Article

Not peer-reviewed version

Structured Code Review in Mental Health Research

Inês Pereira [*] , Herman Galioulline [*] , Réka Enz , Alexander Hess , Matthias Müller-Schrader , Dina von Werder , Imre Kertesz , Florian Schönleitner , Stephanie Mellor , Katja Brand , Lars Kasper , Lilian Weber , Frederike Petzschner , Sandra Iglesias , Jakob Heinzle , Klaas Stephan

Posted Date: 10 November 2025

doi: 10.20944/preprints202510.0133.v2

Keywords: robust science; scientific integrity; reproducibility; software engineering; translational neuromodeling; computational psychiatry; neuroimaging

Article

Structured Code Review in Mental Health Research

Inês Pereira [1,*], Herman Galioulline [1,*], Réka Enz [1,2,3], Alexander J. Hess [1], Matthias Müller-Schrader [1,4], Dina von Werder [1], Imre Kertesz [1], Florian M. Schönleitner [1], Stephanie Mellor [1,5,6], Katja Brand [1], Lars Kasper [7,8], Lilian A. Weber [9,10,11], Frederike H. Petzschner [12,13], Sandra Iglesias [1], Jakob Heinzle [1] and Klaas Enno Stephan [1,14]

1 Translational Neuromodeling Unit (TNU), Institute for Biomedical Engineering, University of Zurich & ETH Zurich, Zurich, Switzerland
2 Department of Neurology, Schulthess Clinic, Zürich, Switzerland
3 Department of Health Sciences and Technology (DHEST), ETH Zurich, Zurich, Switzerland
4 Fields Institute for Research in Mathematical Sciences, Toronto, Ontario, Canada
5 Department of Imaging Neuroscience, UCL Queen Square Institute of Neurology, University College London, London, United Kingdom
6 Spinal Cord Injury Center, Balgrist University Hospital, University of Zurich, Zurich, Switzerland
7 Krembil Brain Institute, University Health Network, Toronto, Ontario, Canada
8 Toronto Neuroimaging Facility, Department of Psychology, University of Toronto, Toronto, Ontario, Canada
9 Department of Psychiatry, University of Oxford, Oxford, UK
10 Wellcome Centre for Integrative Neuroimaging (WIN), Department of Experimental Psychology, University of Oxford, Oxford, UK
11 Institute for Cognitive Science, Osnabrueck University, Osnabrueck, Germany
12 Cognitive and Psychological Sciences, Brown University, Providence, RI, USA
13 Robert J. and Nancy D. Carney Institute for Brain Science, Brown University, Providence, RI, USA
14 Max Planck Institute for Metabolism Research, Cologne, Germany
* Correspondence: ines.pereira@alumni.ethz.ch (I.P.); hermang@ethz.ch (H.G.)

Abstract

Errors in scientific code pose a significant risk to the accuracy of research. Yet, formal code review remains uncommon in academia. This is a particular challenge for interdisciplinary fields, such as mental health research, which increasingly rely on computational approaches. This paper presents a pragmatic experience-based framework for code review procedures in mental health research. In order to facilitate practical implementation, it includes a structured checklist for identifying common coding issues, from data handling errors to flawed statistical analyses. We discuss barriers to introducing code review and how to overcome them, revisit best practices from software engineering, and highlight the emerging role of large language models in automating aspects of code review. We argue that, despite perceived costs, code review significantly enhances the reliability of research results 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: robust science; scientific integrity; reproducibility; software engineering; translational neuromodeling; computational psychiatry; neuroimaging

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 from the recent past where scientific conclusions have been affected by coding errors[14–23]. These examples include 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 investigations 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 mental health research, a scientific field that increasingly relies on computational approaches and brings together scientists from very different backgrounds. In particular, we draw on our experience in computational subfields of mental health, 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 in mental health research because they have a long tradition of using sophisticated computational methods (e.g., generative models, machine learning) and numerous data types (e.g. neuroimaging, electrophysiology, behavior, self-report, clinical outcomes).

