

Article

Not peer-reviewed version

Structured Code Review in Translational Neuromodeling and Computational Psychiatry

Inês Pereira ^{*†}, [Herman Galiouline](#) ^{*†}, Réka Enz, [Alexander Hess](#), [Matthias Müller-Schrader](#), [Dina von Werder](#), Imre Kertesz, [Florian Schönleitner](#), Katja Brand, Lars Kasper, Lilian Weber, Frederike Petzschnner, Sandra Iglesias, Jakob Heinzle, Klaas Stephan

Posted Date: 2 October 2025

doi: 10.20944/preprints202510.0133.v1

Keywords: mental health; scientific integrity; reproducibility; software engineering



Preprints.org is a free multidisciplinary platform providing preprint service that is dedicated to making early versions of research outputs permanently available and citable. Preprints posted at Preprints.org appear in Web of Science, Crossref, Google Scholar, Scilit, Europe PMC.

Copyright: This open access article is published under a Creative Commons CC BY 4.0 license, which permit the free download, distribution, and reuse, provided that the author and preprint are cited in any reuse.

Article

Structured Code Review in Translational Neuromodeling and Computational Psychiatry

Inês Pereira ^{1,*†}, Herman Galiouline ^{1,*†}, Réka Enz ^{1,11,12}, Alexander J. Hess ¹, Matthias Müller-Schrader ^{1,10}, Dina von Werder ¹, Imre Kertesz ¹, Florian M. Schönleitner ¹, Katja Brand ¹, Lars Kasper ^{2,3}, Lilian A. Weber ^{4,5,6}, Frederike H. Petzschner ^{7,8}, Sandra Iglesias ¹, Jakob Heinzle ¹ and Klaas Enno Stephan ^{1,9}

¹ Translational Neuromodeling Unit (TNU), Institute for Biomedical Engineering, University of Zurich & ETH Zurich, 8032 Zurich, Switzerland

² Krembil Brain Institute, University Health Network, Toronto, Ontario, Canada

³ Toronto Neuroimaging Facility, Department of Psychology, University of Toronto, Toronto, Ontario, Canada

⁴ Department of Psychiatry, University of Oxford, Oxford, UK

⁵ Wellcome Centre for Integrative Neuroimaging (WIN), Department of Experimental Psychology, University of Oxford, Oxford, UK

⁶ Institute for Cognitive Science, Osnabrueck University, Osnabrueck, Germany

⁷ Cognitive and Psychological Sciences, Brown University, Providence, RI, USA

⁸ Robert J. and Nancy D. Carney Institute for Brain Science, Brown University, Providence, RI, USA

⁹ Max Planck Institute for Metabolism Research, Cologne, Germany

¹⁰ Fields Institute for Research in Mathematical Sciences, Toronto, Ontario, Canada

¹¹ Department of Neurology, Schulthess Clinic, Zürich, Switzerland

¹² Department of Health Sciences and Technology (DHEST), ETH Zurich, Zurich, Switzerland

* Correspondence: ines.pereira@alumni.ethz.ch (I.P.); hermang@ethz.ch (H.G.)

† shared first-authorship.

Abstract

Errors in scientific code pose a significant risk to the accuracy of research. Yet, formal code review remains uncommon in academia. Drawing on our experience implementing code review in Translational Neuromodeling and Computational Psychiatry, we present a pragmatic framework for improving code quality, reproducibility, and collaboration. Our structured checklist (organized by priority and type of work: experimental, theoretical, machine learning) offers a practical guide for identifying common coding issues, from data handling errors to flawed statistical analyses. We also integrate best practices from software engineering and highlight the emerging role of large language models in automating aspects of review. We argue that, despite perceived costs, code review significantly enhances scientific reliability and fosters a culture of transparency and continuous learning. Our proposal provides an adaptable model for integrating code review into research workflows, helping mitigate errors before publication and strengthening trust in scientific results.

Keywords: mental health; scientific integrity; reproducibility; software engineering

Introduction & Motivation

Very few error-free computer programs have ever been written [1]. In fact, in professional software development, the industry average is around 1–25 errors per 1,000 lines of code for delivered software [2]. While there is a growing interest in improving programming skills in academia [3–8], few researchers benefit from comprehensive training in software engineering and testing [3,9]. Whereas scientists invest a significant portion of their time in creating software, they frequently fail

to leverage modern advancements in software engineering [10]. One reason for the limited adoption of software engineering practices is a general lack of awareness about their benefits [11]. For instance, in a survey in 2017, only around 50% of postdoctoral researchers had received any formal training in software development [12]. Illustrating this further, a minority of scientists are familiar with or make use of basic testing procedures [13]. Thus, it is most likely the case that the error rate in code written by academic scientists is much higher than in the software industry.

