Code review in practice: A checklist for computational reproducibility and collaborative research in ecology and evolution

Friederike Hillemann[1], Joseph B. Burant[2], Antica Čulina[3], Stefan J. G. Vriend[4]

[1]Department of Animal Ecology, Netherlands Institute of Ecology, Wageningen, The Netherlands
f.hillemann@web.de
ORCiD: 0000-0002-8992-0676

[2]Department of Animal Ecology, Netherlands Institute of Ecology, Wageningen, The Netherlands
j.burant@nioo.knaw.nl
ORCiD: 0000-0002-0713-3100

[3]Ruder Boskovic Institute, Zagreb, Croatia and Department of Animal Ecology, Netherlands Institute of Ecology, Wageningen, The Netherlands
aculina@irb.hr
ORCiD: 0000-0003-2910-8085

[4]LTER-LIFE and Department of Aquatic Ecology, Netherlands Institute of Ecology, Wageningen, The Netherlands
s.vriend@nioo.knaw.nl
ORCiD: 0000-0002-9006-5988

Data & Code Availability

This manuscript did not generate or use any data or code.

Author Contributions

Conceptualisation: JBB, FH, SJGV, AC

Writing - Original Draft: FH

Writing - Review & Editing: all authors

Visualisation: JBB, FH, SJGV

Funding acquisition: JBB, AC, SJGV

Conflicts of Interest

The authors declare no competing interests, financial or otherwise.

Acknowledgements

We are grateful to Amélie Fargevieille and Haneul Jang for helpful feedback to the checklist, and to Ed R. Ivimey-Cook, Joel L. Pick, and Saras M. Windecker for their encouragement and early conversations that helped shape this work.

Funding

FH was funded by the NWO Open Science Fund (2023) from the Dutch Research Council (NWO), grant number NWO OSF23.1.025, project title: CoreBirds: Connecting Open Research outputs in the Ecology of Birds, awarded to Marcel Visser (applicant), and JBB, AC, SJGV (team members).

Abstract

Ensuring that research, along with its data and code, is credible and remains accessible is crucial for advancing scientific knowledge—especially in ecology and evolutionary biology, where the climate crisis and biodiversity loss accelerate and demand urgent, transparent science. Yet, code is rarely shared alongside scientific publications, and when it is, unclear implementation and insufficient documentation often make it difficult to use. Code review—whether as self-assessment or peer review—can improve key aspects of code quality: reusability, i.e., ensuring technical functionality and that the code is well-documented, and validity, i.e., ensuring the code implements the intended analyses faithfully. While assessing validity requires domain expertise for methodological assessment, code review for reusability can be conducted by anyone with basic understanding of programming practices. Here, we introduce a checklist-based, customisable approach to code review that focuses on reusability. Informed by best practices in software development and recommendations from commentary pieces and blog posts, the checklist organises specific review prompts around seven key attributes of high-quality reusable scientific code: Reporting, Running, Reliability, Reproducibility, Robustness, Readability, and Release. By defining and structuring these principles of code review and turning them into a practical tool, our template guides through a systematic evaluation that is also flexible to be tailored to specific needs. This includes providing researchers with a clear path to proactively improve their own code. Ultimately, this approach to code review aims to reinforce reproducible coding practices, and strengthens both the credibility and collaborative potential of research.

Keywords

1. Research Software
2. Code Quality
3. Computational Reproducibility
4. Open Science
5. Collaborative Research

Introduction: Code as scientific output

Code-based pipelines for scientific data processing and analysis have become standard in the Life Sciences, supporting tasks such as file management, statistical modelling, visualisation, and generating reproducible reports (Perkel 2016, Abdill et al. 2024). As such, scientific code is not only a tool but a core component of the research workflow and output, and should be shared and peer-reviewed like other methodological details, to ensure research integrity and reproducibility (Ivimey-Cook et al. 2023).

In the face of global challenges such as climate change, ensuring that science is transparent and cumulative is not only good practice but an ethical obligation, and reusable code and data are essential components of this responsibility (Sandve et al. 2013; Bledsoe et al. 2022; Gomes 2025). At the same time, unverifiable research risks becoming an unstable foundation for future research and fuelling the ongoing crisis of confidence in science.

The Open Science movement has promoted the publication of data and code, shifting norms towards treating methods, including data-processing and analysis scripts, as research outputs worthy of recognition and review. While several journals now encourage or mandate code availability, policies suggested to improve the reproducibility potential (Walters 2020; Sánchez-Tójar et al. 2025), compliance remains low (Ivimey-Cook et al. 2025). Most articles do not share code, and available code is often poorly documented and unusable (Kellner et al. 2025; Culina et al. 2020). Journal policies have largely prioritised transparency, with minimal expectations for usability, rather than fostering practices that make code genuinely reusable. Yet, the benefits of code sharing and code review extend beyond transparency of methods and improved code quality; they promote a culture of cooperation and collaboration, and benefit individual researchers by providing opportunities for feedback and professional development (Culina et al. 2020), and by increasing citation potential (Maitner et al. 2023).

Still, sharing code publicly—and especially for review—can be daunting. Many researchers hesitate, citing concerns about intellectual property, the additional effort of documentation, or fear of critique (Gomes et al. 2022). In fields such as ecology and evolutionary biology, analytical pipelines are often custom-built to address specific questions. These scripts are usually developed by researchers without formal training in software engineering and vary considerably in style and documentation. Coding is a skill that takes time to develop, and support for training remains uneven across institutions and career stages. Researchers who are self-taught may feel especially vulnerable when exposing their code to scrutiny.

This apprehension often extends to the review process itself. Anxiety about giving and receiving feedback on code is common, and can deter engagement (Lee & Hicks 2024). To counter this, we emphasise a shift in expectations: there is no such thing as 'perfect code'—or, as others have put it, *your code is good enough to share* (Barnes 2010, Wilson et al. 2017). By reinforcing this mindset, we hope to normalise code review as a constructive and collaborative process, a professional service to others and a practical necessity for credible science, rather than an exercise in intimidating scrutiny. In doing so, we support a research culture where code is valued, improved, and reused, a practice that benefits authors, their collaborators, and the wider research community.