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 in Zurich — 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 any articles for publication. 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 initial 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 implications 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 incorrect data files.
- Errors in the calculations of test statistics.
- Indexing errors (e.g., in vector operations, during data anonymisation) leading to row-shifted data, resulting in 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 discovered, 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 scrutinized and criticized can be a difficult process, even when the reviewer is a colleague with good intentions. In order to increase acceptance of code review in our unit, a culture of openness is encouraged, and polite and constructive feedback is 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 their future careers, 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 been developing a structured checklist based on our own experience, discussions within the lab, and external sources. The checklist is neither intended as a rigid standard nor does it claim to cover all possible aspects of code review; instead, it is meant to serve as a practical guide that lowers the barrier for research groups to introduce code review and helps them prioritize the most important steps in this process. By offering this pragmatic checklist, this paper aims to encourage a culture of regular and collaborative code review that ultimately increases the trustworthiness and robustness of mental health research.

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[49,50]. The list is not intended to be exhaustive or final, but to provide a structured and adaptable reference to researchers and laboratories not yet familiar with code review.

Fundamental checks: basic correctness, reproducibility, usability

General points

- Documentation:
  - Are there clearly specified instructions or a structured README file?
  - Are operating system(s), dependencies, versions, and required toolboxes specified? Is there a separate configuration file (e.g., yml, .txt)?
  - Can you follow the setup instructions?
  - Are inline comments and function documentation present where necessary (e.g., more complex parts of the code)?
- Version control & dependencies:
  - Is version-control software (e.g., Git) used?
  - 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 you 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, indications of overfitting) that suggests further checks are necessary?

Empirical data analyses

Detailed checks for **1-3 randomly selected subjects**:

- If an analysis plan exists: Do the main scripts align with the 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)
  - When working with tabular data, are the correct entries extracted (i.e. is the indexing of rows/columns correct)?
- 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 dimensions and orientations (e.g., transposes) of datatypes (e.g., arrays, structures) 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 (e.g., in the correct order, plausible time intervals)?
  - 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) clearly 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 defined or stored in a manner that they can be reused?
- Sanity checks: Do expected patterns emerge in the results (e.g., typical 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 or available manuscripts?
- Does the implemented model behave as expected on controlled synthetic datasets where the ground truth is known?
- Does running the same analysis multiple times with the same seeds produce identical results?
- For generative models, are the priors specified as intended and documented?
- 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 optimization process (e.g. objective function value across iterations), as a basis for detecting potential problems?
  - If applicable: Does the code include measures to avoid being trapped in local extrema (e.g. multi-start approaches)?

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 (e.g., if the training data are standardized, is the test set standardized as well?)?
- Are the assumptions of the model met (e.g., is the target variable mean-centered and the features standardized when using an Elastic Net regression model?)?

Advanced checks: performance, maintainability, robustness

General points

- Dealing with data and large files:
  - Are code and data fully separated?
  - Are non-code files (e.g., data files and other large files) excluded from version control (e.g., through a .gitignore or equivalent mechanism)?
- Code optimization:
  - Can the code be simplified? For example:
    - Are there instances of duplication? If so, should it be refactored or is this acceptable at this stage?
    - Is there any unused code? Check with the code author if it can be deleted. If it should be kept, clearly document why.
  - Are there any obvious performance enhancements (e.g., vectorization, parallelization over subjects)?
  - Are there obvious opportunities for optimizing data structures (e.g. 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)?
  - Check test coverage. Do the tests cover:

        - The most important parts of the code?
        - Potentially confusing or complicated sections of code?
        - Edge cases?
    - Are there any cases that have not been considered and should be covered by tests?

Empirical data analyses

- Is the analysis pipeline end-to-end: from raw data to final results and associated figures?
- Is the analysis pipeline a "one-click" pipeline, i.e., is there one script or command that runs the entire analysis?
- Are resulting statistics at trial-level, subject-level and group level stored and available for future (re-)analysis?
- Are corrections for multiple comparisons correctly applied in statistical analyses?

