Reply to Reviewer 2: Facundo Fabián Sapienza

Dear Reviewer,

We appreciate your prompt and critical review of our paper. Your thoughtful comments and suggestions have greatly improved the quality of our revised manuscript.

In the following sections, we have addressed your comments point-by-point and changed the manuscript accordingly. Please find the tracked changes attached to this letter. Your suggestions are highlighted in blue, while our responses are in black and new text italics. All section and line numbers mentioned correspond to the revised manuscript. In summary, we have revised aspects of the introduction and enhanced the results and discussion sections.

The manuscript contributes to the scientific literature by quantifying the level of software sustainability of different climate change impact models. This is done by first defining a series of nine indicators that evaluate factors such as automation, documentation, testing, and good coding practices present in the model.

I recommend the publication of the manuscript in GMD after the revisions and comments here presented are addressed. I believe this work brings light into the robustness, reproducibility and overall health of the software that we scientist develop. I personally enjoyed the reading of the manuscript, and I would be glad of seeing this work published. However, before publication I strongly suggest the authors to review and address the following major and minor points. If well overall the manuscript is easy to read and follow, some parts are a bit vague and required more explanations and/or more bibliographical references.

We appreciate the positive feedback on our work and have addressed the raised points in the following.

All comments, both major and minors, are aimed to improve the quality of the paper or to dilucidated some of my doubts or questions as I was reading the manuscript.

Major Comments

- It will help to the discussion the early introduction of one or more examples of software that follows the guidelines of “sustainable research software”. I suggest using standard examples for this, for example a major library in Python (numpy, pandas, sklearn) with a link to their respective source code would help the reader to have an idea of how good software looks like.

  Thank you for your valuable feedback. We have revised our introduction to include your suggestion.

  (Section 1, Line 68-74) *“There are various interpretations of the meaning of “sustainable research software”. Anzt et al. (2021) define research software as software that is maintainable, extensible, flexible (adapts to user requirements), has a defined software architecture, is testable, has comprehensive in-code and external documentation, and is accessible (the software is licensed as Open Source with a digital object identifier (DOI) for proper attribution) (Anzt et al., 2021). For example, NumPy (https://numpy.org/) is a widely used scientific software package that exemplifies many of these qualities (Harris et al., 2020). Although NumPy is not an impact model, it is an exemplar of sustainable research software; it*

is open-source, maintains rigorous version control and testing practices, and is extensively documented, making it highly reusable and extensible for the scientific community."

- [146] **Version control**. My understanding is that in the manuscript the authors assess whether the models use or use not git or a similar version control system. However, I think it is also important to evaluate how version control has been used during the development of the different versions of the model. Did the contributors follow good version control practices (modular commits, push requests, discussions, versioning, etc), or simply put the final version of the software in GitHub or similar? I understand this is a finer analysis which I don't think is necessary to perform in the manuscript, but I would suggest mentioning this in the Version Control section since it is an important point.

  Thank you for your thoughtful comment. We agree that evaluating the implementation of version control practices, such as modular commits, pull requests, discussions, and versioning would provide valuable insights and would be interesting for follow-up studies. In line with your suggestion, we have mentioned these practices in the revised section. The updated version now reads:

  (Section 2, Line 153-161) *"Version control. Version control systems such as Git and Mercurial facilitate track changes, and collaborative development, and provide a history of software evolution. To assess whether GIMs use version control for development, we focused on commonly used open-source version control hosting repositories such as GitLab, GitHub, BitBucket, Google Code, and Source Forge. The hostname such as "github" or "gitlab" in the source link of models provides clear indications of version control adoption in their development process. For other models, we searched within the Google search engine using model names and keywords such as "Bitbucket", "Google Code", and "Source Forge". While we focus on identifying the use of version control systems, evaluating how version control was implemented during the development process — such as the use of modular commits, pull requests, discussions, and proper versioning — is a finer analysis that falls beyond the scope of this study. However, such practices are crucial for ensuring high-quality software development and collaborative practices."*

- [154] **Use of open source license**. I think this section requires more discussion or at least a few references supporting the statement "Open-source licenses foster collaboration and transparency by enabling community contributions and ensuring that software remains freely accessible". I partially agree with this statement, but I would like to see the why of this. Furthermore, I think is also important to mention the difference between different types of major licenses (copyleft or permissive), since I think this also has an impact in the meaning of the sentence. This point pops up every time the topic of licenses is addressed in the paper, so I strongly suggest the authors to address what do they mean by open-source licenses and how they relate to copyleft/permissive licenses.

  Thank you for your insightful feedback. We have removed the statement regarding open-source licenses fostering collaboration and transparency, as we agree that it requires more discussion and supporting references. Additionally, we have expanded the section to include a detailed explanation of the key differences between copyleft and permissive licenses, the two major categories of open-source licenses. In our study, we focus solely on the presence of an open-source license regardless of the type of open source license. The revised section now reads:

(Section 2, Line 164-172) *"Use of an open-source license. We determined the existence of open-source licenses by checking license files within repositories or official websites against licenses approved by the Open Source Initiative (OSI) (https://opensource.org/licenses). Specifically, we looked for licenses that conform to the Open Source Definition, which ensures that software can be freely used, modified, and shared (Colazo and Fang, 2009; Rashid et al., 2019). There are two major categories of open-source licenses: permissive licenses, such as MIT or Apache, that allow for minimal restrictions on how the software can be used (e.g., providing attribution), and copyleft licenses, like GPL, that require derivatives to maintain the same licensing terms (Colazo and Fang, 2009; Rashid et al., 2019). Although these licenses differ in their terms, both contribute to collaboration and transparency. In this study, we only check if the software is open-source, regardless of the type of open-source license."*

- [159] **Number of active developers**. I think the number of active developers is a good proxy for evaluating how robust is the development of the software, but I don't think is the only one. I think with the same philosophy one can evaluate the number of commits, push requests, open and closed issues, etc. I suggest mentioning this in the manuscript, and maybe changing the "number of active developments" tag to something more generic that includes these other proxies or a more general concept (e.g., "software robustness"). Furthermore, it may be important to emphasize the distinction between developers and contributors: one can contribute to a project without writing source code, but for example opening issues, managing a project, writing the documentation, etc. In this sense, contributors help to improve the robustness of the software, even when they don't directly write any line of code.
  We agree that the number of active developers is not the only indicator if the goal is to measure the project's robustness, and we appreciate your suggestion to consider additional factors such as commits, pull requests, and open/closed issues. Our measure of active developers serves as an indicator of ongoing maintenance and the prevention of software stagnation, rather than a definitive measure of a project's robustness (see Section 2, Line 174). Therefore, we prefer to keep the term "number of active developers". Also, to avoid confusion between the terms "contributor" and "developer," we use "developer" exclusively, as contributors also write lines of code according to the definition from the Mozilla Public License (https://www.mozilla.org/en-US/MPL/2.0/).

- [166] **Containerization**. I will suggest here making also a reference to cloud supported containers, such as Binder or GoogleColab, that allow the re-execution of the software. Under the hood, this also work as a container, but I think the deeper concept here is the capacity of re-executing an model with the computational environment requirements. This further resonates with the concept of analysis-ready data, cloud-optimized formats (see "Cloud-Native Repositories for Big Scientific Data" by Abernathey et. al, 2021) in the case of datasets. I think the same ideas apply here for models.
  Thank you for your thoughtful comment. We have incorporated your suggestions into the section and it now reads.

  (Section 2, Line 185-189) *"Some popular containerization solutions include Docker (https://www.docker.com/) and Apptainer (https://apptainer.org/). There are also cloud-supported container solutions such as Binder (https://mybinder.org/) with the capacity to execute a model with the computational environment requirements analogous to the concept of analysis-ready data and cloud-optimized formats for datasets (Abernathey et al., 2021)."*

- [178] **Public availability of an (automated) testing suite**. I really like this point. I this is a good idea to look for automation of the tests. However, I would like to point out here that the concept of "automation" is in principle independent and complementary to the existence of the testing suite. Furthermore, the concept of automation applies also to some other indicators. For example, one can automate the creation of documentation using GitHub Pages, using GitHub actions to ensure the containerization of the software (even create all the ingredients for a docker file), and even ensure the compliance with coding standards of the software (see for example the Julia code formatter action: https://github.com/julia-actions/julia-format and I imagine there must be a way to automate the use of Pylint with Python).

  Thank you for your feedback. We agree that the concept of automation is both independent and complementary to the existence of a testing suite. Since this section only accesses the availability of an (automated) testing suite, we have expanded our recommendations to include automation not only for testing but also for other aspects such as documentation creation, containerization, and coding standards compliance.

  (Section 5, Line 567-571) "*Integrate automation in development practices. Automation plays a key role in streamlining software development by reducing manual effort and ensuring consistency (Wijendra and Hewagamage, 2021). We encourage developers to integrate automation into their workflows to improve efficiency. For instance, developers can use GitHub Actions to automate various tasks like running test suites, generating documentation, ensuring adherence to coding standards, and managing dependencies.*"

- [269] How well documented are the models that just have a README file? In my experience, README documentations tend to be very plain and difficult to navigate. I think it is important to mention something about the quality of the documentation, at least as an observation.

  Thank you for the comment. We have incorporated your suggestions and the full text now reads:

  (Section 3, Line 288-294) *"Our analysis reveals that 75% of the GIMs (24 out of 32) have publicly accessible documentation (Table 2). We observed a range of documentation formats across these GIMs. Specifically, 6 GIMs provided readme files, 13 had dedicated webpages for documentation, and 5 included comprehensive manuals (see supplementary file ISIMIP_models.xlsx). While README files tend to be more minimal and sometimes difficult to navigate, we observed that they generally contain essential information such as instructions on how to run the research software. The prevalence of documentation practices among most models underscores the importance of documenting research software. However, a notable portion (25%) of the studied models either lack documentation or documentation has not been made publicly available (Table 2). "*

- [295] Following my previous comment, I think it is important to state why licensing is important and the difference between licenses. References or further support here is needed. Copyleft and permissive licenses are very different. I would also suggest pointing which one of the licenses in Figure 2 are copyleft of permissive.

Thank you for your valuable comment. We refer you to our response to [154] regarding the discussion of open-source licenses, where we expanded on the importance of licensing and the key differences between copyleft and permissive licenses, including references for further support. As the focus of our study is on the general use of open-source licenses rather than the specific restrictions associated with each type, we believe that distinguishing between copyleft and permissive licenses in Figure 2 would be outside the scope of this analysis.

- [378] I don't understand the purpose of this section. The effort here is calculated using equation 1, which (besides being based in many assumptions that the authors do take care of) already suggested that more lines of code mean more effort, which is an expected concussion. With this point in mind, I don't fully understand what new conclusion are made in Section 3.3. There is a chance that here I am missing an important point, and in that case, I would like the authors to clarify. I think here it would be interesting to see how the effort correlated with the number of satisfied indicators. Without further analysis, I would suggest removing or perform a different analysis in this section.