This raises concerns regarding the accuracy of scientific results. Indeed, there are numerous high-profile examples where scientific conclusions have been affected by coding errors [14–23], including cases with major implications for political and societal decisions [16,17,24]. While honest errors are an inevitable part of science — and their voluntary disclosure should be applauded, not punished — it is our duty to mitigate their occurrence.

One way to tackle this challenge is to establish code review as an integral step of all scientific projects. Code review is defined as “an activity in which people other than the author of a software deliverable examine it for defects and improvement opportunities” [25], with some hailing this practice as “the single biggest thing [one] can do to improve [their] code” [26]. Peer code review represents one of the most effective methods for detecting errors and is a standard process in the software industry [2].

Yet, code review is not a common practice in science. Several reasons account for this: lack of knowledge, cultural opposition, or resistance to change [25]. Indeed, there is often the perception that a certain project is not large or critical enough to warrant code review. In addition, there is a widely held belief that code reviews take up too much time and could slow projects down [25]. However, this perspective neglects the fact that any technique enabling early corrections and improvements has the potential to save disproportionately more time in the future. In addition, early detection of critical issues can avoid costly corrections down the line, including amendments or, in the worst case, retraction of publications [27]. Most importantly, it is central to the scientific process that any known source of potential errors is addressed proactively in order to protect the reliability of specific conclusions and the trustworthiness of science in general.

While we are not aware of any data that compare the frequency and/or impact of software errors across scientific fields, we deem it plausible that interdisciplinary fields which combine computational with other approaches may face particular challenges. This is because the very nature of these fields requires scientists, regardless of their background and experience with software engineering, to engage in the use, modification and construction of complex software. Here, we focus on fields that apply computational methods to mental health problems, specifically Translational Neuromodeling (TN; developing computational assays of brain [dys]function) and Computational Psychiatry (CP; the application of computational assays to clinical problems). These closely related fields are useful targets for discussing principles of code review because they use sophisticated computational methods (e.g., generative models, machine learning) and numerous data types (e.g., neuroimaging, electrophysiology, behavior, self-report, clinical outcomes). As a consequence, many challenges and solutions for code review that concern TN/CP are likely to occur for neuroscience and mental health research as well.

This article does not address the general principles of good code design and documentation directly. These principles have been expanded upon in many previous excellent (and sometimes even humorous) references and resources [9,28–37]. In addition, we do not aim to prescribe rules or guidelines for code review. Again, many experienced software engineers have written excellent articles on the topic [25,38–41], and several publications related to code review in research already exist [42–44]. Instead, in order to lower barriers for researchers not yet familiar with code review, we aim to share our concrete workflows, experience with, and learnings from introducing code review into our research work. We hope that this pragmatic approach will provide a concrete point of entry for researchers who are looking to add code review to their project workflow but are unsure where to start.

Our Experience

At the institution where all co-authors work or used to work — the Translational Neuromodeling Unit at Zürich — code review by an independent scientist became mandatory in early 2019, after a prolonged phase-in period. We have since made independent code review a condition that must be fulfilled before accessing test (held-out) data in machine learning analyses and, in general, before submitting articles for publication, either to a journal or a preprint server. This decision, as well as the requirements for a sufficient code review, were documented in an internal working instruction (WI). Initially, no detailed checklist was provided; instead, the WI described the mindset, principles and goals of code review in our unit. It focused on three main points: (1) reproduction of results on a different system/machine; (2) checking the overall steps of the analysis pipeline and searching for any obvious flaws; and (3) checking for “typical” errors that occur frequently in the context of the specific analysis approach. Finally, the code reviewer was asked to submit a brief written review, which documented what was done and what issues (if any) were encountered. This report was then timestamped and stored along with the code.

This process of code review also has implication for the code author. In addition to requiring well-structured and readable code, analyses are to be conducted in end-to-end pipelines, from raw data to the final results, including figures. Manual interventions are discouraged; if they are inevitable, they need to be documented in a precise manner so that the results can be reproduced.

In the last few years, this procedure has led to several significant errors being uncovered in our work, some early in the analyses, others later on. Examples of mistakes detected in our research group include:

- Loading of the incorrect data files.
- Errors in the calculations of test statistics.
- Indexing errors in vector operations leading to row-shifted data (with subjects being associated with incorrect data).
- Incorrect custom implementations of complex operations (i.e., nested cross-validation).
- Unintended use of default settings in (external) open-source toolboxes. In this particular case, issues in the code were detected after publication. Once the programming mistake was detected, the journal was proactively contacted and informed of the situation, analyses were rerun with the correct settings, and the updated results were published as a Corrigendum [18].