Learning from practices in data management and software development

Although research data and code are deeply interconnected, code is often treated as a mere tool rather than a central part of the scientific method and output, and rarely receives the same level of scrutiny and standardisation as data. However, collaborations require community conventions and quality standards. Global databases in ecology and evolution demonstrate their power for large-scale collaboration; notable examples include COMADRE for animal demography (Salguero-Gómez et al. 2016), SPI-Birds for avian ecology (Culina et al. 2020), bio-logging standardisation frameworks (Sequeira et al. 2021), and MacaqueNet for primate behavioural ecology (De Moor et al. 2025). These initiatives adhere to established data management principles such as FAIR (Findable, Accessible, Interoperable, and Reusable) and TRUST (Transparency, Responsibility, User Focus, Sustainability, and Technology), ensuring that data remain reusable.

Crucially, these initiatives all rely on code-based workflows for data processing and integration, and quality control pipelines. Given that these databases already bring together large research communities using shared data standards, they provide a strong foundation for extending FAIR and TRUST principles to code workflows to foster better documentation, reproducibility, and long-term accessibility. Reviewing and sharing code further strengthens collaboration within research communities. For instance, researchers from The Norwegian Institute for Nature Research (NINA), Norway's leading institution for applied ecological research, have developed community-led approaches to code review (Kolstad et al. 2023).

While still emerging in scientific research, code review is a long-standing practice in professional software development and computational disciplines such as engineering, where it plays a crucial role in ensuring software quality and maintainability. The foundational Fagan Inspection process, developed in the 1970s, is a structured multi-step approach that involves distinct process operations (Overview, Preparation, Inspection, Rework, and Follow-up) with clear objectives or focused tasks such as finding errors, fixing them, and ensuring all fixes are correctly applied (Fagan 1976). This method also includes communications and education as part of the inspection, ensuring that the team learns from the process. In software developing projects today, systematic code review is integrated alongside automated testing, version control, and continuous integration to catch errors, improve clarity and efficiency, and maintain good coding standards.

Scientific code review, though not yet as formalised, serves a similar role in supporting long-term sustainability of code and collaboration. Researchers can adopt key practices from software development, including thorough documentation, modular design, and structured review processes.

BOX: Code review in research context — Scope and limits

Code review is the systematic evaluation of software code. The aim is to identify errors and improve the code quality. Code quality can broadly be assessed in two key aspects: reusability

(ensuring the code is functional, modular, well-documented, and licensed) and validity (ensuring the code accurately implements the reported methods without introducing errors in consecutive steps).

Code review is a key part of research validity. While manuscript peer review evaluates the scientific soundness of a study and its methods, code review ensures that the computational steps producing the results are transparent, free of errors, and reproducible. Together, these processes contribute to the credibility of research findings.

Code review is inherently context-specific. Code review primarily strengthens computational reproducibility but its focus, depth, and outcomes depend on the expertise of the reviewer, the stage at which the review occurs, and the specific goals of the assessment. Some reviews may prioritise technical functionality, while others focus on the code being comprehensible to a broad audience.

Code review is a tool for maintaining high research standards. Given that code is part of the scientific output, often essential to the methods and results, code review ensures that computational workflows are transparent, comprehensible, and appropriately implemented. It also promotes ethical data practices, long-term sustainability, and open research.

Code review is a means of fostering collaboration, knowledge exchange, and innovation. Engaging in code review can even help researchers refine their own coding skills and adopt or share more efficient approaches and better practices.

Code review is not a guarantee of correctness. Much like manuscript peer review, code peer review does not ensure absolute validity (Smith 2006; Drozdz & Ladomery 2024).

Code review is not an assessment of methodological choices. Depending on the specific aim of the review, code reviewers may not be familiar with the research context and instead focus solely on computational aspects. Code reviewers check whether the analysis is correctly implemented as described in the manuscript but does not determine whether the chosen analysis is appropriate for the research question—that usually remains within the scope of scientific peer review.

Code review is not a stylistic critique. Unless a standardised style guide applies, minor stylistic choices are not the focus. While consistency is important, clarity, accuracy, and documentation take priority over stylistic preferences.

Code review is not code revision. Reviewers provide feedback, but responsibility for implementing changes remains with the code's authors.

Putting code review in practice: A practical checklist

Existing primers to code review—such as the '4-Rs' framework (Running, Reporting, Reliability, and Reproducibility; Ivimey-Cook et al. 2023)—advocate for integrating code review into

reproducible scientific practice. Others highlight the value of kind, community-oriented review, providing principled advice for code review with emphasis on social etiquette such as inviting collaborators, mentors, and students to review, being kind, and reciprocating feedback (Rokem 2024). While conceptually rich, these resources offer limited guidance for day-to-day implementation.

To bridge this gap, we reviewed existing best-practice guidelines (Sandve et al. 2013; Cooper & Hsing 2017; Wilson et al. 2017; Barker et al. 2022; Filazzola & Lortie 2022; Jenkins et al. 2023) and developed a practical checklist researchers can use for self-assessment and peer review. We extend the 4-R framework to a 7-R guide, introducing additional dimensions of code quality (Robustness, Readability, and Release) to support a more comprehensive assessment of scientific code reusability.

The checklist provides structured but flexible prompts to guide reflection and evaluation, helping code developers and reviewers to identify areas for improvement. Though comprehensive, it is aligned with common, attainable standards. By making peer review more accessible, systematic, and standardised, we aim to help researchers improve the quality and impact of their code, and to contribute to collaborative Open research practices.

User-friendly and customisable versions of the checklist, including a PDF and a Markdown template, are available in the supplementary materials.

Reporting: Check that it does what it claims.

Code is used to solve a specific problem or perform tasks, and code review should verify whether it does what it is intended to do—or claims to do. In research contexts, this usually means assessing whether the code faithfully implements the methods outlined in the associated manuscript. All critical steps from data wrangling to specifying statistical models should be present in the code as reported—and *vice versa*, though the focus here is on reviewing code. Any discrepancies, as small as applying a different data filter, can undermine the reproducibility of the research, and necessary deviations should be documented (e.g., manual steps or unreported additional steps). Verifying that the code matches the reported methods eliminates misinterpretations due to unreported differences between documentation and implementation.

Suggested focus to guide the assessment:

Methods Alignment: Does the code implement the methods as described in the associated documentation or research outputs?
Documentation: Is there sufficient metadata (e.g., in a README file or code header) to understand and use the code independently of external documentation?

Running — Check that it works.

From typo to missing dependencies, to platform incompatibilities—various factors can prevent code from running. A key aspect of assessing code is verifying that it is executable and that it runs as expected. Code that is difficult to install, requires excessive manual intervention, or does not perform within reasonable time constraints is not user-friendly. To help with successful

setup of dependencies, tools like the *groundhog* R package (Simonsohn & Gruson 2025) can lock package versions, ensuring that the same versions are loaded each time a script is run.