Thank you for your insightful feedback however the primary goal of Section 3.3 is not to draw new conclusions about the relationship between lines of code and effort, but rather to provide a rough effort estimate involved in developing these complex research software tools. This estimate aims to give developers and funders a sense of the scale of effort required, encouraging developers to invest in best practices for code development once funded, and making funders aware of the necessary support.

We have removed the first line of the paragraph, which we believe is confusing. We hope this clarification addresses your concerns. The revised now reads:

(Section 3, Line 404-419) *"To provide a rough cost estimate for the software development of the 32 impact models, we use the cost estimate model from Sachan et al. (2016) (see section 2.4) in a scenario of "what if we would hire a commercial software company to develop the source code of the global impact models?" This cost estimate does not include developing the science (e.g., concepts, algorithms, and input data) nor costs of documenting, running, and maintaining the software, only the implementation of code. We assume that the COCOMO model is transferable to research software as the NASA projects used in cost model contain software that is similar to research software. As the TLOC of the impact model codes ranges from 262 to 500,000 TLOC (Fig. 7), the effort required to produce these models ranges from 1 to 495 person-months (Fig. 7). With a small additive change of ±0.1 of the COCOMO model coefficients, the range of estimated effort changes to 1 to 255 person-months in the case of -0.1 , and to 1 to 960 person-months in the case of +0.1 (Supplementary Fig. S2).*

The results suggest that these complex research software programs are expensive tools that require adequate funding for development and maintenance to make them sustainable. This is consistent with previous studies that have highlighted funding challenges for developing and maintaining sustainable research software in various domains (Carver et al., 2013, 2022; Eeuwijk et al., 2021; Merow et al., 2023; Reinecke et al., 2022). Merow et al. (2023) also emphasized that the accuracy and reproducibility of scientific results increasingly depend on updating and maintaining software. However, the incentive structure in academia for software development — and especially maintenance — is insufficient (Merow et al., 2023)."

- [410] I think this is a very important question and should be first raised in lines 197-200, and maybe postponing the discussion until later. It is not clear for me that 30-60% is the desired number, since most of the time other heuristics are used for determining the number of comments. In a nutshell, better code requires minimal commenting and clarity in the code. (See book "Beautiful Code" for a nice collection of essays around this precise point.) Furthermore, I think is important to mention the entanglement between writing good documentation and commenting, since many software documentations are generated automatically based on the comments in the code (e.g., based on the docstrings in Python and Julia).

  Thank you for your valuable feedback. Regarding minimal commenting on high-quality code, we agree that well-written code often requires fewer comments. However, it's important to note that the need for comments can vary depending on factors such as the programming language used, the complexity of the algorithms, and the expertise of the developers. Our paper specifically focuses on novice developers, particularly PhD students and postdocs in academic settings, who may not be expert programmers. In these environments, frequent turnover of personnel can result in new researchers inheriting poorly documented code, which can pose significant challenges. Therefore, while minimal commenting may be appropriate for highly experienced developers, the context of academic research and novice coders often requires more explicit comments for clarity and maintainability.

  Regarding the comment density recommendation, we acknowledge that our initial phrasing around the 30-60% comment density may have come across as too prescriptive. Our intention was to reference this range as it is commonly cited in the literature, not to imply it as a strict rule. We have revised our text to clarify this, now stating:

  (Section 2, Line 216-219) " . *Arafat et al. (2009) and He (2019) suggest that comment density between 30-60% may be optimal. For most programming languages, this range is considered to represent a compromise between providing sufficient comments for code explanation and having too many comments that may distract from the code logic (Arafat and Riehle, 2009; He, 2019).* "

  We recognize that not all programming languages natively support automatic generation of documentation from comments. Our discussion of this feature now intends to highlight its potential benefits where available, rather than suggest it as a universal solution (see Section 5, line 567 on automation).

- [490] Here it is mentioned a very important point: design principles, or design patterns. More than writing code that follows certain standards, it is important to think about the overall software architecture of the model. E.g., if I am working in Python, what are going to be my classes? How do they interact with each other? How data will be processed? Etc. None of the indicators really address this aspect of software development (which I think is fine for the scope of the manuscript), but I think the authors should emphasize this point in the recommendation section.

  Thank you for your comment. We now emphasise the points suggested. The section now reads

  (Section 5, Line 541-544) *"Design principles help adhere to the principles of sustainable research software, such as modularity, reusability and interoperability. These principles also*

guide the design of software by determining, for instance, the interaction of classes addressing aspects such as separation of concerns, abstraction, and encapsulation (Plösch et al., 2016).”

- [496] As it was already mentioned, I think is important to remark that what here is referred as internal and external documentation sometimes are the same. Documentation can be created from comments in the code, specially docstrings, and this is actually a great documentation practice, to make the internal and external documentation to be the same so there is no repetition nor contradiction between the two.

  Thank you for your feedback. While we recognize external documentation can be generated from code comments, we respectfully disagree that internal and external documentation should always be the same. Not all programming languages support this feature, and external documentation often includes additional resources such as videos, publications, tutorials (as discussed in section 5, lines 550) that go beyond what is covered in docstrings.

- Since automation is mentioned as playing an important role, I would suggest including automation in a more general sense in the recommendation section. This includes using GitHub Actions to run the test suite, automate the creation of the documentation (static or dynamic), check for code style, check packages dependencies, etc. Another interesting tool the authors may want to consider mentioning is Makefile.
  We refer you to our response to [178] regarding the discussion of automation.

- Since the paper addresses the important aspect or sustainable software, I would strongly suggest that the code used to generate the analysis, and the figures of the manuscript are presented in the same standard that the nine indicators dictate. I think this will really improve the quality of the work, and it can serve as an example in the manuscript itself of “how things should be done”. I think readers would like to see that, and I think it is part of the philosophy of the manuscript to promote these good practices. Furthermore, I would make the point that the same tools that had been used in this manuscript can be used for analyzing other source code models, so the code used in the manuscript can be re-usable by other users.
  Thank you for the suggestion. While we understand the desire to showcase our analysis code as an example, the scripts (although developed with the best practices) used for data processing and visualization differ significantly from the complex research software models the nine indicators are designed for. These simpler scripts don't require the same architectural planning or extensive documentation, nor do they fully embody indicators like testing or licensing. However, for those interested in sustainable research software, models like HydroPy (see section 3.4) are a great starting point. This can serve as a better example of how to apply the indicators discussed in the manuscript.
  We now state that in the revised manuscript (Section 3, Line 445-446)

  “The HydroPy model is great starting point for sustainable research software development as it illustrate the application of the sustainability indicators.”

Minor Comments

- [30] I am a bit confused by what Earth system modelling entitles. When presenting the models, we are talking about models in agriculture, biomes, fire, etc. In line 33, it says that “While so-called Earth System Models always include the simulation of atmospheric processes and thus

compute climate variables and how they change due to greenhouse gas emissions […]". However, this excludes many "Earth system models", including all those not based in atmospheric processes. I think the phrasing of Earth system model should be narrowed to what the models are for. For example, all the models in the ISIMIP are about the impact of climate change.

Thank you for your comment. We have now revised the line 33 for clarity which now reads:

(Section 1, Line 33-35) *"A specific class of simulation models of the Earth called impact models enables us to quantitatively estimate the potential impacts of climate change on, e.g., floods (Sauer et al., 2021), droughts (Satoh et al., 2022), and food security (Schmidhuber and Tubiello, 2007)."*

- [66-70] Repeats reference Anzt et. al. (2021)

We have revised this section for clarity, referring to the definition by the authors: Anzt et al. (2021), which now reads:

(Section 1, Line 68-71) *"Anzt et al. (2021) define research software as software that is maintainable, extensible, flexible (adapts to user requirements), has a defined software architecture, is testable, has comprehensive in-code and external documentation, and is accessible (the software is licensed as Open Source with a digital object identifier (DOI) for proper attribution) (Anzt et al., 2021)."*

- [83-99] I suggest splitting this paragraph into two, with the second stating what is done specifically in this work (maybe just break before "In this study, we assess…").

We have split the paragraph into two as suggested. The paragraph with "In this study, we assess…" snow starts from Section 1 Lines 96.

- [105] Data in this line means "model"? I will suggest not to use "data" to refer to the models in the ISIMIP database, since it is a bit confusing. If this is referring to another type of data, maybe explain.

Thank you for the comment. We have revised the section accordingly:

(Section 2, Line 111-112) "*As the focus of our analysis is on global impact models, we sorted the models by spatial domain and filtered out models operating at local and regional scales, resulting in a subset of 264 GIMs."*

- [110-111] This sentence requires rephasing, since it is ambiguous what it means by "in the described way" (meaning GitHub/GitLab or also including source code in reference papers).

We have rephrased this section to remove ambiguity. This section now reads.

(Section 2, Line 116-118) *"As of April 2024, 32 out of the 112 unique model source codes were accessible either through direct links from the ISIMIP database or via manual searches on platforms like GitHub and GitLab, as well as in code availability sections of reference papers."*

- [189] Is PEP8 the de-facto coding style in Python? I think there used to be some alternatives and may be important to mention something like this. Furthermore, in other programming languages there are more than one coding style that are accepted by the community (e.g., in

Julia), so it may be important to mention that the important aspect of this is that the developers of one model stick with one style, rather that sticking with one single style.

PEP8 is widely regarded as the standard style guide for Python (https://arxiv.org/pdf/2408.14566), although some organizations, such as Google, have their own internal versions. In Line 189, which focuses solely on the methods, we only discuss the process of accessing coding standards and, for simplicity, concentrate on PEP8 in Python, as there are tools for measuring compliance to PEP8 standard. As suggested, we now note that there can be more than one coding style that is accepted by the language community (e.g., Julia). The revised section now reads:

(Section 2, Line 202-204) *"Analysing the conformance to these standards can be complex, particularly when the source code is written in multiple languages. Different languages may have various coding styles or style guides. For instance, multiple style guides are available and accepted by the Julia community (JuliaReachDevDocs, 2024)."*

We agree that developers consistently follow one coding style for their project hence we revise or recommendation section to clearly state this.

*(*Section 5, Line 537-540) *"Use coding standards accepted by your community (e.g., PEP8 for Python), good and consistent variable names, design principles, code quality metrics, peer code review, linters and software testing: Coding standards help you write clear, consistent, and readable code that follows the best practices of your programming language and domain. It is key that developers consistently follow a coding style recognized by the relevant language community. …"*

- Table 1. Following the logic in the text, I will suggest making the division in the table between best practice in software engineering and source code quality (e.g., adding this information in the table, dividing them with a horizontal line). I think this will make the conceptual difference clearer than using the footnote. Check punctuation at the end of the descriptions. Some of the items end with dot, not dot, or comma.
  Thank you very much for the feedback. Table 1 has been revised as suggested.

- [245] These are the 32 models mentioned in line 110, right? I would mention this again for clarity.
  Thank you for your comment. We have already clarified this in the relevant section. Specifically, we mention:
  (Section 3, Lines 263-264) *"The source code of the 32 GIMs is written in 10 programming languages (Fig. 1a). Fortran and Python are the most widely used, with 11 and 10 models, respectively."*

- Figure 1. I will suggest sorting the bars in increasing/decreasing order. This is a comment I had with all the rest of the tables of the paper, where I would order the bars for clarity.
  We have revised Figures 1, 2, 3, 4, 5, 7 to sort the bars in decreasing order for clarity. Note that for Figures 3, 5 and 7, we have sorted the data by decreasing values within each sector.