Having one’s work criticized can be a difficult process. In order to increase acceptance, a culture of openness was encouraged, and polite and constructive feedback was defined as a must. Furthermore, issues of wider interest (e.g., errors in the context of complex statistical or computational approaches) are openly discussed in larger rounds, e.g., our weekly methods or group meetings. Thus, while communication typically happens most intensively between the code author and the code reviewer, any other member of the research group can be consulted. In addition, the time for code review is factored into the timeline of a project. Thus, despite the additional efforts, the overall feeling in our research group is that code review not only plays a crucial role for detecting errors and fosters more trust in our results, but also offers learning opportunities for researchers that will likely prove useful for the future career, whether in academia or industry.

A Proposal

In a dual effort to further standardize our code review procedures and make them available to interested scientists outside our unit, we have recently developed a structured checklist based on our own experience, discussions within the lab, and external sources. The checklist is intended not as a rigid standard, but as a practical guide that helps reviewers and authors prioritize the most critical aspects of a computational project. By offering a clear framework, it aims to reduce ambiguity, support constructive feedback, and encourage a culture of regular and collaborative code review.

In terms of external sources, we supplemented our own lab experience with established guidance from the broader software engineering community. Several resources contributed to our

understanding of what makes code review effective, including practical guides on identifying potential issues related to functionality, performance, and maintainability [45,46]. These emphasize the importance of clear reviewer responsibilities, the role of testing, and the value of early and frequent feedback during code development [26,38,39]. Best practices from large-scale software projects provided insight into systematic review standards, including expectations around readability, modularity, and minimizing technical debt [32]. We also consulted curated collections of tools and methodologies for managing code review in collaborative settings, as well as discussions of common pitfalls specific to machine learning pipelines [47,48].

The checklist presented below is organized along two axes: level of priority and type of work. At the top level, we differentiate between fundamental checks (basic correctness, reproducibility, usability) and more advanced checks (performance, maintainability, robustness). Within each of these, we further separate general review points from domain-specific considerations. The latter are grouped according to the nature of the work: empirical data analyses, computational modeling, and machine learning. The list is not intended to be exhaustive, but to provide a structured and adaptable reference to researchers and laboratories not yet familiar with code review.

Fundamental

General points

- Documentation:
 - Are there clearly specified instructions or a structured README file?
 - Are dependencies, versions, and required toolboxes specified? If not, is there a separate configuration file (e.g., .yaml, .txt) that does this?
 - Can you follow the setup instructions?
 - Are inline comments and function documentation present where necessary (e.g., more complex parts of the code)?
- Dependencies:
 - If applicable, were the correct versions of critical toolboxes/software packages used by the code author?
- Input:
 - Are the correct data files loaded?
- General functionality:
 - Does the code do what it was supposed to do? (See below for concrete checks for different types of projects.)
 - Can you run the pipeline without errors and reproduce the results, incl. all plots?
- Readability:
 - Can I understand what the code does by reading it?
 - Are function and variable names descriptive and consistent?
 - Do the names (of fields, variables, parameters, methods, etc.) reflect the things they represent?

- Quality control:
 - If there is a testing framework (e.g., unit testing), can you run it without any errors?
 - If there are automated data checks, are these functional, comprehensive and sufficient? Are there any other checks that you would deem necessary? See examples in later sections for some ideas.
- Plotting:
 - Are the analysis results represented correctly in the figures/plots the code produces?
 - Do the plots seem plausible? Is there anything surprising (e.g., outliers, unusual scale levels) that suggests further checks are necessary?

Empirical data analyses

Detailed checks for 1-3 randomly selected subjects:

- Do the main scripts align with the experimental design and intended analysis pipeline?
- Are file paths, parameters, and processing steps clear and correct?
- Input:
 - Are the correct data files loaded at each analysis stage?
 - Do the data files have the expected dimensions and parameters? (e.g., number of slices in MRI, sampling rates in EEG, or timepoints in physiological recordings)
- Processing:
 - Are the outputs of individual preprocessing steps as expected? For example:
 - Imaging: Motion correction, co-registration, normalization, and masking.
 - EEG: Filtering, artifact removal, and channel alignment.
 - Behavioral: Missing value handling, reaction time calculations, or stimulus-response alignment.
 - Are matrix dimensions and orientations (i.e., transposes) correct throughout the pipeline?
 - Are behavioral, physiological, and imaging variables within reasonable ranges?
 - If tasks involve stimulus presentation or response logging, are timestamps and condition assignments accurate (i.e., in the correct order, reasonable time intervals, etc.)?
 - Are subject/session IDs consistently aligned across different data types (i.e., a subject's fMRI data is assigned to the same subject ID as their physiological data)?
 - Are alignment procedures (e.g., imaging and physiological data synchronization) verified?