Suggested focus to guide the assessment:

Functioning: Does the code run without errors from start to finish?
Dependencies: Does the code specify all required libraries/packages or is set to install automatically?
Cross-Platform Compatibility: Does the code run on a different operating system than the one it was developed on?
Run Time: Does the code provide information on run time to manage user expectations?
Complete Check: Did you run the entire code?

Reproducibility — Check that it gives consistent results.

Ensuring reproducibility allows findings to be verified independently, which is a key aspect to scientific integrity. For code to be considered reproducible, it must consistently generate the claimed functional outputs when provided with the same input data and computational conditions. This applies to both numerical outputs (e.g., statistical summaries, simulation results) and visual outputs (e.g., figures, tables). Ideally and conveniently, though not the only way to achieve reproducible research, the entire workflow is scripted and self-contained, avoiding manual interventions such as editing data in external spreadsheets. The code should explicitly document data sources, data wrangling steps, and analysis choices, to ensure that others can follow the same procedures. Dependency management extends to the computational setup. In R, running *sessionInfo()* provides a snapshot record of the current software environment, while the *packrat* package (Atkins et al., 2025) stores package copies, creating a local library of package dependencies directly within a specified analysis directory.

Suggested focus to guide the assessment:

Numerical Reproducibility: Does the code generate the same functional outputs, i.e., statistical or simulation results, when provided with identical data and parameters?
Visual Reproducibility: Does the code generate consistent visual outputs (e.g., figures, maps) across repeated executions with the same input?
Requirements: Does the code include or clearly specify all necessary data, or provide mock data
Compartmentalisation: Does the code ensure the workflow is self-contained, with all external software dependencies documented and accessible for execution in other environments?

where applicable, to enable independent reproduction?

Reliability — Check that it is built to minimise potential errors.

Reliability refers to the ability of code to consistently produce correct and expected results when given valid, well-defined inputs. However, errors can propagate through seemingly error-free code; code that runs without warnings and produces an output may still yield incorrect results. For example, this could happen when the wrong column is selected in a dataset or when a variable is inadvertently overwritten. To minimise these risks, code should include input validation and rigorous testing to ensure it works as intended. Implementing unit tests, for example, can help identify issues early by validating individual steps. By focusing on minimising potential errors and verifying that it works as intended, throughout the workflow, the likelihood of undetected issues that could compromise results is reduced.

Suggested focus to guide the assessment:

Expected Results: Does the code produce the correct type of output for each step, e.g., correct data transformations or statistical results?
Validation & Internal Checks: Does the code include safeguards such as assertions, unit tests, or manual checks to verify that key steps are performed as intended?
Warning & Error Handling: Does the code anticipate limitations related to data quality or input constraints and provide comments, warnings, or error messages?

Robustness — Check that it handles unexpected or invalid inputs gracefully.

Robustness refers to the ability of code to handle unexpected edge cases or invalid inputs gracefully, avoiding crashes or misleading results. Robust code anticipates potential problems by minimising redundancy, using generalisable structures ensuring adaptability, and producing clear error messages where needed. For example, in R, embedding file paths directly is fragile (e.g., setting the working directory with *setwd()*) whereas the here package (Müller 2020) improves portability by using relative paths within projects. Similarly, replacing repeated blocks with functions or loops makes code modular and therefore easier to debug and maintain. Functional programming principles further support robustness by structuring code into self-contained steps that do not modify global states, ensuring independent testing, or reuse and extending. Libraries such as *purrr* in R (Wickham & Henry 2025) and *toolz* in Python (Rocklin et al. 2023) promote this approach. Robust code is efficient; everything in it should be necessary for its function.

Suggested focus to guide the assessment:

Feedback: Does the code provide clear, interpretable comments/messages on potential issues?
Parameterisation: Does the code avoid hard-coding? For instance, does it use relative file paths instead of absolute ones?
Efficiency: Does the code efficiently avoid redundancy and include only what is necessary?
Functional Programming Principles: Does the code minimise global state changes using functions and pipelines (e.g., R tidyverse packages)?

Readability — Check that it is clear and clean.

Code that is effortlessly understandable, is fun to work with. Not only does it simplify collaboration, but writing neat and well-structured code is easier to maintain and reduces the likelihood of errors during the development. Code should be organised logically, with a clear structure of segments that reflect a specific purpose, and any names both within the code as well as file names should be informative, allowing users to follow the intended workflow with minimal guesswork.

Suggested focus to guide the assessment:

Organisation: Does the code follow a logical order, guiding users through the workflow and clearly conveying its function?
Modularity: Does the code consist of manageable, logical sections (e.g., functions, sections, modular scripts) that together form a coherent workflow?
Naming Conventions: Does the code use informative names for variables, functions, and objects?
Style Conventions: Does the code consistently follow a style guide, such as tidyverse style for R?

Release — Check that it's ready for sharing and reuse

Now that the code is written and reviewed, authors and contributors want others to use it. Clear instructions encourage responsible reuse and further development, fostering collaborative cultures and extending code impact. A licence is essential to specify terms for reuse; it defines how others can use, modify, and distribute the code. Without one, copyright laws automatically restrict reuse under agreements like the Berne Convention (World Intellectual Property Organization, 1979), which grants creators exclusive rights by default. A well-chosen licence provides legal clarity while ensuring contributors receive appropriate recognition (see paragraph on selecting a software licence under Beyond the Checklist: Additional Considerations). Guidelines on citation, and how users can contribute to, or seek support for the code should be provided in the metadata, along with instructions for feedback, issue reporting, and collaboration. Assigning a Persistent Identifier (PID), such as a Digital Object Identifier (DOI), makes it easier to cite the code. Further, connecting the code to other research outputs via a enhances the visibility and credibility of the work, and facilitates tracking of its impact.

Suggested focus to guide the assessment:

Contact: Do the authors or maintainers provide guidance on how to report feedback or obtain support?
Legal Permissions: Does the code include a licence specifying how it can be used, modified, shared?
Attribution: Does the code have a Persistent Identifier (e.g., Digital Object Identifier DOI), making it easy to cite and give proper credit in academic and research contexts?

Flexibility in code review and synergies

Our practical guide offers a structured approach to reviewing scientific code. While the checklist presented here focuses on reviewing the overall reusability of code, along specific domains that contribute to it—Running, Reporting, Reliability, Reproducibility, Robustness, Readability, and Release—it is not an exhaustive list of questions, nor is it the only way to categorise questions.