Computational modeling

- For gradient-based optimization: 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?
- Does the model handle extreme values 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 for 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[51,52]. 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 should 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 research contexts, such as work with direct clinical or societal impact, 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. 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[53–56]. These systems are increasingly capable of generating readable comments, detecting latent defects, and even repairing code based on review feedback[57–62]. Of particular interest in scientific contexts is their emerging ability to retroactively analyze published code repositories (as well as the associated papers[63]) 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[64,65]. 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 our experiences with challenges encountered in computational subfields of mental health research, in particular Translational Neuromodeling and Computational Psychiatry, but will also broadly apply to other areas of neuroscience. 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 considering the potential of emerging tools such as large language models, we hope to contribute to a more systematic and sustainable review culture in academic research. 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.

Acknowledgements: We would like to thank all present and past members of the Translational Neuromodeling Unit for their contributions to discussions and feedback about the code review process. This work was supported by funding from the René and Susanne Braginsky Foundation, ETH Foundation, and the University of Zurich (K.E.S.).

References

1. Soergel, D. A. W. Rampant software errors may undermine scientific results. Preprint at https://doi.org/10.12688/f1000research.5930.2 (2015).
2. McConnell, S. *Code Complete*. (Microsoft Press, Redmond, Washington, 2004).
3. Arvanitou, E.-M., Ampatzoglou, A., Chatzigeorgiou, A. & Carver, J. C. Software engineering practices for scientific software development: A systematic mapping study. *Journal of Systems and Software* **172**, 110848 (2021).
4. Storer, T. Bridging the Chasm: A Survey of Software Engineering Practice in Scientific Programming. *ACM Comput. Surv.* **50**, 1–32 (2018).
5. Woolston, C. Why science needs more research software engineers. *Nature* https://doi.org/10.1038/d41586-022-01516-2 (2022) doi:10.1038/d41586-022-01516-2.
6. RSECon 2025. *RSECon25* https://rsecon25.society-rse.org/ (2025).
7. Nordic RSE. *Nordic RSE* https://nordic-rse.org/.
8. nl-rse. *nl-rse* https://nl-rse.org/.
9. Merali, Z. Computational science: ...Error. *Nature* **467**, 775–777 (2010).
10. Heaton, D. & Carver, J. C. Claims about the use of software engineering practices in science: A systematic literature review. *Information and Software Technology* **67**, 207–219 (2015).
11. Schmidberger, M. & Brügge, B. Need of Software Engineering Methods for High Performance Computing Applications. in *2012 11th International Symposium on Parallel and Distributed Computing* 40–46 (2012). doi:10.1109/ISPDC.2012.14.
12. Nangia, U. & Katz, D. S. Track 1 Paper: Surveying the U.S. National Postdoctoral Association Regarding Software Use and Training in Research. 4361681 Bytes (2017) doi:10.6084/M9.FIGSHARE.5328442.
13. Wilson, G. Software Carpentry: Getting Scientists to Write Better Code by Making Them More Productive. *Computing in Science & Engineering* **8**, (2006).
14. Reynolds, M. Science Is Full of Errors. Bounty Hunters Are Here to Find Them. *Wired*.
15. Knight, W. Sloppy Use of Machine Learning Is Causing a 'Reproducibility Crisis' in Science. *Wired*.
16. He, X. *et al.* Author Correction: Temporal dynamics in viral shedding and transmissibility of COVID-19. *Nat Med* **26**, 1491–1493 (2020).
17. Ashcroft, P. *et al.* COVID-19 infectivity profile correction. *Swiss Medical Weekly* **150**, w20336–w20336 (2020).
18. Iglesias, S. *et al.* Hierarchical Prediction Errors in Midbrain and Basal Forebrain during Sensory Learning. *Neuron* **101**, 1196–1201 (2019).
19. Marcus, A. Doing the right thing: Psychology researchers retract paper three days after learning of coding error. *Retraction Watch* https://retractionwatch.com/2019/08/13/doing-the-right-thing-psychology-researchers-retract-paper-three-days-after-learning-of-coding-error/ (2019).
20. Dolk, T., Freigang, C., Bogon, J. & Dreisbach, G. RETRACTED: Auditory (dis-)fluency triggers sequential processing adjustments. *Acta Psychologica* **191**, 69–75 (2018).
21. Henson, K. E. *et al.* 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* **34**, 803–809 (2016).
22. In Sickness and in Health? Physical Illness as a Risk Factor for Marital Dissolution in Later Life - Amelia Karraker, Kenzie Latham, 2015. https://journals.sagepub.com/doi/abs/10.1177/0022146515596354.
23. Mandhane, P. J. Notice of Retraction: Hahn LM, et al. Post–COVID-19 Condition in Children. JAMA Pediatrics. 2023;177(11):1226-1228. *JAMA Pediatr* **178**, 1085–1086 (2024).
24. Reinhart, C. & Rogoff, K. Errata: "Growth in A Time of Debt". *Harvard University* https://carmenreinhart.com/wp-content/uploads/2020/02/36_data.pdf (2013).