With the full dataset:

- Are the correct subjects included in the analysis?
- Are edge cases (e.g., missing data, outliers, or inconsistent responses) documented and handled?
- Are statistical tests correctly implemented?
- If sources of variability exist that render replication problematic (e.g., context-dependent seeds for random number generation, differences between operating systems), can they be removed? For instance, are seeds of random number generators being set?
- Do expected patterns emerge? (e.g., learning curves in behavioral data, typical connectivity patterns in rs-fMRI, or standard ERP components in EEG)

Computational modeling

- Does the implementation of a computational model correctly follow the theoretical model as described in published papers?
- For generative models, are the priors specified as intended?
- Does the implementation handle floating-point precision issues, avoiding numerical instability (e.g., underflow, overflow, catastrophic cancellation)?
- If the computations involve optimization:
 - Does the code explicitly check for convergence?
 - Does the code log information about the optimisation process (e.g., objective function value across iterations), as a basis for detecting problems?
 - Does the code include measures to avoid being trapped in local extrema (e.g., multi-start approaches)
- Do functions behave as expected, without producing anomalous values?
- Does running the same analysis multiple times with the same seeds produce identical results?
- Does the model behave as expected on controlled synthetic datasets where the ground truth is known?

Machine learning

- Are there any instances of information leakage, i.e., did held-out test data influence any aspect of the training phase (e.g., normalization, standardization, or selection of features)?
- If cross validation is used, is it implemented correctly (e.g., no leakage between folds, proper loop nesting)?
- Is feature preprocessing consistent? Example: if the training data are standardized, is the test set standardized as well?

- Are the assumptions of the model met? Example: is the target variable mean-centered and the features standardized when using an Elastic Net regression model?

Advanced

General points

- Dealing with data and large files:
 - Are code and data fully separated?
 - Are unnecessary files (e.g., data files and other large files) excluded via .gitignore?
- Code optimization:
 - Can the code be simplified?
 - Are there instances of duplication? If so, should it be refactored or is this acceptable at this stage?
 - Are there any obvious performance enhancements? Example: vectorization, parallelization over subjects.
 - Is there any unused code? Check with the code author if it can be deleted. If it should be kept, clearly document why.
 - Data structures: any obvious optimizations? Example: cumbersome loops over a list when a dictionary could allow for a key-based search.
- Coding style:
 - Is formatting consistent? Automation tools can be used.
 - If required, was the style guide followed?
- Code testing:
 - Do tests check that things run *and* that things break in an intended way (i.e., produce an expected error)?
 - Do the tests at least cover potentially confusing or complicated sections of code?
 - Do they cover edge cases?
 - Are there cases that have not been considered and should be covered?
 - Check test coverage: do the tests cover the important parts?

Empirical data analyses

- Is the pipeline a “one-click” pipeline? Is there one script that runs everything for the user?
- Are trial-level and summary statistics both available for further (re-)analysis?

- Are corrections for multiple comparisons correctly applied in statistical analyses?

Computational modeling

- Do all mathematical equations in the code match those in the referenced papers exactly?
- Are gradients stable, ensuring they do not explode or vanish in optimization steps?
- Are data transformations applied correctly (e.g., z-scaling, log-space computations)?
- When working in a Bayesian setting, do posterior distributions update correctly with new observations, avoiding degeneracies (e.g., unexpected shapes)?
- Are probability distributions correctly marginalized and normalized when approximated numerically?
- Does the model handle extreme values (e.g., very high or low precision) correctly?
- Does the model handle missing data gracefully, without causing undefined behavior or nonsensical outputs?
- If using empirical (real) data, are there simple sanity checks that can be performed that check the plausibility of results? For instance, is it possible to reproduce known observed neural or behavioral phenomena?

Machine learning

- Are results qualitatively stable when retraining models with different random seeds?
- Are multiple performance metrics beyond accuracy reported (e.g., F1-score, ROC-AUC, calibration curves)?

Code Review and Analysis Plans

For clarity, it is important to distinguish the goals of code review from those of predefining analyses before the data are touched (i.e., analysis plans). Analysis plans prespecify the scientific questions (e.g., hypotheses) of a project and how they are addressed by concretely defined analyses. Defining analysis plans ex ante is an important and effective strategy to improve reproducibility and robustness of the scientific process. For example, they protect against cognitive biases, shield against mistaking postdictions for predictions, and prevent p-hacking [49,50]. In our unit, analysis plans were made mandatory in parallel to code review.