Improvements during code review often have synergistic effects, i.e. they often overlap and benefit multiple dimensions of code quality:

- Replacing repeated code with functions or loops strengthens Robustness in various ways: modular code is easier to maintain and modify (functional programming principles), reduced redundant execution is faster (efficiency), and functions allow for flexible reuse instead of hardcoding different inputs in repeated sections (parameterisation).
- Using relative file paths instead of hard-coded ones improves Robustness by ensuring adaptability when file locations change. It also enhances Reliability by reducing errors from incorrect paths, and strengthens Reproducibility by standardising inputs so the code runs consistently across different machines.
- Writing well-documented code enhances Readability by making it easier to understand and improves Reproducibility by removing guesswork, making findings easier to replicate. Not only will collaborators and future users appreciate it—it's also a gift to your future self!

The central role of code review in the code development cycle

Scientific code development typically progresses through several phases, from initial conceptualisation, usually by an individual researcher (*create*), to distribution among collaborators (*sharing*), to publication alongside other research outputs (*release*), and eventually leading to reuse that may contribute to other projects. We present this process as a cycle to emphasise the continuous improvement of code and the incremental nature of building on existing work (Fig. 1). Code review is valuable at any and every stage of development and can serve as a formal checkpoint before code progresses to the next phase. Ideally, it addresses all seven checklist dimensions, each targeting a key aspect of code quality and reusability. In practice, however, review priorities will shift depending on the development phase, the context of the review, the reviewer's expertise, and the code's intended use. A flexible approach—focusing on the most relevant checklist dimensions—ensures maximum impact at each stage

In the 'create' phase, code is planned, designed, and written, usually by a single researcher or a small team. This phase may consist of several iterations as different approaches are explored to prepare the data for analyses, or visualise outputs. At this stage, authors involved in writing code may use the checklist as an *aide memoire* to review good practices and to help ensure that the code works as expected (Running) and contains all necessary information and functionality for its intended purpose (Reporting). Documentation is key, even if the code does not work as

expected and even if the code is not yet intended for sharing; stating the purpose of code and any known issues is good practice and provides valuable context during future code development.

The 'share' phase involves distributing code to others, typically collaborators or lab members. When conducting code review at this stage, it is crucial to communicate the purpose of the code and the context or focus of the review, as this will shape the focus of the review. Code shared within a community context, with lab members or collaborators, may prioritise consistent naming conventions that adhere to community standards and practices (Readability), and focus on flexible code that can handle a range of different inputs (Reliability, Robustness) to support collaborative use and future development with the community. In contrast, code that is shared mainly for transparency, as part of a scientific paper, should be reviewed with focus on ensuring it aligns with the methods described in the manuscript (Reporting).

In the 'publish' phase, code becomes available to a wide group of users. This may include publishing code associated with a scientific paper to an online repository, or the release of a package to a library. During this phase, the focus of code review should be on ensuring that the purpose and intended functionality of the code are clearly documented for potential users (Reporting), and that others can legally use the code, and appropriately cite and credit the source and its developers (Release).

Code development and review should not end when code is published, but often does as a result of the short-term research grants that teams rely on (Coelho 2024). Yet, published code requires ongoing maintenance to ensure that it continues to achieve its goals as intended despite changes to its software dependencies. Whether building on existing code to implement new features or accommodating to new versions of dependencies, revisiting the principles and priorities applied in the initial iteration of the development cycle can support the long-term usability and sustainability of this crucial part of the research output.

Conclusion

Sharing and publishing code is a key step towards research transparency—but to maximise its impact, shared code must also be reusable. We present a checklist designed to support this goal by improving code quality across key domains of reusability.

Code review can take place at many points throughout the development cycle, with its focus shaped by context, i.e., whether the review is conducted by the original author or peers, and whether it is reviewed before sharing it with close collaborators or when finalising code for publication. We encourage researchers to embrace the flexibility of this approach and engage in code review both as developers and as reviewers. Code review is not merely about evaluating and improving code—it is a collaborative and rewarding practice that fosters learning and contributes to the transparency and reproducibility in research, facilitating long-term accessibility of research outputs.

Beyond the Checklist: Additional Considerations

Version-controlled workflows

Version control systems manage and track changes to files and are considered best practice in research—from data management to developing analysis code to writing outputs. Git and its web interface GitHub are commonly used tools for creating annotated, version-controlled workflows (Perkel 2016). Braga et al. (2023) provide an entry-level overview of how GitHub features can be used in ecology and evolution research, from tracking of code development to collaborative and asynchronous editing, and merging changes into the main project. A next step builds on the principle of continuous integration (CI), a standard process in professional software, which automates quality control and version-controlled code integration; GitHub Actions is GitHub's built-in implementation of CI.

Tools for automated code review

While our guide focuses on manual code review, automated tools can streamline the process by efficiently detecting common errors and enforcing a predefined style. For example, the R package *lintr* (Hester et al. 2025) checks style consistency, and the package *testthat* (Wickham 2011) provides unit tests for technical functionality. Automated review can be integrated into CI pipelines. By automating error and style checks, developers and reviewers can focus on more complex and nuanced aspects of their code.

Choosing a software licence

To select an appropriate licence, code creators can refer to information and comparisons provided on choosealicense.com, an open-source project maintained by GitHub. Common research licences include the permissive Massachusetts Institute of Technology (MIT) and Apache License, which are easy to understand and allow use, modification, and redistribution with minimal restrictions. These licences are compatible with others, allowing code to be combined with projects under a different licence, including those that might put the code behind a paywall. In contrast, restrictive copy-left licences, such as the GNU General Public License (GPL), require that any derivative works that use or modify the original code are also adopt the same licence term. This protection builds trust within the scientific community by limiting concerns about lack of recognition for code developers, and ensuring that the code remains open and accessible for future research and development.

Reviewer crediting

Peer review is essential for validating research methods and outputs, including code. Due to the fundamental role of code in data analysis, code review is critical to research integrity. Acknowledging reviewers, either by name or anonymously, in the code's documentation or connected publications gives credit to their valuable contributions and highlights the collaborative nature of research.