25. Wiegers, K. Humanizing Peer Reviews. https://web.archive.org/web/20060315135514/http://www.processimpact.com/articles/humanizing_reviews.html (2006).
26. Atwood, J. Code Reviews: Just Do It. *Coding Horror* https://blog.codinghorror.com/code-reviews-just-do-it/ (2006).
27. Miller, G. A Scientist's Nightmare: Software Problem Leads to Five Retractions. *Science* **314**, 1856–1857 (2006).
28. MIT. The Missing Semester of Your CS Education. *Missing Semester* https://missing.csail.mit.edu/.
29. Balaban, G., Grytten, I., Rand, K. D., Scheffer, L. & Sandve, G. K. Ten simple rules for quick and dirty scientific programming. *PLOS Computational Biology* **17**, e1008549 (2021).
30. Green, R. How To Write Unmaintainable Code. https://www.doc.ic.ac.uk/~susan/475/unmain.html.
31. Motivation — Reproducible research documentation. https://coderefinery.github.io/reproducible-research/motivation/.
32. Martin, R. C. *Clean Code: A Handbook of Agile Software Craftsmanship*. (Pearson, Upper Saddle River, NJ, 2008).
33. Lynch, M. & published. Rules for Writing Software Tutorials. https://refactoringenglish.com/ (2025).
34. Software Carpentry Lessons. *Software Carpentry* https://software-carpentry.org/lessons/.
35. CodeRefinery. https://coderefinery.org/lessons/.
36. UNIVERSE-HPC. Byte-sized RSE. *UNIVERSE-HPC* http://www.universe-hpc.ac.uk/events/byte-sized-rse/.
37. Hastings, J., Haug, K. & Steinbeck, C. Ten recommendations for software engineering in research. *GigaSci* **3**, 31 (2014).
38. Lynch, M. How to Do Code Reviews Like a Human (Part One). https://mtlynch.io/human-code-reviews-1/ (2017).
39. Lynch, M. How to Do Code Reviews Like a Human (Part Two). https://mtlynch.io/human-code-reviews-2/ (2017).
40. Lynch, M. How to Make Your Code Reviewer Fall in Love with You. https://mtlynch.io/code-review-love/ (2020).
41. Tatham, S. Code review antipatterns. https://www.chiark.greenend.org.uk/~sgtatham/quasiblog/code-review-antipatterns/ (2024).
42. Ivimey-Cook, E. R. *et al.* Implementing code review in the scientific workflow: Insights from ecology and evolutionary biology. *Journal of Evolutionary Biology* **36**, 1347–1356 (2023).
43. Rokem, A. Ten simple rules for scientific code review. *PLOS Computational Biology* **20**, e1012375 (2024).
44. Vable, A. M., Diehl, S. F. & Glymour, M. M. Code Review as a Simple Trick to Enhance Reproducibility, Accelerate Learning, and Improve the Quality of Your Team's Research. *American Journal of Epidemiology* **190**, 2172–2177 (2021).
45. Gee, T. *What to Look for in a Code Review*. (Leanpub, 2015).
46. What to look for in a code review. *eng-practices* https://google.github.io/eng-practices/review/reviewer/looking-for.html.
47. Common pitfalls and recommended practices. *scikit-learn* https://scikit-learn.org/stable/common_pitfalls.html.
48. Barton, J. awesome-code-review. *GitHub* https://github.com/joho/awesome-code-review/blob/main/readme.md.
49. Bennett, D., Silverstein, S. M. & Niv, Y. The Two Cultures of Computational Psychiatry. *JAMA Psychiatry* **76**, 563 (2019).
50. Breiman, L. Statistical Modeling: The Two Cultures (with comments and a rejoinder by the author). *Statistical Science* **16**, 199–231 (2001).
51. Nosek, B. A., Ebersole, C. R., DeHaven, A. C. & Mellor, D. T. The preregistration revolution. *Proceedings of the National Academy of Sciences* **115**, 2600–2606 (2018).
52. Brodeur, A., Cook, N. M., Hartley, J. S. & Heyes, A. *Do Pre-Registration and Pre-Analysis Plans Reduce p-Hacking and Publication Bias?* https://www.econstor.eu/handle/10419/262738 (2022).