Analysis plans often include planned checks for robustness, such as sensitivity analyses or benchmarking predictive models of interest against simpler baseline models. In order to place constraints on the efforts of code reviewers, it is important to emphasize that their responsibility is not to search for potential gaps in analysis plans. In other words, the checks for robustness that code reviewers perform are conditional on the given analysis plan (if it exists). Of course, this should not prevent code reviewers from alerting the responsible scientists to missing robustness checks in an analysis plan; these can then be implemented post hoc as a deviation from the analysis plan that must be reported transparently in the manuscript.

Benefits, Costs, and a Way Forward

Despite the growing awareness of the importance of code quality in scientific computing, formal code review is rarely applied in the academic environment. One barrier may be the perceived costs: code review can be time-consuming and may expose errors that complicate ongoing work. However, in our view, the benefits (early error detection, enhanced reproducibility, and more robust scientific results) vastly outweigh these costs. Especially in high-impact or clinical research contexts, the cost of undetected software errors may be far higher than the investment required for a review process. Furthermore, code review can also enhance the reputation of individual researchers and research groups as it signals a commitment to quality. Institutions and journals may further incentivize adoption by developing norms around reviewer credit and by framing rigorous code validation as a mark of high-quality science. As reproducibility crises across disciplines have shown, it is far more costly, professionally and scientifically, to issue corrections or retractions than to catch problems early in the process.

Looking ahead, the integration of large language models (LLMs) into the code review process presents an opportunity to both scale and systematize review efforts. Recent advances have shown that LLMs, particularly those with long-context capabilities, can assist in identifying bugs, suggest code improvements, and enforce standards across entire codebases. In particular, fine-tuned models have demonstrated significantly higher accuracy than general-purpose models, and techniques such as few-shot prompting, metadata augmentation, and parameter-efficient fine-tuning continue to improve performance while reducing computational overhead [51–54]. These systems are increasingly capable of generating readable comments, detecting latent defects, and even repairing code based on review feedback [55–60]. Of particular interest in scientific contexts is their emerging ability to retroactively analyze published code repositories (as well as the associated papers [61]) and surface previously undetected errors, potentially preventing flawed findings from being propagated. At the same time, challenges such as hallucinations, training data bias, and interpretability of automated reviews highlight the need for human oversight and domain-specific tuning [62,63]. Nevertheless, as LLM tools become more reliable and privacy-aware, their role in routine code validation, and ultimately in the safeguarding of scientific integrity, is likely to expand.

In summary, we have presented a pragmatic, experience-based proposal for structured code review. The proposed framework is specifically motivated by the challenges encountered in Translational Neuromodeling and Computational Psychiatry, but will also broadly apply to other areas of neuroscience and mental health research. Our proposed checklist offers a structured yet adaptable framework for identifying and prioritizing common issues, spanning experimental pipelines, theoretical modeling, and machine learning applications. By combining lessons from our own practice with insights from software engineering and emerging tools such as large language models, we hope to contribute to a more systematic and sustainable review culture in academic research. In parallel, we highlight the importance of incentivizing participation: recognizing reviewers' efforts, formalizing contributions, and embracing automation where appropriate. As the research community continues to confront challenges of reproducibility and software reliability, we see structured code review not as an administrative burden but as an opportunity for collaboration, quality assurance, and scientific integrity.

References

1. D. A. W. Soergel, "Rampant software errors may undermine scientific results," Jul. 29, 2015, *F1000Research*: 3:303. doi: 10.12688/f1000research.5930.2.
2. S. McConnell, *Code complete*, Second edition. Redmond, Washington: Microsoft Press, 2004.
3. E.-M. Arvanitou, A. Ampatzoglou, A. Chatzigeorgiou, and J. C. Carver, "Software engineering practices for scientific software development: A systematic mapping study," *Journal of Systems and Software*, vol. 172, p. 110848, Feb. 2021, doi: 10.1016/j.jss.2020.110848.