- [255] I will suggest not starting the sentence with a numeral since this is uncommon and non recommended in formal English.

We have revised the sentence accordingly.

(Section 3, Line 274) *"We find that 24 (75%) of the readily accessible 32 GIMs were hosted on GitHub (Fig. 1b)."*

- [289] There is some repetition between what is said here and the paragraph in line 255 and Figure 1. I think mentioning git and the corresponding platform (GitHub, …) should be made once in the same section for clarity in the text.
  We have revised the section accordingly. This now reads:

  (Section 3, Line 309-311) *"We find that 81% (26 out of 32) of GIMs uses Git as their version control system reflecting the widespread acceptance of Git across the sectors (Table 2). In the remaining cases, information about the specific version control system used for these GIMs was unavailable."*

- Consider introducing Table 2 before Figure 1, since it contains the overall information of the model.
  Thank you for your suggestion, however, we prefer to keep the current order, with Figure 1 introduced before Table 2, as it aligns better with the overall flow and structure of the paper.

- It would be great to add an extra column to Table 2 with the year of the model (last version) and sort by this. I think it would be interesting to see if the availability of documentation, version control, etc, had improved over the years. Also interesting to see programming language used and how this changed over the years.
  Thank you for your valuable suggestion. We have added an extra column to Table 2 to include the year of the model's last version. However, we found that sorting the models by year resulted in poor readability. Therefore, we have maintained the sorting by sectors, as it aligns better with the logic of most figures.

- [291] I don't agree with the statement that "Developers' preference for Git highlights its user-friendly nature and effectiveness in supporting collaborative efforts". I think there may be other reasons for this, since there were and are other version control systems that are as user-friendly as git but didn't became that popular. One reason for this is that GitHub naturally uses git, and that many developers use VSCode with also supports git and GitHub. If the authors want to keep this sentence as it is, the statement should be supported by references.
  Thank you for your feedback. We have considered your comments and have decided to remove the statement regarding "developers' preference for Git…". (Section 3, Line 308)

- [306] It is important to start saying what is a good number of contributors, or what is expected to see here. It is unclear to me that ~10 contributors is robust enough.
  Thank you for your comment. We do not specify a required number of developers for a project to be considered robust, as this can vary significantly depending on the project's size and complexity. While a larger number of active developers may indicate a robust and well-maintained project, it is not a strict requirement for all GIMs. Smaller or less complex projects can be effectively maintained by even a single experienced developer.

Our measure of active developers serves as an indicator of ongoing maintenance and the prevention of software stagnation, rather than a definitive measure of a project's robustness. We have stated the goal of using this indicator in the method section (Section 2, Line 174)

- [320] Mention which container platform these 5 models used. Did all used Docker? If so, how do they share it?

  Thank you for the suggestion. Apart from the CLASSIC model, which uses Apptainer, the remaining four models utilize Docker as their containerization technology. The CLASSIC container is shared via Zenodo, whereas the Docker containers for the other four models are distributed through GitHub. We have updated our text to reflect these details:

  (Section 3, Line 340-344) *"Only 5 (16%) of the GIMs have implemented containerized solutions (Table 2). While the CLASSIC model uses Apptainer, the other four models use Docker as their containerization technology. The CLASSIC container is shared via Zenodo, whereas the Docker containers for the remaining models are distributed through GitHub. Despite the recognized benefits of containerization in promoting reproducible research, provisioning of the software in containers is not yet a common practice in GIM development."*

- [326] I am curious: do the models with test suite use a preferred programming language? Does the programming language plays a role in how easy is to implement the test suite? Maybe the authors want to answer to this question in the manuscript, I think it will make an interesting point.

  Thank you for the suggestion. We now answer both questions in the manuscript.

  (Section 3, Line 349-354) "*Our research indicates that 28% (9 out of 32) of the examined GIMs have a testing suite in place to test the software's functionality (Table 2). The models with test suites do not use a preferred programming language but have various languages, including Python, Fortran, R, and C++ (Table 2). While the choice of programming language can influence the ease of implementing test suites (e.g., due to the availability of testing libraries), we observe that for these complex models, which often prioritize computational performance, implementing a test suite remains essential regardless of the programming language used."*

- [249] I will suggest not starting the sentence with a numeral.

  Thank you for your comment. The actual line is 349. We have revised the sentence accordingly and all other occurrences in the paper.

  (Section 3, Line 375-376) *"Our results indicate that 25% (8 of 32) of the GIMs have well-commented source code, i.e. 30-60% of all source lines of code are comment lines (Fig. 5)."*

- [429] I really enjoyed this section, and I think it will improve to the communication of the paper to put this section earlier in the manuscript, maybe between the introduction of the indicators and the analysis and the results.

  Thank you for your positive feedback on this section. We appreciate your suggestion to move it earlier in the manuscript. However, we believe that its current placement aligns with the overall logic and structure of the paper.

- [478] Same point about permissive and copyleft licenses. What do the authors mean by open-source licenses? Do you mean permissive?
  Thank you for your valuable comment. We refer you to our response to [154] regarding the discussion of open-source licenses

- [482] I am not sure that version control ensures software reproducibility, not with other important tools (environment or containerization, testing suite, etc).
  Thank you for your feedback. We agree with the reviewer that version control alone does not guarantee software reproducibility. As noted by Jiménez et al. (2017), version control facilitates the reproducibility of scientific results generated by all prior versions of the software. Therefore, we have revised our statement as follows:

  (Section 5, Line 531-533) "Version control can help you track and manage changes to your source code, which ensures the traceability of your software and facilitates reproducibility of scientific results generated by all prior versions of the software (Jiménez et al., 2017)."

General comments

- I strongly suggest sorting all the tables in decreasing or increasing order for readability.
  Thank you for the comment. All relevant tables including figures have been sorted accordingly.
- I suggest the authors to do a English style revision of the manuscript since there are different styles in the manuscript. This includes the starting of the sentences with numerals and the use or not use of contractions.
  We have revised relevant sentences that occur in the paper.
- In the same style that Figure 9, I think a figure at the beginning of the manuscript summarizing the indicators and what are good software practices will improve the manuscript.
  Thank you for the suggestion, however, we believe that Table 1 already effectively summarizes the indicators and good software practices.

Software sustainability of global impact models

Emmanuel Nyenah[1], Petra Döll[1, 2], Daniel S. Katz[3], and Robert Reinecke[4]

[1]Institute of Physical Geography, Goethe-University Frankfurt, 60438 Frankfurt am Main, Germany
[2]Senckenberg Biodiversity and Climate Research Centre (SBiK-F), 60438 Frankfurt am Main, Germany
[3]NCSA & CS & ECE & iSchool, University of Illinois Urbana-Champaign, Urbana, IL, 61801, USA
[4]Institute of Geography, Johannes Gutenberg-University Mainz, 55128 Mainz, Germany

Correspondence to: Emmanuel Nyenah (Nyenah@em.uni-frankfurt.de)

Abstract. Research software for simulating Earth processes enables estimating past, current, and future world states and guides policy. However, this modelling software is often developed by scientists with limited training, time, and funding, leading to software that is hard to understand, (re)use, modify, and maintain, and is, in this sense, non-sustainable. Here we evaluate the sustainability of global-scale impact models across ten research fields. We use nine sustainability indicators for our assessment. Five of these indicators – documentation, version control, open-source license, provision of software in containers, and the number of active developers – are related to best practices in software engineering and characterize overall software sustainability. The remaining four – comment density, modularity, automated testing, and adherence to coding standards – contribute to code quality, an important factor in software sustainability. We found that 29% (32 out of 112) of the global impact models (GIMs) participating in the Inter-Sectoral Impact Model Intercomparison Project were accessible without contacting the developers. Regarding best practices in software engineering, 75% of the 32 GIMs have some kind of documentation, 81% use version control, and 69% have open-source license. Only 16% provide the software in containerized form which can potentially limit result reproducibility. Four models had no active development after 2020. Regarding code quality, we found that models suffer from low code quality, which impedes model improvement, maintenance, reusability, and reliability. Key issues include a non-optimal comment density in 75%, insufficient modularity in 88%, and the absence of a testing suite in 72% of the GIMs. Furthermore, only 5 out of 10 models for which the source code, either in part or in its entirety, is written in Python show good compliance with PEP 8 coding standards, with the rest showing low compliance. To improve the sustainability of GIM and other research software, we recommend best practices for sustainable software development to the scientific community. As an example of implementing these best practices, we show how reprogramming a legacy model using best practices has improved software sustainability.

1 Introduction

Simulation models of the Earth system are essential tools for scientists and their outcomes are relevant for decision-makers (Prinn, 2013). They improve our understanding of complex subsystems of the Earth (Prinn, 2013; Warszawski et al., 2014) and enable us to perform numerical experiments that would otherwise be impossible in the real world, e.g., exploring future pathways (Kemp et al., 2022; Satoh et al., 2022; Wan et al., 2022). A specific class of simulation models of the Earth called ~~While so-called Earth System Models always include the simulation of atmospheric processes and thus compute climate variables and how they change due to greenhouse gas emissions, so-called~~ impact models enables us to quantitatively estimate the potential impacts of climate change on, e.g., ~~-~~floods (Sauer et al., 2021), droughts (Satoh et al., 2022), and food security (Schmidhuber and Tubiello, 2007). These impact models also quantify the historical development and current situation of key environmental issues such as water stress, wildfire hazard, and fish population. ~~-~~The outputs of these models whether data, publications or reports thus provide crucial information for policymakers, scientists, and citizens~~These impact models also quantify the historical development and current situation of, e.g., water stress, wildfire hazard, and fish population, thus providing crucial information for policymakers, scientists, and citizens~~. The central role of impact models can be seen in model intercomparison efforts of ISIMIP (Inter-Sectoral Impact Model Intercomparison Project) (ISIMIP, 2024; Warszawski et al., 2014) which encompasses more than 130 sectoral models (Frieler and Vega, 2019). ISIMIP uses bias-corrected climate forcings to assess the potential impacts of climate change in controlled experiments, and their outputs provide valuable contributions to the Intergovernmental Panel on Climate Change reports (Warszawski et al., 2014).

Impact models quantify physical processes related to specific components of the Earth system at various spatial and temporal scales by using mathematical equations. The complexity of impact models is influenced by the complexity of the included physical processes, the choice of the perceptual and mathematical model, the computational effort needed for simulation, as well as their spatial-temporal resolution and spatial extent of the simulated domain (Azmi et al., 2021; Wagener et al., 2021). This complexity can result in models with very large source codes (Alexander and Easterbrook, 2015).