References

Abdill, R. J., Talarico, E., & Grieneisen, L. (2024). A how-to guide for code sharing in biology. *PLoS Biology*, 22(9), e3002815. https://doi.org/10.1371/journal.pbio.3002815

Atkins, A., Allen, T., Ushey, K., McPherson, J., Cheng, J., & Allaire, J. (2025). *packrat: A dependency management system for projects and their R package dependencies*. R package version 0.9.2.9000. Retrieved from https://github.com/rstudio/packrat

Barker, M., Chue Hong, N. P., Katz, D. S., Lamprecht, A. L., Martinez-Ortiz, C., Psomopoulos, F., … & Honeyman, T. (2022). Introducing the FAIR principles for research software. *Scientific Data, 9*(1), 622. https://doi.org/10.1038/s41597-022-01710-x

Barnes, N. (2010). Publish your computer code: It is good enough. *Nature, 467*(7317), 753-753. https://doi.org/10.1038/467753a

Bledsoe, E. K., Burant, J. B., Higino, G. T., Roche, D. G., Binning, S. A., Finlay, K., Pither, J., Pollock, L. S., Sunday, J. M., & Srivastava, D. S. (2022). Data rescue: saving environmental data from extinction. Proceedings of the Royal Society B, 289(1979), 20220938. https://doi.org/10.1098/rspb.2022.0938

Braga, P. H. P., Hébert, K., Hudgins, E. J., Scott, E. R., Edwards, B. P. M., Sánchez Reyes, L. L., Grainger, M. J., Foroughirad, V., Hillemann, F., Binley, A., Brookson, C., Gaynor, K., Sabet, S. S., Güncan, A., Weierbach, H., Gomes, D. G. E., & Crystal-Ornelas R. (2023). Not just for programmers: How GitHub can accelerate collaborative and reproducible research in ecology and evolution. *Methods in Ecology and Evolution, 14*(6), 1364–1380. https://doi.org/10.1111/2041-210X.14108

Coelho, L.P. (2024). For long-term sustainable software in bioinformatics. *PLoS Computational Biology*, 20(3): e1011920. https://doi.org/10.1371/journal.pcbi.1011920

Cooper, N., & Hsing, P. (2017). *A guide to reproducible code in ecology and evolution.* British Ecological Society. Retrieved from https://www.britishecologicalsociety.org/publications

Culina, A., Adriaensen, F., Bailey, L. D., Burgess, M. D., Charmantier, A., Cole, E. F., … & Visser, M. E. (2021). Connecting the data landscape of long-term ecological studies: The SPI-Birds data hub. *Journal of Animal Ecology, 90*(9), 2147-2160. https://doi.org/10.1111/1365-2656.13388

Culina, A., Ivimey-Cook, E. R., Pick, J. L., Bairos-Novak, K. R., Gould, E., Grainger, M., Windecker, S. M., & others. (2020). Low availability of code in ecology: A call for urgent action. *PLoS Biology*. https://doi.org/10.1371/journal.pbio.3000763

De Moor, D., Skelton, M., MacaqueNet, Amici, F., Arlet, M. E., Balasubramaniam, K. N., … & Brent, L. J. (2025). MacaqueNet: Advancing comparative behavioural research through large-scale collaboration. *Journal of Animal Ecology.* https://doi.org/10.1111/1365-2656.14223

Drozdz, J. A., & Ladomery, M. R. (2024). The peer review process: Past, present, and future. *British Journal of Biomedical Science, 81*, 12054. https://doi.org/10.3389/bjbs.2024.12054

Fagan, M. E. (1976). Design and code inspections to reduce errors in program development. *IBM Systems Journal, 15*(3), 182–211. https://doi.org/10.1147/sj.153.0182

Filazzola, A., & Lortie, C. (2022). A call for clean code to effectively communicate science. *Methods in Ecology and Evolution, 13*, 2119–2128. https://doi.org/10.1111/2041-210X.13961

Gomes, D. G. (2025). How will we prepare for an uncertain future? The value of open data and code for unborn generations facing climate change. *Proceedings of the Royal Society B, 292*(2040), 20241515. https://doi.org/10.1098/rspb.2024.1515

Gomes, et al. (2022). Why don't we share data and code? Perceived barriers and benefits to public archiving practices. *Proceedings of the Royal Society B, 289*, 20221113. https://doi.org/10.1098/rspb.2022.1113

Hester J, Angly F, Hyde R, Chirico M, Ren K, Rosenstock A, Patil I (2025). lintr: A 'Linter' for R Code. R package version 3.2.0, https://github.com/r-lib/lintr, https://lintr.r-lib.org.

Ivimey-Cook, E. R. , Pick, J.L., Bairos-Novak, K. R., Culina, A., Gould, E., Grainger, M., Marshall, B. M., Moreau, D., Paquet, M., Royauté, R., Sánchez-Tójar, A., Silva, I., Windecker, S. M. (2023). Implementing code review in the scientific workflow: Insights from ecology and evolutionary biology. *Journal of Evolutionary Biology, 36*(10), 1347–1356. https://doi.org/10.1111/jeb.14230

Ivimey-Cook, E. R., Sánchez-Tójar, A., Berberi, I., Culina, A., Roche, D. G., Almeida, R. A., ... & Moran, N. P. (2025). From Policy to Practice: Progress towards Data-and Code-Sharing in Ecology and Evolution. Preprint, *EcoEvoRxiv*. https://doi.org/10.32942/X21S7H

Jenkins, G. B., Beckerman, A. P., Bellard, C., Benítez-López, A., Ellison, A. M., Foote, C. G., … & Peres-Neto, P. R. (2023). Reproducibility in ecology and evolution: Minimum standards for data and code. *Ecology and Evolution, 13*, e9961. https://doi.org/10.1002/ece3.9961

Kellner, K. F., Doser, J. W., & Belant, J. L. (2025). Functional R code is rare in species distribution and abundance papers. *Ecology, 106*(1), e4475. https://doi.org/10.1002/ecy.4475

Lee, C. S., & Hicks, C. M. (2024). Understanding and effectively mitigating code review anxiety. Empirical Software Engineering, 29(6), 161. https://doi.org/10.1007/s10664-024-10550-9

Maitner, B. S., Fitzpatrick, M. C., & Alvarado, A. S. (2023). Code sharing increases citations, but remains uncommon. Preprint. *Research Square*. https://doi.org/10.21203/rs.3.rs-3222221/v1

Müller, K. (2020). *here: A simpler way to find your files.* R package version 1.0.1. Retrieved from https://CRAN.R-project.org/package=here

O'Dea, R. E., Parker, T. H., Chee, Y. E., Culina, A., Drobniak, S. M., Duncan, D. H., Fidler, F., Gould, E., Ihle, M., Kelly, C. D., Lagisz, M., Roche, D. G., Sánchez-Tójar, A., Wilkinson, D. P., Wintle, B. C., & Nakagawa, S. (2021). Towards open, reliable, and transparent ecology and evolutionary biology. *BMC Biology*, *19*(1), 68.