4. T. Storer, "Bridging the Chasm: A Survey of Software Engineering Practice in Scientific Programming," *ACM Comput. Surv.*, vol. 50, no. 4, pp. 1–32, Jul. 2018, doi: 10.1145/3084225.
5. C. Woolston, "Why science needs more research software engineers," *Nature*, May 2022, doi: 10.1038/d41586-022-01516-2.
6. "RSECon 2025," RSECon25. Accessed: Apr. 26, 2025. [Online]. Available: <https://rsecon25.society-rse.org/>
7. "Nordic RSE," Nordic RSE. Accessed: Apr. 26, 2025. [Online]. Available: <https://nordic-rse.org/>
8. "nl-rse," nl-rse. Accessed: Apr. 26, 2025. [Online]. Available: <https://nl-rse.org/>
9. Z. Merali, "Computational science: ...Error," *Nature*, vol. 467, no. 7317, pp. 775–777, Oct. 2010, doi: 10.1038/467775a.
10. D. Heaton and J. C. Carver, "Claims about the use of software engineering practices in science: A systematic literature review," *Information and Software Technology*, vol. 67, pp. 207–219, Nov. 2015, doi: 10.1016/j.infsof.2015.07.011.
11. M. Schmidberger and B. Brügge, "Need of Software Engineering Methods for High Performance Computing Applications," in *2012 11th International Symposium on Parallel and Distributed Computing*, Jun. 2012, pp. 40–46. doi: 10.1109/ISPDC.2012.14.
12. U. Nangia and D. S. Katz, "Track 1 Paper: Surveying the U.S. National Postdoctoral Association Regarding Software Use and Training in Research," p. 4361681 Bytes, 2017, doi: 10.6084/M9.FIGSHARE.5328442.
13. G. Wilson, "Software Carpentry: Getting Scientists to Write Better Code by Making Them More Productive," *Computing in Science & Engineering*, vol. 8, no. 6, 2006, doi: 10.1109/MCSE.2006.122.
14. M. Reynolds, "Science Is Full of Errors. Bounty Hunters Are Here to Find Them," *Wired*. Accessed: Apr. 24, 2025. [Online]. Available: <https://www.wired.com/story/bounty-hunters-are-here-to-save-academia-bug-bounty/>
15. W. Knight, "Sloppy Use of Machine Learning Is Causing a 'Reproducibility Crisis' in Science," *Wired*. Accessed: Apr. 24, 2025. [Online]. Available: <https://www.wired.com/story/machine-learning-reproducibility-crisis/>
16. X. He et al., "Author Correction: Temporal dynamics in viral shedding and transmissibility of COVID-19," *Nat Med*, vol. 26, no. 9, pp. 1491–1493, Sep. 2020, doi: 10.1038/s41591-020-1016-z.
17. P. Ashcroft et al., "COVID-19 infectivity profile correction," *Swiss Medical Weekly*, vol. 150, no. 3132, Art. no. 3132, Aug. 2020, doi: 10.4414/smw.2020.20336.
18. S. Iglesias et al., "Hierarchical Prediction Errors in Midbrain and Basal Forebrain during Sensory Learning," *Neuron*, vol. 101, no. 6, pp. 1196–1201, Mar. 2019, doi: 10.1016/j.neuron.2019.03.001.
19. A. Marcus, "Doing the right thing: Psychology researchers retract paper three days after learning of coding error," *Retraction Watch*. Accessed: Sep. 04, 2025. [Online]. Available: <https://retractionwatch.com/2019/08/13/doing-the-right-thing-psychology-researchers-retract-paper-three-days-after-learning-of-coding-error/>
20. T. Dolk, C. Freigang, J. Bogon, and G. Dreisbach, "RETRACTED: Auditory (dis-)fluency triggers sequential processing adjustments," *Acta Psychologica*, vol. 191, pp. 69–75, Nov. 2018, doi: 10.1016/j.actpsy.2018.08.021.
21. K. E. Henson, R. Jagsi, D. Cutter, P. McGale, C. Taylor, and S. C. Darby, "RETRACTED: Inferring the Effects of Cancer Treatment: Divergent Results From Early Breast Cancer Trialists' Collaborative Group Meta-Analyses of Randomized Trials and Observational Data From SEER Registries," *JCO*, vol. 34, no. 8, pp. 803–809, Mar. 2016, doi: 10.1200/JCO.2015.62.0294.
22. "In Sickness and in Health? Physical Illness as a Risk Factor for Marital Dissolution in Later Life - Amelia Karraker, Kenzie Latham, 2015." Accessed: Sep. 04, 2025. [Online]. Available: <https://journals.sagepub.com/doi/abs/10.1177/0022146515596354>
23. P. J. Mandhane, "Notice of Retraction: Hahn LM, et al. Post-COVID-19 Condition in Children. *JAMA Pediatrics*. 2023;177(11):1226-1228.," *JAMA Pediatr*, vol. 178, no. 10, pp. 1085–1086, Oct. 2024, doi: 10.1001/jamapediatrics.2024.2996.
24. C. Reinhart and K. Rogoff, "Errata: 'Growth in A Time of Debt,'" *Harvard University*, 2013, Accessed: Sep. 04, 2025. [Online]. Available: https://carmenreinhardt.com/wp-content/uploads/2020/02/36_data.pdf