The software for these impact models is categorized as research software, which includes “source code files, algorithms, computational workflows, and executables developed during the research process or for a research objective” (Barker et al., 2022). Impact modelling research software is predominantly developed and maintained by scientists without formal training in software engineering (Barton et al., 2022; Carver et al., 2022; Hannay et al., 2009; Reinecke et al., 2022). Most of these researchers are self-taught software developers (Nangia and Katz, 2017; Reinecke et al., 2022) with little knowledge of software requirements (specifications and features of software), industry-standard software design patterns (Gamma et al., 1994), good coding practices (e.g., using descriptive variable names), version control, software documentation, automated testing and project management practice (e.g. agile) (Carver et al., 2013, 2022; Hannay et al., 2009; Reinecke et al., 2022). We hypothesize that this leads to the creation of source code that is not well-structured, not easily (re)usable, difficult to modify

and maintain, has scarce internal documentation (code comments) and external documentation (e.g. manuals, guides, and tutorials), and poorly documented workflows.

Research software that suffers from these shortcomings is likely difficult to sustain and has severe drawbacks for scientific research. For example, it can impede research progress, decrease research efficiency, and hinder scientific progress, as implementing new ideas or correcting mistakes in code that is not well-structured is more difficult and time-consuming. In addition, it increases the likelihood of erroneous results, thereby reducing reliability and hindering reproducibility (Reinecke et al., 2022). We argue that these harmful properties can be averted, to some extent, with sustainable research software.

There are various interpretations of the meaning of "sustainable research software". Anzt et al. (2021) define research software as software that is ~~Anzt et al. (2021) describe it as research software that is~~ maintainable, extensible, flexible (adapts to user requirements), has a defined software architecture, is testable, has comprehensive in-code and external documentation, and is accessible (the software is licensed as Open Source with a digital object identifier (DOI) for proper attribution) (Anzt et al., 2021). For example, NumPy (https://numpy.org/) is a widely used scientific software package that exemplifies many of these qualities (Harris et al., 2020). Although NumPy is not an impact model, it is an ~~exceptional example~~exemplar of sustainable research software; it is open-source, maintains rigorous version control and testing practices, and is extensively documented, making it highly reusable and extensible for the scientific community.

Katz views research software sustainability as the process of developing and maintaining software that continues to meet its purpose over time (Katz, 2022). This includes adding new capabilities as needed by its users, responding to bugs and other problems that are discovered, and porting to work with new versions of the underlying layers, including software as well as new hardware (Katz, 2022). Both definitions share common aspects like the adaptation to user requirements but differ in scope and perspective. Katz's definition is more user-oriented, focusing on the software's ability to continue meeting its purpose over time. On the other hand, Anzt et al.'s definition is more developer-oriented, aiming to improve the quality and robustness of research software. We chose to adopt Anzt et al.'s definition in the following because it provides measurable qualities relevant to this study. In contrast, Katz's definition is more challenging to measure and evaluate but is likely closer to the reality of software development. For example, one of the models in our analysis is more than 25 years old (Nyenah et al., 2023) and thus certainly was sustained during that period, while at the same time, it does not meet some sustainability requirements of Anzt et al.'s definition. It is possible that such software can be sustained but requires substantial additional resources.

Recent advances in developing sustainable research software have led to a set of community standard principles: FAIR (findable, accessible, interoperable, reusable) for research software (FAIR4RS), aimed towards increasing transparency, reproducibility, and reusability of research (Barker et al., 2022; Chue Hong et al., 2022). Software quality which impacts sustainability overlaps with the FAIR4RS principles, particularly reusability, but is not directly addressed by them (Chue Hong et al., 2022). Reusable software here means software can be understood, modified, built upon, or incorporated into other

software (Chue Hong et al., 2022). A high degree of reusability is therefore important for efficient further development and improvement of research software, and thus for scientific progress. However, many models are not FAIR (Barton et al., 2022). To our knowledge, research software sustainability in Earth System Sciences has not been evaluated before.

As an example of complex research software in the Earth System Sciences, in this study, we assess the sustainability of the software of global impact models (GIMs) that participate in the ISIMIP project to investigate factors that contribute to sustainable software development. The GIMs belong to the ten research fields (or impact sectors): agriculture, biomes, fire, fisheries, health, lakes, water (resources), water quality, gGroundwater, and terrestrial biodiversity. In our assessment, we consider nine indicators of research software sustainability, five of them related to best practices in software engineering and four related to source code quality. We further provide first-order cost estimates required to develop these GIMs but do not address the cost of re-implementing or making code reproducible versus the cost of maintaining old code in this study. We also demonstrate how reprogramming legacy software using best practices can lead to significant improvements in code quality and thus sustainability. Finally, we offer actionable recommendations for developing sustainable research software for the scientific community.

2 Methods

2.1 Accessing GIM Source code

ISIMIP manages a comprehensive database of participating impact models (available in an Excel file at https://www.isimip.org/impactmodels/download/), which provides essential information such as model ownership, name, source code links, and simulation rounds. Initially, we identified 375 models across five simulation rounds (fast track, 2a, 2b, 3a, and 3b). As the focus of our analysis is on global impact models, we sorted the modelsdata by spatial domain and filtered out models operating at local and regional scales, resulting in a subset of 264 GIMs. We then removed duplicate models, prioritizing the most recent versions for inclusion, resulting in 112 unique models. For models with available source links, we obtained their source code directly. In instances where source links were not readily available, we conducted manual searches for source code by referring to code availability sections in reference papers. Additionally, we searched for source code using model names along with keywords such as "GitHub" and "GitLab" using the Google search engine. As of April 2024, 32 out of the 112 unique model source codes were accessible either through direct links from the ISIMIP database or via manual searches on platforms like GitHub and GitLab, as well as in code availability sections of reference papersAs of April 2024, 32 model source codes out of the 112 unique model source codes were accessible in the described way. However, it's important to note that our sample may suffer from a "survivor bias," as we are not investigating models that are no longer in use (GIMs that couldn't be sustained over time). This bias could potentially skew our analysis towards models that have survived i.e., they are still in use and their source code is accessible. Due to time constraints, we refrained from contacting developers for models that were not immediately accessible.

2.2 Research software sustainability indicators

We examine nine indicators of research software sustainability, distinguishing five indicators related to the best practice in software engineering and four indicators of source code quality (Table 1).

Table 1: Indicators used for the assessment of research software sustainability

No.	Indicator	Description
Best practices in software engineering		
1	Documentation	Enables software use and also makes software maintenance easier (Wilson et al., 2014).
2	Version control	Provides transparency and traceability throughout the software development lifecycle and enables collaboration between developers as well as user communities (Wilson et al., 2014).
3	Use of an open-source license	Allows code copying and reuse. This openness fosters a collaborative environment where the user community can provide valuable feedback and support. Users can potentially contribute to the software's development and maintenance, enhancing its overall quality (Jiménez et al., 2017).
4	Number of active developers	Prevent single points of failure in the development process and make software

		development as well as maintenance easier (Long, 2006).
5	Containerization	Makes the software easy to install and facilitates reproducibility (Nüst et al., 2020; Wilson et al., 2014).
	Source code quality	
6	Public availability of an (automated) testing suite [‡]	Shows that software functionality can be or was tested.
7	Compliance with coding standards (e.g. PEP 8) [‡]	Improves code quality, readability and makes maintenance easier (Capiluppi et al., 2009; Simmons et al., 2020; Wang et al., 2008).
8	Comment density [‡]	Precursor to software maintainability and re-usability (Arafat and Riehle, 2009; He, 2019; Stamelos et al., 2002).
9	Modularity [‡]	Necessary for extensible and flexible research software (Sarkar et al., 2008; Stamelos et al., 2002).

[‡] Indicators that impact research software quality

In the following, we describe the indicators and their rationale and how we evaluated the GIMs with respect to each indicator.

Documentation. Documentation is crucial for understanding and effectively utilizing software (Wilson et al., 2014). This includes various materials such as manuals, guides, tutorials that explain the usage and functionality of the software as well reference model description papers. When assessing documentation availability, relying solely on a reference model description paper may be insufficient, as it may not provide the level of detail necessary for the effective utilization and maintenance of the research software. All GIMs used in this assessment have an associated description or reference paper (see supplementary file ISIMIP_models.xlsx). Therefore, in addition to the reference model paper we checked for available manuals, guides, readme files, and tutorials. We consider any of these resources, alongside the reference model paper, as

documentation for the model. These resources provide essential information such as user, contributor, and troubleshooting guides, which are valuable for model usage and maintenance. In our assessment, we searched within the source code and official websites (if available). We also utilized the Google search engine to find model documentation by inputting model names along with keywords such as 'documentation,' 'manuals,' 'readme,' 'guides,' and 'tutorials'.

Version control. Version control systems such as Git and Mercurial facilitate track changes, and collaborative development, and provide a history of software evolution. To assess whether GIMs use version control for development, we focused on commonly used open-source version control hosting repositories such as GitLab, GitHub, BitBucket, Google Code, and Source Forge. The hostname such as “github” or “gitlab” in the source link of models provides clear indications of version control adoption in their development process. For other models, we searched within the Google search engine using model names and keywords such as “Bitbucket”, “Google Code”, and “Source Forge”. While we focus on identifying the use of version control systems, evaluating how version control was implemented during the development process — such as the use of modular commits, pull requests, discussions, and proper versioning — is a finer analysis that falls beyond the scope of this study. However, such practices are crucial for ensuring high-quality software development and collaborative practices.

Use of an oOpen- source license. ~~Open-source licenses foster collaboration and transparency by enabling community contributions and ensuring that software remains freely accessible.~~ We determined the existence of open-source licenses by checking license files within repositories or official websites against licenses approved by the Open Ssource Iinitiative (OSI) ~~approved licenses~~ (https://opensource.org/licenses). ~~This means these we select~~Specifically, we looked for licenses ~~which~~that conform to the Open Source Definition, which ensures that software can be freely used, modified, and shared (Colazo and Fang, 2009; Rashid et al., 2019). There are two major categories of open-source licenses: permissive licenses, such as MIT or Apache, ~~which~~that allow for minimal restrictions on how the software can be used (e.g., providing attribution), and copyleft licenses, like GPL, ~~which~~that require derivatives to maintain the same licensing terms (Colazo and Fang, 2009; Rashid et al., 2019). Although these licenses differ in their terms, both contribute to collaboration and transparency. In this study, we only check if the software is open-source, regardless of the type of open–source license.

Formatted: English (United States)

Number of active developers. The presence of multiple active developers serves as a safeguard against halts within the development process. In instances where a sole developer departs or transitions roles, the absence of additional ~~contributors~~ developers could lead to disruptions or challenges in maintaining and advancing the software. We measured the number of active developers by counting the individuals who made commits or contributions to the projects codebase within the period 2020-2024. A higher number of developers indicates a greater capacity for bug review (enhancing source code quality) and code maintenance. It can also lead to more frequent updates to the source code. On the other hand, the absence of active developers suggests potential stagnation in software evolution, possibly impacting the relevance and usability of the software.