Perkel, J. M. (2016). Democratic databases: Science on GitHub. *Nature, 538*(7623), 127–128. https://doi.org/10.1038/538127a

Rokem, A. (2024). Ten simple rules for scientific code review. *PLOS Computational Biology, 20*(9), e1012375. https://doi.org/10.1371/journal.pcbi.1012375

Sánchez-Tójar, A., Bezine, A., Purgar, M., & Culina, A. (2025). Code-sharing policies are associated with increased reproducibility potential of ecological findings. Preprint. *EcoEvoRxiv*, https://doi.org/10.32942/X21S7H

Sandve, G. K., Nekrutenko, A., Taylor, J., & Hovig, E. (2013). Ten simple rules for reproducible computational research. *PLoS Computational Biology*, 9(10), e1003285. https://doi.org/10.1371/journal.pcbi.1003285

Walters, W. P. (2020). Code sharing in the open science era. *Journal of Chemical Information and Modeling*, 60(10), 4417-4420. https://doi.org/10.1021/acs.jcim.0c01000

Wickham H (2011). "testthat: Get Started with Testing." The R Journal, 3, 5–10. ttps://journal.r-project.org/archive/2011-1/RJournal_2011-1_Wickham.pdf.

Wickham, H. & Henry, L (2023). *purrr: Functional programming tools*. R package version 1.0.4. Available at: https://CRAN.R-project.org/package=purrr

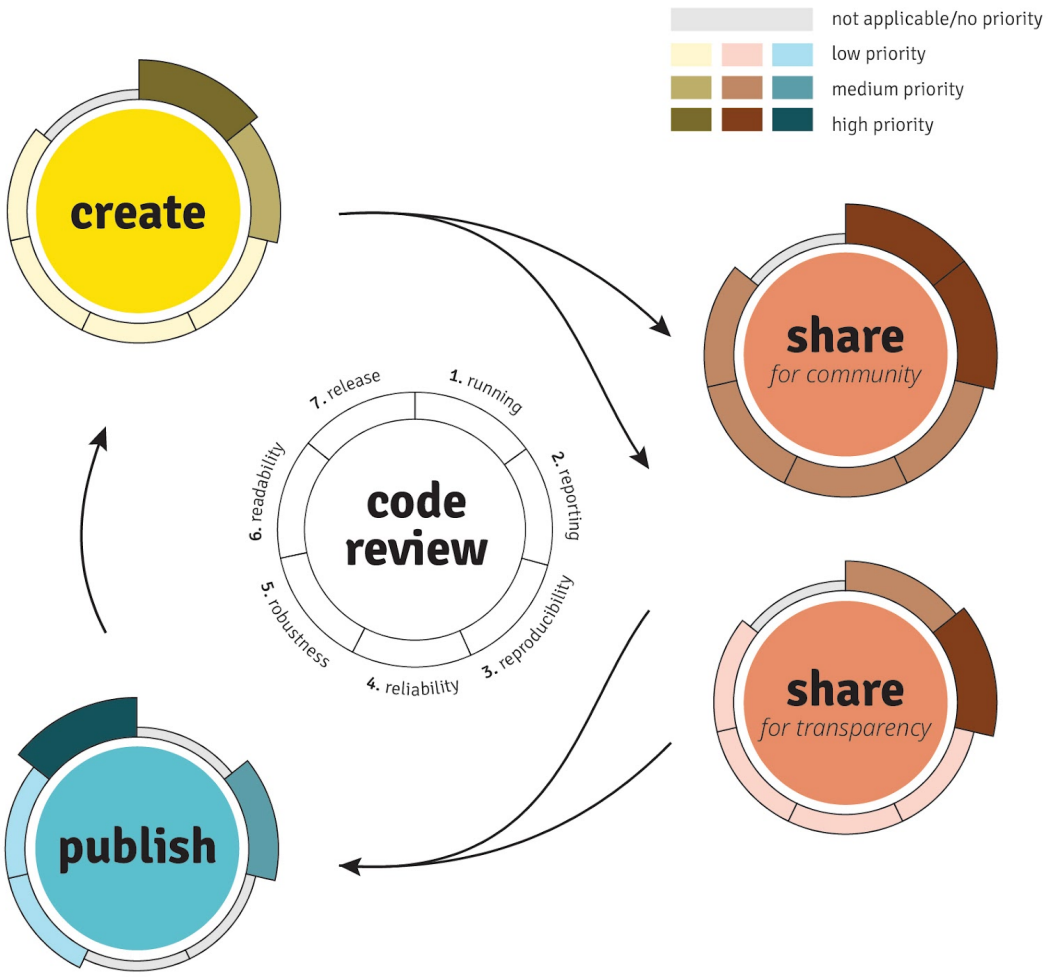

Figure 1. Review of scientific code can occur at different points throughout the code development cycle, with focus varying based on the code's purpose and review context. Reviewing code during initial development will prioritise different domains compared to reviews of code shared within a smaller research community or lab, or reviewing code before publication. Colours indicate different phases in the code development cycle (i.e., create, share, publish). The rings with seven wedges correspond to the seven domains of the code review checklist. Shading and wedge size indicate priority (grey: no priority, light: low priority, dark: high priority).

Code review in practice: A checklist for computational reproducibility and collaborative research in ecology and evolution

This checklist guides code review, whether as self-assessment or peer review, across key dimensions of reusability: Reporting, Running, Reproducibility, Reliability, Robustness, Readability, and Release. Criteria may be marked as YES (met), NO (not met), UNSURE (unclear or not evaluated), or N/A (not applicable). Designed as a flexible template, it can be tailored to different contexts by modifying, omitting, or adding criteria. This checklist is licensed under a CC BY-NC 4.0 International License, permitting sharing and adaptation for non-commercial use with attribution. Editable versions (.md, .xlsx) are available in the supplementary materials of the accompanying paper. Please cite the paper for attribution: **ADD PAPER and DOI**

REVIEW METADATA AND REVIEWER ACKNOWLEDGEMENT	GENERAL NOTES
Review of: *Code identifier, incl. version if applicable*	*Use this space for any general remarks that do not fit into specific checklist items.*
Date review completed: *Month and year*	
Operating system used: *Reviewer OS and software version*	
Review by: *Reviewer name*	
☐ I agree to be acknowledged as a code reviewer by name.	
☐ I prefer to stay anonymous in the acknowledgements.	