25. K. Wieggers, "Humanizing Peer Reviews." Accessed: Apr. 25, 2025. [Online]. Available: https://web.archive.org/web/20060315135514/http://www.processimpact.com/articles/humanizing_reviews.html
26. J. Atwood, "Code Reviews: Just Do It," Coding Horror. Accessed: Apr. 17, 2025. [Online]. Available: <https://blog.codinghorror.com/code-reviews-just-do-it/>
27. G. Miller, "A Scientist's Nightmare: Software Problem Leads to Five Retractions," *Science*, vol. 314, no. 5807, pp. 1856–1857, Dec. 2006, doi: 10.1126/science.314.5807.1856.
28. MIT, "The Missing Semester of Your CS Education," Missing Semester. Accessed: Apr. 25, 2025. [Online]. Available: <https://missing.csail.mit.edu/>
29. G. Balaban, I. Grytten, K. D. Rand, L. Scheffer, and G. K. Sandve, "Ten simple rules for quick and dirty scientific programming," *PLOS Computational Biology*, vol. 17, no. 3, p. e1008549, Mar. 2021, doi: 10.1371/journal.pcbi.1008549.
30. R. Green, "How To Write Unmaintainable Code." Accessed: Apr. 25, 2025. [Online]. Available: <https://www.doc.ic.ac.uk/~susan/475/unmain.html>
31. "Motivation: Reproducible research documentation." Accessed: Apr. 25, 2025. [Online]. Available: <https://coderefinery.github.io/reproducible-research/motivation/>
32. R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*, 1st edition. Upper Saddle River, NJ: Pearson, 2008.
33. M. Lynch and published, "Rules for Writing Software Tutorials." Accessed: Apr. 26, 2025. [Online]. Available: <https://refactoringenglish.com/>
34. "Software Carpentry Lessons," Software Carpentry. Accessed: Apr. 26, 2025. [Online]. Available: <https://software-carpentry.org/lessons/>
35. "CodeRefinery." Accessed: Apr. 26, 2025. [Online]. Available: <https://coderefinery.org/lessons/>
36. UNIVERSE-HPC, "Byte-sized RSE," UNIVERSE-HPC. Accessed: Apr. 26, 2025. [Online]. Available: <http://www.universe-hpc.ac.uk/events/byte-sized-rse/>
37. J. Hastings, K. Haug, and C. Steinbeck, "Ten recommendations for software engineering in research," *GigaSci*, vol. 3, no. 1, p. 31, Dec. 2014, doi: 10.1186/2047-217X-3-31.
38. M. Lynch, "How to Do Code Reviews Like a Human (Part One)." Accessed: Apr. 26, 2025. [Online]. Available: <https://mtlynch.io/human-code-reviews-1/>
39. M. Lynch, "How to Do Code Reviews Like a Human (Part Two)." Accessed: Apr. 26, 2025. [Online]. Available: <https://mtlynch.io/human-code-reviews-2/>
40. M. Lynch, "How to Make Your Code Reviewer Fall in Love with You." Accessed: Apr. 26, 2025. [Online]. Available: <https://mtlynch.io/code-review-love/>
41. S. Tatham, "Code review antipatterns." Accessed: Apr. 26, 2025. [Online]. Available: <https://www.chiark.greenend.org.uk/~sgtatham/quasiblog/code-review-antipatterns/>
42. E. R. Ivimey--Cook et al., "Implementing code review in the scientific workflow: Insights from ecology and evolutionary biology," *Journal of Evolutionary Biology*, vol. 36, no. 10, pp. 1347–1356, Oct. 2023, doi: 10.1111/jeb.14230.
43. A. Rokem, "Ten simple rules for scientific code review," *PLOS Computational Biology*, vol. 20, no. 9, p. e1012375, Sep. 2024, doi: 10.1371/journal.pcbi.1012375.
44. A. M. Vable, S. F. Diehl, and M. M. Glymour, "Code Review as a Simple Trick to Enhance Reproducibility, Accelerate Learning, and Improve the Quality of Your Team's Research," *American Journal of Epidemiology*, vol. 190, no. 10, pp. 2172–2177, Oct. 2021, doi: 10.1093/aje/kwab092.
45. T. Gee, *What to Look for in a Code Review*. Leanpub, 2015. Accessed: Apr. 17, 2025. [Online]. Available: <https://leanpub.next/whattolookforinacodereview>
46. "What to look for in a code review," eng-practices. Accessed: Apr. 17, 2025. [Online]. Available: <https://google.github.io/eng-practices/review/reviewer/looking-for.html>
47. "Common pitfalls and recommended practices," scikit-learn. Accessed: Apr. 17, 2025. [Online]. Available: https://scikit-learn.org/stable/common_pitfalls.html
48. J. Barton, "awesome-code-review," GitHub. Accessed: Apr. 17, 2025. [Online]. Available: <https://github.com/joho/awesome-code-review/blob/main/readme.md>