53. Pornprasit, C. & Tantithamthavorn, C. Fine-tuning and prompt engineering for large language models-based code review automation. *Information and Software Technology* **175**, 107523 (2024).
54. Yu, Y. *et al.* Fine-Tuning Large Language Models to Improve Accuracy and Comprehensibility of Automated Code Review. *ACM Trans. Softw. Eng. Methodol.* **34**, 14:1-14:26 (2024).
55. Haider, M. A., Mostofa, A. B., Mosaddek, S. S. B., Iqbal, A. & Ahmed, T. Prompting and Fine-tuning Large Language Models for Automated Code Review Comment Generation. Preprint at https://doi.org/10.48550/arXiv.2411.10129 (2024).
56. Lu, J., Yu, L., Li, X., Yang, L. & Zuo, C. 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)* 647–658 (2023). doi:10.1109/ISSRE59848.2023.00026.
57. Kwon, S., Lee, S., Kim, T., Ryu, D. & Baik, J. Exploring LLM-based Automated Repairing of Ansible Script in Edge-Cloud Infrastructures. *Journal of Web Engineering* 889–912 (2023) doi:10.13052/jwe1540-9589.2263.
58. Li, Z. *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* 1035–1047 (Association for Computing Machinery, New York, NY, USA, 2022). doi:10.1145/3540250.3549081.
59. Fan, L. *et al.* Exploring the Capabilities of LLMs for Code Change Related Tasks. Preprint at https://doi.org/10.48550/arXiv.2407.02824 (2024).
60. Martins, G. F., Firmino, E. C. M. & De Mello, V. P. The Use of Large Language Model in Code Review Automation: An Examination of Enforcing SOLID Principles. in *Artificial Intelligence in HCI* (eds Degen, H. & Ntoa, S.) 86–97 (Springer Nature Switzerland, Cham, 2024). doi:10.1007/978-3-031-60615-1_6.
61. Tang, X. *et al.* CodeAgent: Autonomous Communicative Agents for Code Review. in *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing* (eds Al-Onaizan, Y., Bansal, M. & Chen, Y.-N.) 11279–11313 (Association for Computational Linguistics, Miami, Florida, USA, 2024). doi:10.18653/v1/2024.emnlp-main.632.
62. Zhao, Z. *et al.* The Right Prompts for the Job: Repair Code-Review Defects with Large Language Model. Preprint at https://doi.org/10.48550/arXiv.2312.17485 (2023).
63. Gibney, E. AI tools are spotting errors in research papers: inside a growing movement. *Nature* https://doi.org/10.1038/d41586-025-00648-5 (2025) doi:10.1038/d41586-025-00648-5.
64. Krag, C. H. *et al.* Large language models for abstract screening in systematic- and scoping reviews: A diagnostic test accuracy study. 2024.10.01.24314702 Preprint at https://doi.org/10.1101/2024.10.01.24314702 (2024).
65. Nashaat, M. & Miller, J. Towards Efficient Fine-Tuning of Language Models With Organizational Data for Automated Software Review. *IEEE Transactions on Software Engineering* **50**, 2240–2253 (2024).