QUESTIONS TO GUIDE CODE ASSESSMENT	YES	NO	UNSURE	N/A	COMMENT
Reporting — Check that it does what it claims. Code should match the reported methods. Data transformations and analyses should align with the description—missing or altered steps mean the code is not as reported.					
Methods Alignment: Does the code implement the methods as described in the associated documentation or research outputs?	☐	☐	☐	☐	*Please clarify decisions or suggest improvements.*
Documentation: Is there sufficient metadata (e.g., in a README file or code header) to understand and use the code independently of external documentation?	☐	☐	☐	☐	
Running — Check that it works. Code should execute on a local machine and run its entirety, even for users with limited coding expertise.					
Functioning: Does the code run without errors from start to finish?	☐	☐	☐	☐	
Dependencies: Does the code specify all required libraries/packages or is set to install automatically?	☐	☐	☐	☐	
Cross-Platform Compatibility: Does the code run on a different operating system than the one it was developed on?	☐	☐	☐	☐	
Run Time: Does the code provide information on run time to manage user expectations?	☐	☐	☐	☐	
Complete Check: Did you run the entire code?	☐	☐	☐	☐	
Reproducibility — Check that it gives consistent results. Code should produce the same results when run with the same input—ideally, results that match expected or claimed outputs					
Numerical Reproducibility: Does the code generate the same functional outputs, i.e., statistical or simulation results, when provided with identical data and parameters?	☐	☐	☐	☐	
Visual Reproducibility: Does the code generate consistent visual outputs (e.g., figures, maps) across repeated executions with the same input?	☐	☐	☐	☐	
Requirements: Does the code include or clearly specify all necessary data, or provide mock data where applicable, to enable independent reproduction?	☐	☐	☐	☐	
Compartmentalisation: Does the code ensure the workflow is self-contained, with all external software dependencies documented and accessible for execution in other environments?	☐	☐	☐	☐	
Reliability — Check that it is built to minimise potential errors. Code should perform as intended under typical use cases, producing expected results and including internal checks for common issues to catch errors early.					
Expected Results: Does the code produce the correct type of output for each step, e.g., correct data transformations or statistical results?	☐	☐	☐	☐	
Validation & Internal Checks: Does the code include safeguards such as assertions, unit tests, or manual checks to verify that key steps are performed as intended?	☐	☐	☐	☐	
Warning & Error Handling: Does the code anticipate limitations related to data quality or input constraints and provide comments, warnings, or error messages?	☐	☐	☐	☐	
Robustness — Check that it handles the unexpected. Code should handle invalid inputs gracefully and fail safely, providing meaningful feedback. It should avoid brittle design and support flexible workflows.					
Feedback: Does the code provide clear, interpretable comments/messages on potential issues?	☐	☐	☐	☐	
Parameterisation: Does the code avoid hard-coding? For instance, does it use relative file paths instead of absolute ones?	☐	☐	☐	☐	
Efficiency: Does the code efficiently avoid redundancy and include only what is necessary?	☐	☐	☐	☐	
Functional Programming Principles: Does the code minimise global state changes using functions and pipelines (e.g., R tidyverse packages)?	☐	☐	☐	☐	
Readability — Check that it is clear and clean. Code should be easy to follow, well-structured and logically organised like a manual, and naming of variables and functions should be easy to understand.					
Organisation: Does the code follow a logical order, guiding users through the workflow and clearly conveying its function?	☐	☐	☐	☐	
Modularity: Does the code consist of manageable, logical sections (e.g., functions, sections, modular scripts) that together form a coherent workflow?	☐	☐	☐	☐	
Naming Conventions: Does the code use informative names for variables, functions, and objects?	☐	☐	☐	☐	
Style Conventions: Does the code consistently follow a style guide, such as tidyverse style for R?	☐	☐	☐	☐	
Release — Check that it is ready for sharing and reuse. Code should be prepared for sharing, include licensing, citation information, and relevant metadata to support reuse and attribution.					
Contact: Do the authors or maintainers provide guidance on how to report feedback or obtain support?	☐	☐	☐	☐	
Legal Permissions: Does the code include a licence specifying how it can be used, modified, shared?	☐	☐	☐	☐	
Attribution: Does the code have a Persistent Identifier (e.g., Digital Object Identifier DOI), making it easy to cite and give proper credit in academic and research contexts?	☐	☐	☐	☐	

Code review in practice: A checklist for computational reproducibility and collaborative research in ecology and evolution

This checklist guides code review, whether as self-assessment or peer review, across key dimensions of reusability: Reporting, Running, Reproducibility, Reli
Release. Criteria may be marked as YES (met), NO (not met), UNSURE (unclear or not evaluated), or N/A (not applicable). Designed as a flexible template, it c
modifying, omitting, or adding criteria. This checklist is licensed under a [CC BY-NC 4.0](https://creativecommons.org/licenses/by-nc/4.0/) International Licer
non-commercial use with attribution. Editable versions (.md, .xlsx) are available in the supplementary materials of the accompanying paper. Please cite the pa
DOI

REVIEW METADATA
Review of: _Code identifies/name/author_ <!-- some code identifier, incl. version if applicable -->
Date review completed: _mm YY_ <!-- month and year is fine -->
Operating system and software version used: _..._ <!-- reviewer OS -->

REVIEWER ACKNOWLEDGEMENT
Review by: _Code reviewer name_ <!-- add name and tick as applicable -->
[ ] I agree to be acknowledged as a code reviewer by name.
[ ] I prefer to stay anonymous in the acknowledgements.

GENERAL NOTES
optional <!-- Use this space for any general remarks that do not fit into specific checklist items. -->

QUESTIONS TO GUIDE CODE ASSESSMENT

Reporting — Check that it does what it claims.
Code should match the reported methods. Data transformations and analyses should align with the description—missing or altered steps mean the code is r

- **Methods Alignment:** Does the code implement the methods as described in the associated documentation or research outputs?
  [ ] YES
  [ ] NO
  [ ] UNSURE
  [ ] N/A
  Comment: <!-- Enter any clarifications or recommendations here -->

- **Documentation:** Is there sufficient metadata (e.g., in a README file or code header) to understand and use the code independently of external documen
  [ ] YES
  [ ] NO
  [ ] UNSURE
  [ ] N/A
  Comment: <!-- Enter any clarifications or recommendations here -->

Running — Check that it works.
Code should execute on a local machine and run its entirety, even for users with limited coding expertise.