Containerization. Containerization provides convenient ways to package and distribute software, facilitating reproducibility and deployment. It encapsulates an application along with its environment, ensuring consistent operation across various platforms (Nüst et al., 2020). Despite these benefits, containerization in high-performance computing systems encounters challenges like performance, prompting the proposal of solutions (Zhou et al., 2023). Some popular containerization solutions include Docker (https://www.docker.com/) and Apptainer (https://apptainer.org/)~~Singularity (https://sylabs.io/)~~. There are also cloud-supported container solutions such as Binder (https://mybinder.org/) with the capacity to execute a model with the computational environment requirements analogous to the concept of analysis-ready data and cloud-optimized formats for datasets (Abernathey et al., 2021). To evaluate the availability of container solutions, we conducted searches through reference papers, official websites, and software documentation for links to container images or image-building files such as "Dockerfiles", and "Apptainer ~~singularity~~ definition file (.def file)". In addition, we also searched through source code repositories to identify the previous stated images or image-building files. Lastly, we utilize the Google search engine, inputting the name of the GIM, the sector, and keywords such as "containerization", to ascertain if any other containerized solutions exist.

Public availability of an (automated) testing suite. Test coverage, which verifies the software's functionality, is the property of actual interest. However, research software may have an automatic testing suite but not provide information on test coverage or test results. As a practical approach, we consider the availability of a testing suite as a proxy for the ability to test software functionality. By examining testing suites within repositories, we gain insights into the developers' commitment to software testing, which contributes to enhancing software quality.

Compliance with coding standards. Coding standards are a set of industry-recognized best practices that provide guidelines for developing software code (Wang et al., 2008). Analysing the conformance to these standards can be complex, particularly when the source code is written in multiple languages~~.~~. Different languages may have various coding styles or style guides. For instance, multiple style guides are available and accepted by the Julia community (JuliaReachDevDocs, 2024). ~~-~~As an example analysis, we focused on GIMs containing Python in their source code as it is one of the most prevalent languages used in development. The tool used, known as Pylint , is designed to analyze Python code for potential errors and adherence to coding standards (Molnar et al., 2020; Obermüller et al., 2021). Pylint evaluates source files for their compliance with PEP8 conventions. To quantify adherence to this coding standard~~, it~~, it assigns a maximum score of ~~10 as~~10 as perfect compliance but has no lower bound (Molnar et al., 2020). We consider scores below 6 as indicative of weak compliance as code contain several violations.

Comment density. Good commenting practice is valuable for code comprehension and debugging. Comment density is an indicator of maintainable software (Arafat and Riehle, 2009; He, 2019). Comment density is defined as

$$Comment\ density = \frac{Number\ of\ lines\ of\ comment}{Total\ lines\ of\ code} \quad (1)$$

Here, the total lines of code (TLOC) include both comments and source lines of code (SLOC) (SLOCCount, 2024). SLOC is defined as the physical non-blank, non-comment line in a source file. ~~According to~~ Arafat et al. (2009) and He (2019~~),~~) suggest that comment density between 30-60% may be optimal~~the optimal comment density is 30-60% (Arafat and Riehle, 2009; He, 2019)~~. For most programming languages, this range is considered to represent a compromise between providing sufficient comments for code explanation and having too many comments that may distract from the code logic (Arafat and Riehle, 2009; He, 2019).

Modularity. Researchers typically pursue new knowledge by asking and then attempting to answer new research questions. When the questions can be answered via computation, this requires either building new software, adding new source code, or modifying existing source code. Addition and modification of source code are more easily achieved if the software has a modular structure that is implemented as extensible and flexible software (McConnell, 2004). Therefore, modularity is chosen as another indicator for research software sustainability. Modular programming is an approach where source codes are organised into smaller and well-manageable units (modules) that execute one aspect of the software functionality, such as the computation of evapotranspiration in a hydrological model (Sarkar et al., 2008; Trisovic et al., 2022). The aim is that each module can be easily understood, modified, and reused. Depending on the programming language, a module can be a single file (e.g. Python) or a set of files (e.g. C++).

To assess the modularity of research software, we use the TLOC per file as a metric. This metric reflects the organization of the source code into modules, each performing a specific function (Sarkar et al., 2008; Trisovic et al., 2022). We opted for this approach over measuring TLOC per function or subroutine due to variations in programming languages and the challenges associated with accurately measuring different functions using program-specific tools. For instance, in Python, a module that contains significantly more TLOC than usual (here over 1,000 TLOC) likely includes multiple functions. These functions may perform more than one aspect of the software's functionality, such as reading input files and computing other functions (e.g. evapotranspiration function), which contradicts the principle of modularity. Keeping the length of code in each file concise also enhances readability.

The ideal number of TLOC per file can vary with the language, paradigm (e.g., procedural or object-oriented), and coding style used in a software project (Fowler, 2019; McConnell, 2004). However, a common heuristic is to keep the code size per file under 1,000 lines to prevent potential performance issues such as crashes or slow program execution with some integrated development environments (IDEs) (Fowler, 2019; McConnell, 2004). IDEs are software applications that provide tools like code editors, debuggers, and build automation tools. As reported by Trisovic et al. (2022), based on interviews with top software engineers, a module with a single file should contain at least 10 lines of code, consisting of either functions or statements (Trisovic et al., 2022). We used this heuristic as a criterion for good modularity, assuming that 10-1,000 TLOC per

file indicates adequate modularity. We also varied the upper bounds of the total lines of code to 5,000 and 500 to investigate how modularity changes across models and sectors.

2.3 Source code counter

To count SLOC, comment lines, and TLOC of computational models, the counting tool developed by Ben Boyter (https://github.com/boyter50/scc) was used (Boyter ~~Ben~~, 2024) ~~(Sloc, Cloc, and Code, 2024)~~. This tool builds on the industrial standard source code counter tool called SLOCCount (Source Lines of Code Count) (SLOCCount, 2024).

2.4 Software cost estimation

The cost of developing research software is mostly unknown and depends on many factors, such as project size, computing infrastructure, and developer experience (Boehm, 1981). A model that attempts to estimate the cost of software development is the widely used Constructive Cost Model (COCOMO) (Boehm, 1981; Sachan et al., 2016), which computes the cost of commercial software by deriving the person-months required for developing the code based on the lines of code. Sachan et al. (2016) used the TLOC and effort estimates of 18 very large NASA projects (Average TLOC = 35,000) to optimise the parameters of the COCOMO regression model (Sachan et al., 2016). Effort in person months is estimated following Eq. (2):

$$Effort = 2.022817(kTLOC)^{0.897183} \quad (2)$$

where total lines of code are expressed in 1,000 TLOC (kTLOC) (Sachan et al., 2016). We use this cost model to estimate the cost of GIMs.

3 Results and Discussion

3.1 GIM programming languages and access points

The source code of the 32 GIMs is written in 10 programming languages (Fig. 1a). Fortran and Python are the most widely used, with 11 and 10 models, respectively. The dominance of Fortran stems from its performance, and the fact that it is one of the oldest programming languages designed for scientific computing (Van Snyder, 2007), and was the main such language used at the time some of the GIMs were originally built. This specialization makes it particularly suitable for tasks involving numerical simulations and complex computations. On the other hand, Python enjoys popularity among model developers due to readability, large user community, and rich ecosystem of packages, including those supporting parallel computing. R, C++ and C follow with 5, 5, and 4 models respectively (Fig. 1a). GIMs may employ one or more programming languages to target specific benefits the programming languages offer, such as readability and performance. For example, one of the studied models, HydroPy, written in Python, enhances its runtime performance by integrating a routing scheme built in Fortran (Stacke and Hagemann, 2021).

We find that 24 (75%) of the readily accessible 32 GIMs were hosted on GitHub (Fig. 1b). The rest are made available on GitLab (2, or 6%), Zenodo (4, or 12%), or the official website of the model (2, or 6%) (Fig. 1b~~, see supplementary file ISIMIP_models.xlsx~~).

We note that for one of the GIMs used for analysis, WaterGAP2.2e, only part of the complete model (the global hydrology model) was accessed (Müller Schmied et al., 2021). This might be the case for other models as well.

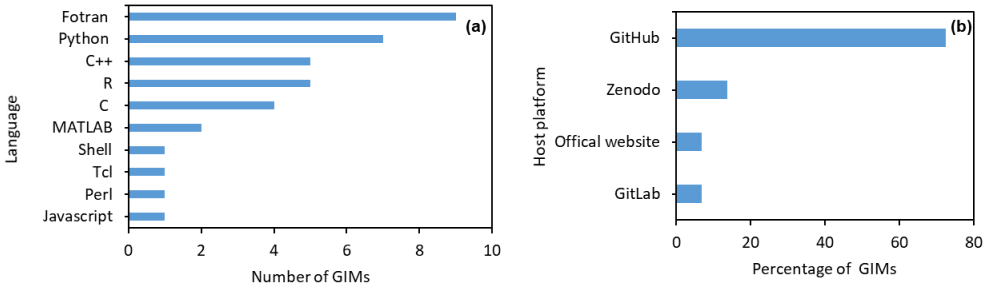

Figure 1: Programming languages for model development and model accessibility. (a) Bar plots showing programming languages used for developing 32 global impact models. (b) Bar plot showing open-source hosting platforms where 32 global impact models were accessed

3.2 Indicators of Software Sustainability

3.2.1 Software Engineering Practices

Documentation:

Our analysis reveals that 75% of the GIMs (24 out of 32) have publicly accessible documentation (Table 2). We observed a range of documentation formats across these GIMs. Specifically, 6 GIMs provided readme files, 13 had dedicated webpages for documentation, and 5 included comprehensive manuals (see supplementary file ISIMIP_models.xlsx). While README files tend to be more minimal and sometimes difficult to navigate, we observed that they generally contain essential information such as instructions on how to run the research software. The~~is~~ prevalence of documentation practices among most models underscores the importance of documenting research software. However, a notable portion (25%) of the studied models either lack documentation or documentation has not been made publicly available (Table 2).

Table 2: Availability of Documentation, Version Control, Open-Source License, Test Suite, and Container for 32 Global Impact Models across 10 Sectors in Earth System Science. '+', '-', 'not valid' and 'no info' represent the availability, unavailability, not OSI-approved and absence of information, respectively.

