213–226.

[16] Margaret Ann Francel and Spencer Rugaber. 2001. The value of slicing while debugging. *Science of Computer Programming* 40, 2-3 (2001), 151–169.

[17] Erik Kamsties and Christopher M Lott. 1995. An empirical evaluation of three defect-detection techniques. In *European Software Engineering Conference*. Springer, 362–383.

[18] Amy J Ko, Thomas D LaToza, Stephen Hull, Ellen A Ko, William Kwok, Jane Quichocho, Harshitha Akkaraju, and Rishin Pandit. 2019. Teaching Explicit Programming Strategies to Adolescents. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. 469–475.

[19] Moritz Krell. 2017. Evaluating an instrument to measure mental load and mental effort considering different sources of validity evidence. *Cogent Education* 4, 1 (2017), 1280256.

[20] Thomas D LaToza, Maryam Arab, Dastyni Loksa, and Andrew J Ko. 2019. Explicit Programming Strategies. *arXiv preprint arXiv:1911.00046* (2019).

[21] David P MacKinnon and Amanda J Fairchild. 2009. Current directions in mediation analysis. *Current directions in psychological science* 18, 1 (2009), 16–20.

[22] David P MacKinnon, Amanda J Fairchild, and Matthew S Fritz. 2007. Mediation analysis. *Annu. Rev. Psychol.* 58 (2007), 593–614.

[23] Laura MacLeod, Michaela Greiler, Margaret-Anne Storey, Christian Bird, and Jacek Czerwonka. 2017. Code reviewing in the trenches: Challenges and best practices. *IEEE Software* 35, 4 (2017), 34–42.

[24] Steve McConnell. 2004. *Code complete*. Pearson Education.

[25] Fred GWC Paas and Jeroen JG Van Merriënboer. 1994. Instructional control of cognitive load in the training of complex cognitive tasks. *Educational psychology review* 6, 4 (1994), 351–371.

[26] Luca Pascarella, Davide Spadini, Fabio Palomba, Magiel Bruntink, and Alberto Bacchelli. 2018. Information needs in contemporary code review. *Proceedings of the ACM on Human-Computer Interaction* 2, CSCW (2018), 135.

[27] Guoping Rong, Jingyi Li, Mingjuan Xie, and Tao Zheng. 2012. The effect of checklist in code review for inexperienced students: An empirical study. In *2012 IEEE 25th Conference on Software Engineering Education and Training*. IEEE, 120–124.

[28] David Rothlisberger, Marcel Harry, Walter Binder, Philippe Moret, Danilo Ansaloni, Alex Villazon, and Oscar Nierstrasz. 2012. Exploiting dynamic information in IDEs improves speed and correctness of software maintenance tasks. *IEEE Transactions on Software Engineering* 38, 3 (2012), 579–591.

[29] Steven J Spencer, Mark P Zanna, and Geoffrey T Fong. 2005. Establishing a causal chain: why experiments are often more effective than mediational analyses in examining psychological processes. *Journal of personality and social psychology* 89, 6 (2005), 845.

[30] Pavlína Wurzel Gonçalves, Enrico Fregnan, Tobias Baum, Kurt Schneider, and Alberto Bacchelli. 2020. *Online appendix*. https://doi.org/10.6084/m9.figshare.11806656