- **Functioning:** Does the code run without errors from start to finish?
  [ ] YES
  [ ] NO
  [ ] UNSURE
  [ ] N/A
  Comment: <!-- Enter any clarifications or recommendations here -->

- **Dependencies:** Does the code specify all required libraries/packages or is set to install automatically?
  [ ] YES
  [ ] NO
  [ ] UNSURE
  [ ] N/A
  Comment: <!-- Enter any clarifications or recommendations here -->

- **Cross-Platform Compatibility:** Does the code run on a different operating system than the one it was developed on?
  [ ] YES
  [ ] NO
  [ ] UNSURE
  [ ] N/A
  Comment: <!-- Enter any clarifications or recommendations here -->

- **Run Time:** Does the code provide information on run time to manage user expectations?
  [ ] YES
  [ ] NO
  [ ] UNSURE
  [ ] N/A
  Comment: <!-- Enter any clarifications or recommendations here -->

- **Complete Check:** Did you run the entire code?
  [ ] YES
  [ ] NO
  [ ] UNSURE
  [ ] N/A
  Comment: <!-- Enter any clarifications or recommendations here -->

Reproducibility — Check that it gives consistent results.
Code should produce the same results when run with the same input—ideally, results that match expected or claimed outputs.

- **Numerical Reproducibility:** Does the code generate the same functional outputs, i.e., statistical or simulation results, when provided with identical data a
  [ ] YES
  [ ] NO
  [ ] UNSURE
  [ ] N/A
  Comment: <!-- Enter any clarifications or recommendations here -->

- **Visual Reproducibility:** Does the code generate consistent visual outputs (e.g., figures, maps) across repeated executions with the same input?
  [ ] YES
  [ ] NO
  [ ] UNSURE
  [ ] N/A
  Comment: <!-- Enter any clarifications or recommendations here -->

- **Requirements:** Does the code include or clearly specify all necessary data, or provide mock data where applicable, to enable independent reproduction?
  [ ] YES
  [ ] NO
  [ ] UNSURE
  [ ] N/A
  Comment: <!-- Enter any clarifications or recommendations here -->

- **Compartmentalisation:** Does the code ensure the workflow is self-contained, with all external software dependencies documented and accessible for ex
  [ ] YES
  [ ] NO
  [ ] UNSURE
  [ ] N/A
  Comment: <!-- Enter any clarifications or recommendations here -->

Reliability — Check that it is built to minimise potential errors.
Code should perform as intended under typical use cases, producing expected results and including internal checks for common issues to catch errors early

- **Expected Results:** Does the code produce the correct type of output for each step, e.g., correct data transformations or statistical results?
  [ ] YES
  [ ] NO
  [ ] UNSURE
  [ ] N/A
  Comment: <!-- Enter any clarifications or recommendations here -->

- **Validation & Internal Checks:** Does the code include safeguards such as assertions, unit tests, or manual checks to verify that key steps are performed a
  [ ] YES
  [ ] NO
  [ ] UNSURE
  [ ] N/A
  Comment: <!-- Enter any clarifications or recommendations here -->

- **Warning & Error Handling:** Does the code anticipate limitations related to data quality or input constraints and provide comments, warnings, or error mes
  [ ] YES
  [ ] NO
  [ ] UNSURE
  [ ] N/A
  Comment: <!-- Enter any clarifications or recommendations here -->

Robustness — Check that it handles the unexpected.
Code should handle invalid inputs gracefully and fail safely, providing meaningful feedback. It should avoid brittle design and support flexible workflows.

- **Feedback:** Does the code provide clear, interpretable comments/messages on potential issues?
  [ ] YES
  [ ] NO
  [ ] UNSURE
  [ ] N/A
  Comment: <!-- Enter any clarifications or recommendations here -->

- **Parameterisation:** Does the code avoid hard-coding? For instance, does it use relative file paths instead of absolute ones?
  [ ] YES
  [ ] NO
  [ ] UNSURE
  [ ] N/A
  Comment: <!-- Enter any clarifications or recommendations here -->

- **Efficiency:** Does the code efficiently avoid redundancy and include only what is necessary?
  [ ] YES
  [ ] NO
  [ ] UNSURE
  [ ] N/A
  Comment: <!-- Enter any clarifications or recommendations here -->

- **Functional Programming Principles:** Does the code minimise global state changes using functions and pipelines (e.g., R tidyverse packages)?
  [ ] YES
  [ ] NO
  [ ] UNSURE
  [ ] N/A
  Comment: <!-- Enter any clarifications or recommendations here -->

Readability — Check that it is clear and clean.
Code should be easy to follow, well-structured and logically organised like a manual, and naming of variables and functions should be easy to understand. <

- **Organisation:** Does the code follow a logical order, guiding users through the workflow and clearly conveying its function?
  [ ] YES
  [ ] NO
  [ ] UNSURE
  [ ] N/A
  Comment: <!-- Enter any clarifications or recommendations here -->

- **Modularity:** Does the code consist of manageable, logical sections (e.g., functions, sections, modular scripts) that together form a coherent workflow? <
  [ ] YES
  [ ] NO
  [ ] UNSURE
  [ ] N/A
  Comment: <!-- Enter any clarifications or recommendations here -->

- **Naming Conventions:** Does the code use informative names for variables, functions, and objects?
  [ ] YES
  [ ] NO
  [ ] UNSURE
  [ ] N/A
  Comment: <!-- Enter any clarifications or recommendations here -->

- **Style Conventions:** Does the code consistently follow a style guide, such as tidyverse style for R?
  [ ] YES
  [ ] NO
  [ ] UNSURE
  [ ] N/A
  Comment: <!-- Enter any clarifications or recommendations here -->

Release — Check that it is ready for sharing and reuse.
Code should be prepared for sharing, include licensing, citation information, and relevant metadata to support reuse and attribution.

- **Contact:** Do the authors or maintainers provide guidance on how to report feedback or obtain support?
  [ ] YES
  [ ] NO
  [ ] UNSURE
  [ ] N/A
  Comment: <!-- Enter any clarifications or recommendations here -->

- **Legal Permissions:** Does the code include a licence specifying how it can be used, modified, shared?
  [ ] YES
  [ ] NO
  [ ] UNSURE
  [ ] N/A
  Comment: <!-- Enter any clarifications or recommendations here -->

- **Attribution:** Does the code have a Persistent Identifier (e.g., Digital Object Identifier DOI), making it easy to cite and give proper credit in academic and re
  [ ] YES
  [ ] NO
  [ ] UNSURE
  [ ] N/A
  Comment: <!-- Enter any clarifications or recommendations here -->

<!-- end of review -->