No.	Sector	Model	Year of Latest Version	Language	Documentation	Version control	Open Source License	Test Suite	Container
1	Agriculture	CGMS-WOFOST	no info	Fortran	+	+	+	-	-
2	Agriculture	DSSAT-Pythia	2024	Python	+	+	no info	+	+
3	Agriculture	EPIC-TAMU	2023	Fortran	+	no info	+	-	-
4	Agriculture	LPJmL	2024	C and JavaScript	+	+	+	-	-
5	Agriculture	ACEA	2024	Python	+	no info	not valid	-	-
6	Agriculture	LPJ-GUESS	2021	C++	+	no info	+	-	-
7	Biomes	CLASSIC	2020	Fortran	+	+	+	+	+
8	Biomes	MC2-USFS-r87g5c1	2022	C++, Fortran and C	+	+	+	-	-
9	Fire	SSiB4/TRIFFID-Fire	2021	Fortran	-	+	no info	-	-
10	Fisheries	BOATS	no info	MATLAB	-	+	no info	-	-
11	Fisheries	DBPM	no info	R	-	+	no info	+	-
12	Fisheries	EcoTroph	no info	R	+	+	no info	-	-
13	Fisheries	FEISTY	no info	MATLAB	-	+	no info	-	-
14	Fisheries	ZooMSS	2020	R and c++	+	+	+	-	-
15	Groundwater	G³M	2018	C++	+	+	+	+	-
16	Groundwater	parflow	2024	C, Tcl, python	+	+	+	+	+
17	Lakes	ALBM	2024	Fortran	+	+	+	-	-
18	Lakes	GOTM	2024	Fortran	+	+	+	+	-
19	Lakes	SIMSTRAT-UoG	2024	Fortran	+	+	+	+	+
20	Terrestrial biodiversity	BioScen15-SDM-GAM/GBM	no info	R	-	+	no info	-	-
21	Terrestrial biodiversity	BioScen1.5-MEM-GAM/GBM	no info	R	-	+	+	-	-
22	Vector-borne diseases (health)	VECTRI	no info	Fortran and python	+	+	+	-	-
23	Water	CWatM	2023	Python	+	+	+	+	-

24	Water	DBH	2006	Fortran	⁎+	no info	not valid	-	-
25	Water	HydroPy	2021	Python	⁎+	no info	⁎+	-	-
26	Water	PCR-GLOBWB	2023	Python	⁎+	⁎+	⁎+	-	-
27	Water	WBM	2023	Perl	⁎+	⁎+	⁎+	-	-
28	Water	WaterGAP2.2e	2023	C++	-	no info	⁎+	-	-
29	Water	VIC	2021	C and Python	⁎+	⁎+	⁎+	⁎+	⁎+
30	Water	H08	2024	Fortran and Shell	⁎+	⁎+	⁎+	-	-
31	Water	WAYS	no info	Python	-	⁎+	⁎+	-	-
32	Water quality	DynQual	2023	Python	⁎+	⁎+	no info	-	-
		Total			24	26	22	9	5

Version control:

We find that 81% (26 out of 32) of GIMs uses Git as their version control system reflecting the widespread acceptance of Git across the sectors (Table 2). In the remaining cases, ~~GIMs were made available on Zenodo and the models' official websites (Table 2, Fig. 1b);~~ information about the specific version control system used for these GIMs was unavailable. ~~Developers' preference for Git highlights its user-friendly nature and effectiveness in supporting collaborative efforts.~~

Use of an open source license:

Most of the research software, 69% (22 out of 32), have open-source licenses (Table 2) with the "GNU General Public License" being the commonly used license (56%, 18 out of 32) (Fig. 2). However, the remaining 31% (10 out of 32) either have no information on the license even though the source code is made publicly available (8 or 25% of GIMs) or uses license which is not OSI-approved (1 GIM each with creative commons license and user agreement) (Fig. 2). This ambiguity or absence of licensing details can deter potential users and contributors, as it raises uncertainties about the permissions and restrictions associated with the software.

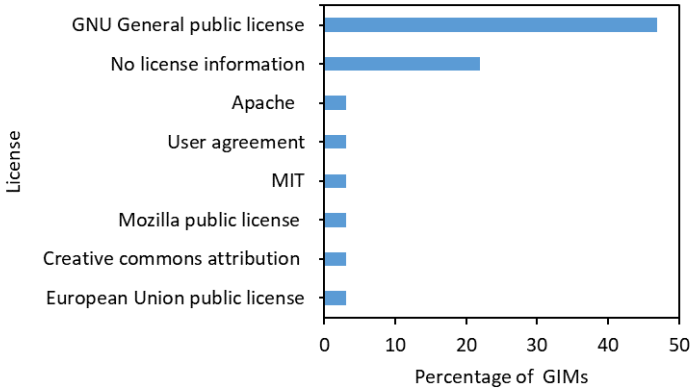

Figure 2: License distribution for 32 global impact models across 10 sectors. 8 (25%) GIMs lack license information, and two (6%) GIMs have licenses that are not OSI-approved.

335

Number of active developers:

Our results reveal a diverse distribution of active developers across the GIMs. We have excluded GIMs without version control information from our results, as without could not be evaluated for this indicator, resulting in data for 26 GIMs. Notably, GIMs such as parflow, CWatM, LPJmL, and GOTM have a significant number of active developers, with 28, 12, 11, and 8 developers
340 respectively (Fig. 3). These values correlates with the size of GIMs source code, as evidenced by TLOC (282,722 for ParFlow, 33,286 for CWatM, 136,002 for LPJmL, and 29,477 for GOTM.). However, models such as WAYS, VIC, BioScen1.5-MEM, and CGMS-WOFOST had no active developers during the considered period of 2020 to 2024 (Fig. 3).

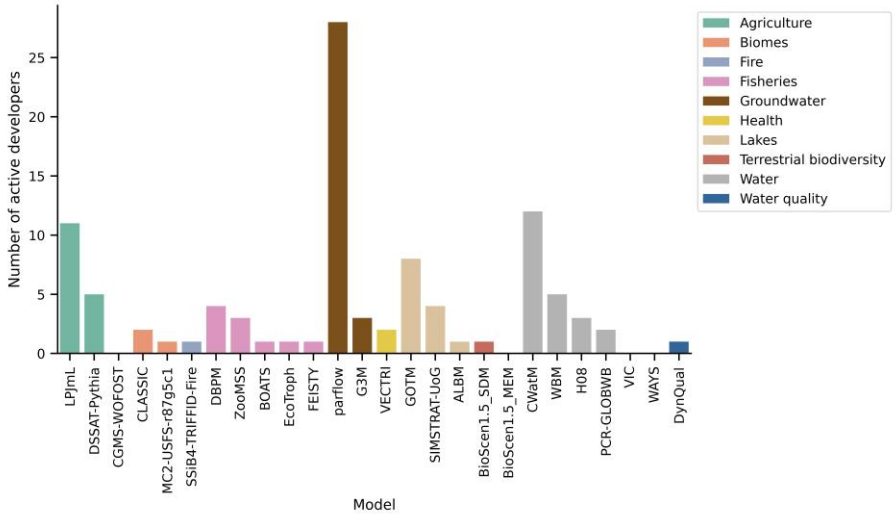

Figure 3: Number of active developers within 5 years (2020-2024) for 26 global impact models across 10 sectors. The results for the 6 remaining GIMs could not be measured since version control information could not be found. Zero value means no active developers within the 5 year period. ~~Bars~~The models are sorted within each sector ~~in~~by ~~decreasing~~the ~~order~~number of active developers ~~within each sector~~.

Containerization:

Only 5 (16%) of the GIMs have implemented containerized solutions (Table 2). ~~Apart from~~While the CLASSIC model~~, which~~ uses Apptainer, the other four models use Docker as their containerization technology. The CLASSIC container is shared via Zenodo, whereas the Docker containers for the remaining models are distributed through GitHub. Despite the recognized benefits of containerization in promoting reproducible research, provisioning of the software in containers is not yet a common practice in GIM development.

3.2.2 Code Quality Indicators

Public availability of an (automated) testing suite:

Our research indicates that 28% (9 out of 32) of the examined GIMs have a testing suite in place to test the software's functionality (Table 2). The models with test suites do not use a preferred programming language but have various languages, including Python, Fortran, R, and C++ (Table 2). While the choice of programming language can influence the ease of implementing test suites (e.g., due to the availability of testing libraries), we observe that for these complex models, which often prioritize computational performance, implementing a test suite remains essential regardless of the programming language used. A typical test might involve ensuring that a global hydrological model such as CWatM runs without errors with different configuration file options (e.g., different resolutions and basins) (Burek et al., 2020). However, this practice is not widespread in the development of GIMs, with the majority (72%) lacking a testing suite (Table 2). This absence of testing suites in GIM development highlights a deficiency in the developers' dedication to software testing. The presence of a testing suite could lead to more frequent testing, thereby enhancing the overall quality of the software.

Compliance with coding standards:

We restricted our analysis to GIMs that include Python in their source code due to challenges described in section 2.2. Among the ten models we examined, we observed varying levels of adherence to the PEP8 style guide for Python. Five models (DSSAT-Pythia, parflow, HydroPy, VIC, and WAYS) demonstrated good compliance, each achieving a lint score above 6 out of a maximum of 10 (Fig. 4). Good compliance indicates minimal PEP8 code violations. However, the remaining five models showed lower compliance, with lint scores between 0 and 3 (Fig. 4). This suggests numerous violations leading to potential issues like poor code readability and an increased likelihood of bugs, which could hinder code maintenance.

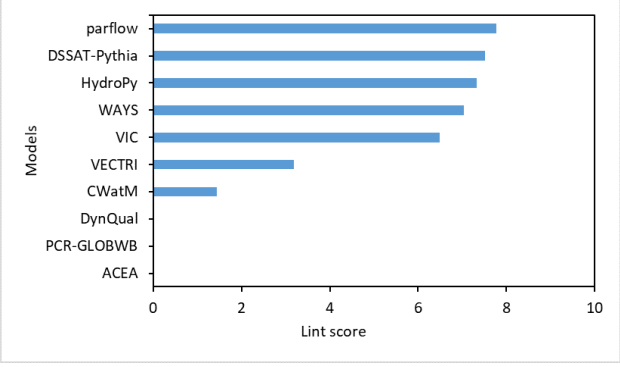

Figure 4: Lint scores of GIMs containing Python code.

Comment density:

Our results indicate that 25% (8 of 32) of the GIMs have well-commented source code, i.e. 30-60% of all source lines of code are comment lines (Fig. 5). The remaining 75% (24) of the GIMs have too few comments, which indicates that overall, commenting practice is low across the studied research fields.

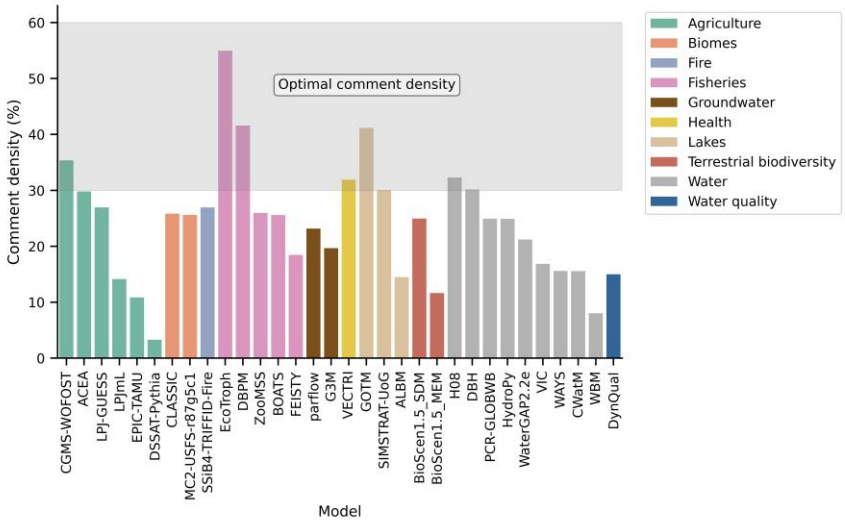

Figure 5: Comment density per model across 10 sectors. The grey zone denotes the optimal comment density (Arafat and Riehle, 2009; He, 2019). ~~Bars~~Models are sorted within each sector ~~in~~by decreasing ~~order within each sector~~comment density.