49. B. A. Nosek, C. R. Ebersole, A. C. DeHaven, and D. T. Mellor, "The preregistration revolution," *Proceedings of the National Academy of Sciences*, vol. 115, no. 11, pp. 2600–2606, Mar. 2018, doi: 10.1073/pnas.1708274114.
50. A. Brodeur, N. M. Cook, J. S. Hartley, and A. Heyes, "Do Pre-Registration and Pre-analysis Plans Reduce p-Hacking and Publication Bias?," GLO Discussion Paper, Working Paper 1147, 2022. Accessed: Sep. 02, 2025. [Online]. Available: <https://www.econstor.eu/handle/10419/262738>
51. C. Pornprasit and C. Tantithamthavorn, "Fine-tuning and prompt engineering for large language models-based code review automation," *Information and Software Technology*, vol. 175, p. 107523, Nov. 2024, doi: 10.1016/j.infsof.2024.107523.
52. Y. Yu et al., "Fine-Tuning Large Language Models to Improve Accuracy and Comprehensibility of Automated Code Review," *ACM Trans. Softw. Eng. Methodol.*, vol. 34, no. 1, p. 14:1-14:26, Dec. 2024, doi: 10.1145/3695993.
53. M. A. Haider, A. B. Mostofa, S. S. B. Mosaddek, A. Iqbal, and T. Ahmed, "Prompting and Fine-tuning Large Language Models for Automated Code Review Comment Generation," Nov. 15, 2024, *arXiv: arXiv:2411.10129*. doi: 10.48550/arXiv.2411.10129.
54. J. Lu, L. Yu, X. Li, L. Yang, and C. Zuo, "LLaMA-Reviewer: Advancing Code Review Automation with Large Language Models through Parameter-Efficient Fine-Tuning," in *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*, Oct. 2023, pp. 647–658. doi: 10.1109/ISSRE59848.2023.00026.
55. S. Kwon, S. Lee, T. Kim, D. Ryu, and J. Baik, "Exploring LLM-based Automated Repairing of Ansible Script in Edge-Cloud Infrastructures," *Journal of Web Engineering*, pp. 889–912, Dec. 2023, doi: 10.13052/jwe1540-9589.2263.
56. Z. Li et al., "Automating code review activities by large-scale pre-training," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, in ESEC/FSE 2022. New York, NY, USA: Association for Computing Machinery, Nov. 2022, pp. 1035–1047. doi: 10.1145/3540250.3549081.
57. L. Fan, J. Liu, Z. Liu, D. Lo, X. Xia, and S. Li, "Exploring the Capabilities of LLMs for Code Change Related Tasks," Jul. 03, 2024, *arXiv: arXiv:2407.02824*. doi: 10.48550/arXiv.2407.02824.
58. G. F. Martins, E. C. M. Firmino, and V. P. De Mello, "The Use of Large Language Model in Code Review Automation: An Examination of Enforcing SOLID Principles," in *Artificial Intelligence in HCI*, H. Degen and S. Ntoa, Eds., Cham: Springer Nature Switzerland, 2024, pp. 86–97. doi: 10.1007/978-3-031-60615-1_6.
59. X. Tang et al., "CodeAgent: Autonomous Communicative Agents for Code Review," in *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, Y. Al-Onaizan, M. Bansal, and Y.-N. Chen, Eds., Miami, Florida, USA: Association for Computational Linguistics, Nov. 2024, pp. 11279–11313. doi: 10.18653/v1/2024.emnlp-main.632.
60. Z. Zhao, Z. Xu, J. Zhu, P. Di, Y. Yao, and X. Ma, "The Right Prompts for the Job: Repair Code-Review Defects with Large Language Model," Dec. 29, 2023, *arXiv: arXiv:2312.17485*. doi: 10.48550/arXiv.2312.17485.
61. E. Gibney, "AI tools are spotting errors in research papers: inside a growing movement," *Nature*, Mar. 2025, doi: 10.1038/d41586-025-00648-5.
62. C. H. Krag et al., "Large language models for abstract screening in systematic- and scoping reviews: A diagnostic test accuracy study," Oct. 02, 2024, *medRxiv*. doi: 10.1101/2024.10.01.24314702.
63. M. Nashaat and J. Miller, "Towards Efficient Fine-Tuning of Language Models With Organizational Data for Automated Software Review," *IEEE Transactions on Software Engineering*, vol. 50, no. 9, pp. 2240–2253, Sep. 2024, doi: 10.1109/TSE.2024.3428324.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.