Modularity:

The investigated GIMs have TLOC values between 262 and 500,000, distributed over 6-2400 files (Fig. 6). Only 4 out of the 32 (12%) simulation models (EcoTroph, ZooMSS, HydroPy, and BioScen1.5_SDM) meet the criterion of having between 10 and 1,000 TLOC per file (Fig. 6). The remaining 28 GIMs either had at least one file exceeding 1,000 TLOC, which likely could be divided into smaller modules with distinct functionality or had at least one file less than 10 TLOC, which makes source code harder to navigate and understand, especially if the files are not well-named or documented. We also performed a sensitivity analysis by changing the criterion to 5,000 and 500 TLOC per file with the same lower limit of 10 TLOC. Nine simulation models (LPJmL, MC2-USFS-r87g5c1, EcoTroph, ZooMSS, BioScen1.5_SDM, BioScen1.5_MEM, H08, HydroPy, and DynQual) meet the 5,000-line criterion and two models (EcoTroph, ZooMSS) met the 500-line criterion (Fig.

6). Because code comments, which are included in TLOC, aid code comprehension, we also assessed modularity using the criterion of 1,000 SLOC instead of 1,000 TLOC with 10 SLOC. Three GIMs (ZooMSS, BioScen1.5_SDM, and Hydropy) meet the 10-1,000 SLOC criterion (see supplementary Fig. S1).

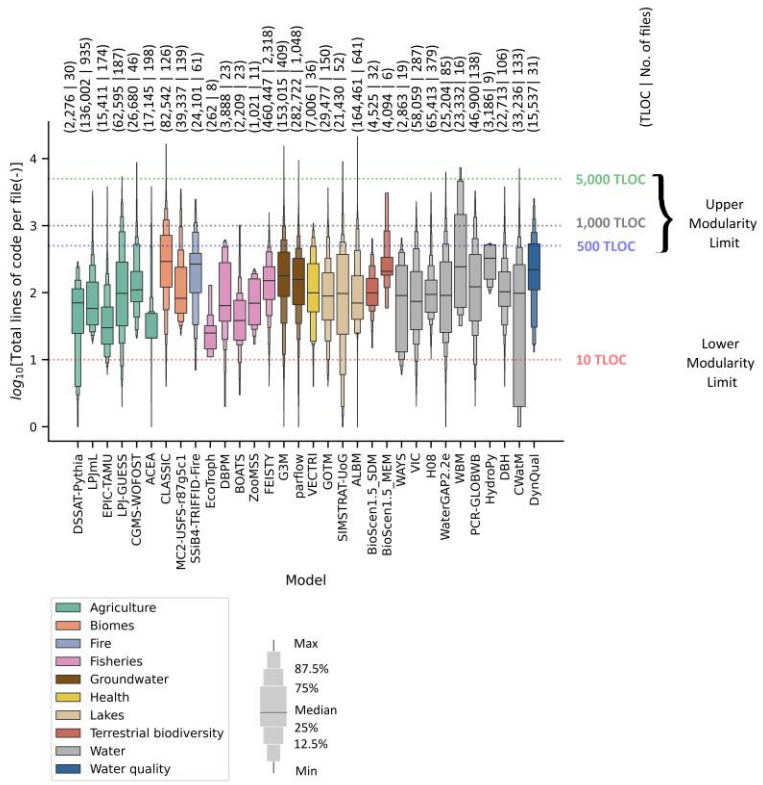

Figure 6: Letter value plot (Hofmann et al., 2017) of total lines of code (TLOC) per file (logarithmic scale) of 32 global impact models across 10 sectors. The dotted blue, black, and green lines show upper modularity limits, the dotted red line the lower limit. The values (x|y) in the upper section of Fig. 6, show, for each GIM, TLOC | Number of files.

3.3 Cost of GIM software development

~~Research software is a valuable and complex research tool that often requires a lot of effort to develop and maintain (Carver et al., 2022; Reinecke et al., 2022).~~ T~~t~~o provide a rough cost estimate for the software development of the 32 impact models~~.~~, ~~Here~~ we use the cost estimate model from Sachan et al. (2016) (see section 2.4) in a scenario of "*what if we would hire a commercial software company to develop the source code of the global impact models*?" ~~to provide a rough cost estimate for the software development of the 32 impact models.~~ This cost estimate does not include developing the science (e.g., concepts, algorithms, and input data) nor costs of documenting, running, and maintaining the software, only the implementation of code. We assume that the COCOMO model is transferable to research software as the NASA projects used in cost model contain software that is similar to research software. As the TLOC of the impact model codes ranges from 262 to 500,000 TLOC (Fig. 7), the effort required to produce these models ranges from 1 to 495 person-months (Fig. 7). With a small additive change of ~~(~~±0.1~~)~~ of the~~in~~ COCOMO model coefficients, the range of estimated effort changes to ~~ranges from approximately~~ 1 to 255 person-months in the case of ~~on~~ -0.1 ~~scenario~~, and to 1 to~~up to about~~ 960 person-months ~~on the~~in the case of +0.1 ~~scenario~~ (Supplementary Fig. S2).

The results suggest that these complex research software programs are expensive tools that require adequate funding for development and maintenance to make them sustainable. This is consistent with previous studies that have highlighted funding challenges for developing and maintaining sustainable research software in various domains (Carver et al., 2013, 2022; Eeuwijk et al., 2021; Merow et al., 2023; Reinecke et al., 2022). Merow et al. (2023) also emphasized that the accuracy and reproducibility of scientific results increasingly depend on updating and maintaining software. However, the incentive structure in academia for software development — and especially maintenance — is insufficient (Merow et al., 2023).

Field Code Changed
Formatted: Spanish (Spain)
Formatted: Spanish (Spain)

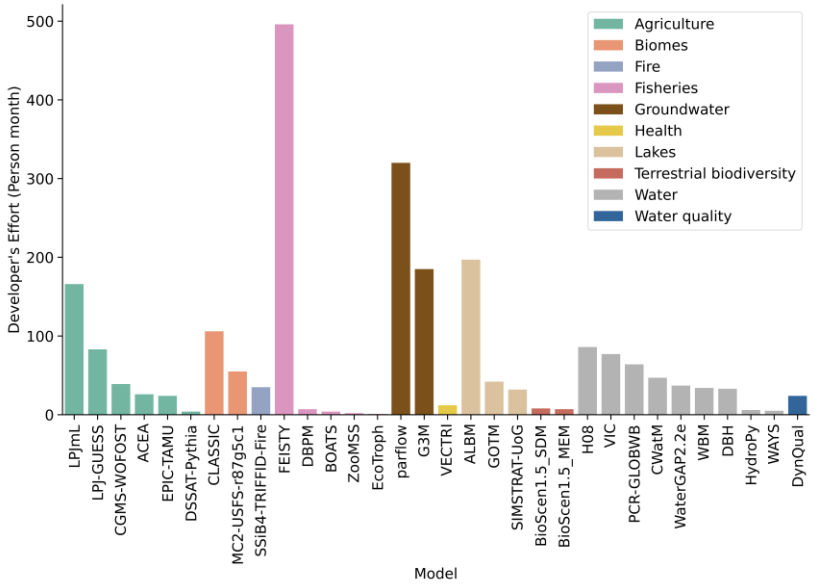

Figure 7 : Effort estimates of 32 global impact models across 10 sectors~~. Each bar represents one GIM. Darker colours represents large TLOC and effort values~~. ~~. Bars~~Models are sorted within each sector by~~in~~ decreasing ~~order within each sector~~amount of developer's effort.

3.4 Case Study: Reprogramming legacy simulation models with best practices

Legacy codes often suffer from poor code readability and poor documentation, which hinder their maintenance, extension, and reuse. To overcome this problem, some of GIMs such as HydroPy (Stacke and Hagemann, 2021; Stacke, Tobias and Hagemann, Stefan, 2021) were reprogrammed, while others (e.g., WaterGAP, Nyenah et al., 2023) are in the process of being reprogrammed. We compared the legacy global hydrological model MPI-HM (in Fortran) and its reprogrammed version HydroPy (in Python) in terms of the sustainability indicators. The reprogrammed model has improved modularity (Fig. 8a), which supports source code modification and extensibility. HydroPy has good compliance with the PEP8 coding standard, which improves readability and lower the likelihood of bugs in source code (Fig. 4). It has an open-source license and a persistent digital object identifier, which makes it easier to cite (Editorial, 2019). This research software refers to its associated publication for information and instructions on Zenodo to setup and run Hydropy. A software testing suite and container are not yet available.

We find that HydroPy has a comment density of 25% (Fig. 8b), which is below the desired 30-60% range, but the developers argue that "the code is self-explanatory and comments are added only when necessary" (Stacke, 2023). MPI-HM has more comments (49%, Fig. 8b) because of its legacy Fortran code that limits variable names to a maximum length of 8 characters, so they have to be described in comments. Another reason is that the MPI-HM developers kept track of the file history in the header, which adds to the comment lines in MPI-HM. This raises a question: *Is the comment density threshold metric still valid if a code is highly readable and comprehensive~~self-explanatory~~*? The need for comments can depend on the language's readability (Python vs. Fortran), the complexity of the implemented algorithms and concepts, and the coder's expertise. While a highly readable and well-structured code might require fewer explanatory comments, the definition of "readable" itself can be subjective and context-dependent. Nevertheless, comment density remains a valuable metric, especially for code written by novice developers.

The HydroPy model is a great starting point for sustainable research software development, as it illustrates the application of the sustainability indicators. Reprogramming legacy code not only allows developers to use more descriptive variable names, which increases code readability and maintainability, but also enables them to share their code and documentation with the scientific community through open source platforms and tools. This practice enhances transparency and accountability, as the code can be inspected, verified, and reproduced by others. Reprogramming legacy code with best practices always improves code quality, which makes software more sustainable.

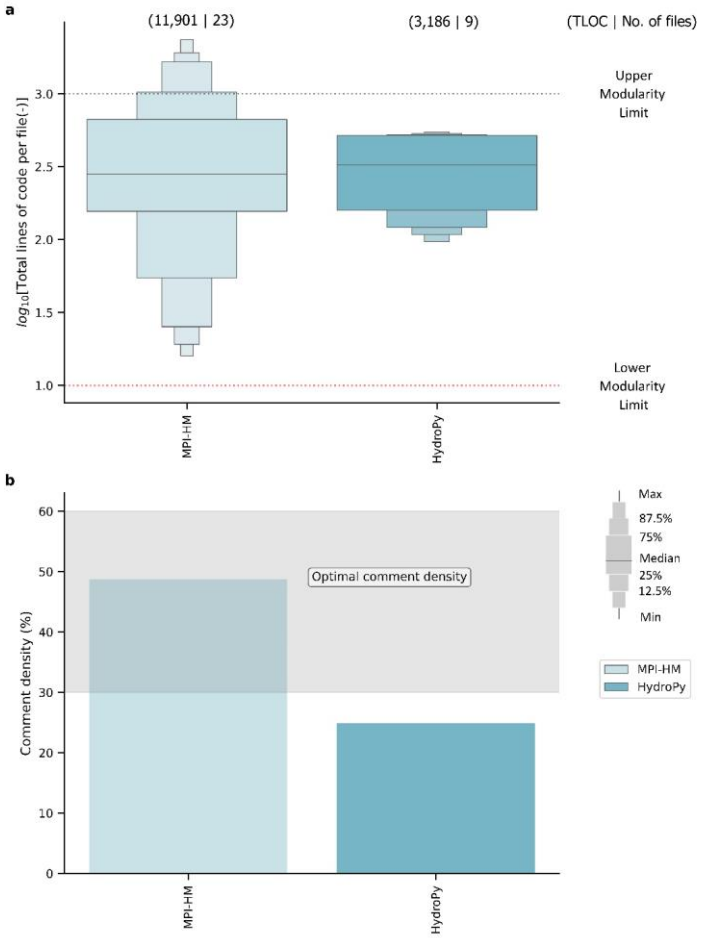

Figure 8: Modularity and commenting practice of a legacy (MPI-HM) and reprogrammed (HydroPy) global simulation model. (a) Letter value plot of total lines of code per file (logarithmic scale) of each model. The dotted black (red) line shows the upper (lower) modularity limit defined as the maximum of 1000 (minimum of 10) total lines of code per file. The values (x|y) shown in the upper section of Fig. 8a correspond to (TLOC | Number of files per model). (b) Comment density per model. The grey zone in Fig. 8b denotes the optimal comment density.

4 Limitations

Our study has limitations in the following regards. In the interest of timely analysis, we did not contact the developers ~~developers were not contacted fo~~of~~r~~ models that were not readily available. This means that older software, particularly ~~those~~that written in less common or outdated programming languages, might be underrepresented. Additionally, software with higher code quality and better documentation is more likely to be made readily available and thus may have been selected more frequently. This selection process could introduce bias in the distribution of models. Specifically, the simulation model distribution does not favour certain sectors. For instance, only 2 out of the 18 global biomes impact models were readily available and therefore included in our assessment. This may affect the generalizability of our findings across different domains of Earth System Sciences~~Our study has limitations in the following regards. In the interest of timely analysis, developers were not contacted for models that were not readily available, causing a bias in the distribution of models. Specifically, the simulation model distribution does not favour certain sectors. For instance, only 2 out of the 18 global biomes impact models were readily available and therefore included in our assessment. This may affect the generalizability of our findings across different domains of Earth System Sciences~~.

Moreover, our sustainability indicators do not cover other relevant aspects of sustainable research software, such as user base size, code development activity (e.g. frequency of code contributions, date of last update or version), number of publications and citations, coupling and cohesion, information content of comments, software adaptability to user requirement and interoperability. A larger user base often results in more reported bugs, which ultimately enhances software reliability. However, determining the exact size of the user base presents challenges due to data reliability issues. Additionally, there is the question of whether to include model output (data) users as part of the user base. Code development activity, such as the frequency of code contributions, indicates an ongoing commitment to improving and maintaining the software, but it does not necessarily reflect the quality of those contributions. In addition, the date of the last update or version is a useful metric, but it can be complex to interpret. For instance, research software might have an old last update date but still be widely used and reliable. Hence, these metrics were not evaluated here. The number of publications and citations referencing a model serves as an indicator of its impact and relevance within the research community. Yet, collecting and analysing this data is a time-consuming and complex task. We further did not evaluate the interdependence of software modules (coupling) and how functions in a module work towards the purpose of the module (cohesion) (Sarkar et al., 2008), as language-specific tools are required to evaluate such properties.

In addition to the previously discussed limitations, the indicator analysed in this study are quantitative metrics that can be measured. Factors such as information content of comments, software adaptability to user requirements and interoperability (Chue Hong et al., 2022) are examples of qualitative metrics that contribute to software sustainability. However, qualitative analysis is outside the scope of this study. We focus on measurable metrics that can be easily applied by the scientific community and by novice developers.

Also, we did not explore the analysis of code compliance to standards for other programming languages used for GIM development. Specifically for Python, the Pylint tool provides a lint score for all source code analysed, making it easier to interpret results. However, the tools for other languages (e.g., lintr for R) does not have this feature, which presents challenges in result interpretation.

Furthermore, future research could compare the sustainability levels of impact models developed by professional software design teams with those created in academic settings by non-professional software developers.

5 Recommendations

Making our research software sustainable requires a combined effort of the modelling community, scientific publishers, funders, and academic and research organizations that employ modelling researchers (Barker et al., 2022; Barton et al., 2022; McKiernan et al., 2023; Research Software Alliance, 2023). Some scientific publishers, research organizations, funders and scientific communities adopted and proposed solutions to this challenge, such as 1) requiring that authors make source code and workflows available, 2) implementing FAIR standards, 3) providing training and certification programs in software engineering and reproducible computational research, 4) providing specific funding for sustainable software development, 5) establishing the support of permanently employed research software engineers for disciplinary software developers and 6) recognizing the scientific merit of sustainable research software by acknowledging and rewarding the development of high-quality, sustainable software as valuable scientific output in evaluation, hiring, promotions, etc. (Carver et al., 2022; Döll et al., 2023; Editorial, 2018; Eeuwijk et al., 2021; Merow et al., 2023). This software should be treated as a citable academic contributions, and included, for instance, in PhD theses (Merow et al., 2023). (), and and 6) establishing the support of permanently employed research software engineers for disciplinary software developers (Carver et al., 2022; Döll et al., 2023; Editorial, 2018; Eeuwijk et al., 2021; Merow et al., 2023).

To assess the current state of these practices in eEarth system science, we conducted an analysis of sustainability indicators across global impact models. Our findings reveal that while some best practices are widely adopted, others are significantly lacking. Specifically, we found high implementation rates for documentation, open-source licensing, version control, and active developer involvement. However, four out of eight sustainability indicators showed poor implementation: automated testing suites, containerization, sufficient comment density, and modularity. Additionally, only 50% of Python-specific models adhere to Python-based coding standards. These results highlight the urgent need for improved software development practices in eEarth system science. . Based on the results of our study, as well as the findings from existing literature, we propose the following actionable best practices for researchers developing software In addition, we recommend the following actionable best practices for researchers developing software, based on literature and our own experience (summarized in Fig. 9):

Commented [IPG1]: Insert citation.
Formatted: Spanish (Spain)
Field Code Changed
Formatted: Spanish (Spain)
Formatted: Spanish (Spain)
Field Code Changed
Formatted: Spanish (Spain)
Formatted: Spanish (Spain)

- *Choose ~~a~~ project management practices that align~~s~~ with your institutional environment, culture, and project requirements~~Apply project management practices in software development (e.g., Agile)~~:* This can help plan, organize, and monitor your software development process, as well as improve collaboration and communication within your team and with stakeholders. Project management practices ~~can~~ also help ~~you~~ identify and mitigate risks, manage~~s~~ ~~handle~~ changes, and deliver quality software on time and within budget (Anzt et al., 2021). While traditional methods may be better suited for projects with fixed requirements, certain principles from more flexible frameworks, such as Agile, can provide benefits in environments where requirements evolve or adaptability is critical. For example, Agile's iterative approach allows for incorporating changing research questions and hence software modifications or extensions, improving responsiveness to new developments (Turk et al., 2005).
- *Consider software architecture (organisation of software components) and requirements (user needs):* This will help design your software in a way that meets the needs and expectations of your users. Considering software architecture (such as Model-Controller-View (Guaman et al., 2021)) and user requirements helps to design a software system that has a clear and coherent structure, well-defined functionality, and suitable quality (Jay and Haines, 2019).
- *Select an open-source license:* Choosing an open-source license will make your software accessible and open to the research community, enable collaborations with other developers and contributors, as well as protect your intellectual property rights (Anzt et al., 2021; Carver et al., 2022). Accessible software is crucial to reduce reliance on email requests (Barton et al., 2022).
- *Use version control:* -Version control can help you track and manage changes to your source code, which ensures the traceability of your software and facilitates reproducibility of scientific results generated by all prior versions of the software ~~and ensure your software is reproducible and traceable~~ (Jiménez et al., 2017). Platforms like GitHub and GitLab are commonly used for this purpose. However, it's important to note that these platforms are not archival - the code can be removed by the developer at any time. A current best practice is to use both GitHub and GitLab for development, and to archive major releases on Zenodo or another archival repository.
- *Use coding standards accepted by your community (e.g., PEP8 for Python), good and consistent variable names, design principles, code quality metrics, peer code review, linters and software testing:* Coding standards help you write clear, consistent, and readable code that follows the best practices of your programming language and domain. It is key that developers consistently follow a coding style recognized by the relevant language community. Good variable names are descriptive and meaningful, reflecting the role and value of the variable. Design principles help adhere to the principles of sustainable research software, such as modularity, reusability and interoperability. These principles also guide the design of software by determining, for instance, the interaction of classes addressing aspects such as separation of concerns, abstraction, and encapsulation (Plösch et al., 2016).
    - . Code quality metrics can help measure and improve the quality of source code in terms of readability, maintainability, reliability, modularity and reusability. (Stamelos et al., 2002). Peer code review and linters (tools that analyse source code for potential errors) can help detect and fix errors, and vulnerabilities in your code, as well as

Formatted: Indent: Left: 1.27 cm, No bullets or numberi

improve your coding skills and knowledge (Jay and Haines, 2019). Software testing verifies if the research software performs as intended.

- *Make internal and external documentation comprehensible*: This can help you explain the purpose, functionality, structure, design, usage, installation, deployment, and maintenance of your software to yourself and others. Internal documentation refers to the comments and annotations within your code that describe what the code does and how it works. External documentation refers to manuals, guides, tutorials and any material that provide information about your software to users and developers. Comprehensible documentation can help you make your software more understandable, maintainable, and reusable. (Barker et al., 2022; Carver et al., 2022; Jay and Haines, 2019; Reinecke et al., 2022; Wilson et al., 2014)
- *Engage the research software community in the software development process.* This will help you get feedback, support, advice, collaboration, contribution and recognition from other researchers and developers who share your interests and goals. Engaging the research software community via conferences and workshops can also help you disseminate your software to a wider audience, increase its impact and visibility, and foster open science practices (Anzt et al., 2021). Additionally, consider utilizing containerization technologies, such as Docker, to simplify the installation and usage of your software (Nüst et al., 2020). It helps eliminate the "it works on my machine" problem. This approach also facilitates easy sharing of your software with software users. Furthermore, implement continuous integration and automated testing to maintain the quality and reliability of your code (Ståhl and Bosch, 2014). Continuous integration merges code changes from contributing developers frequently and automatically into a shared repository.
- *Integrate automation in development practices.* Automation plays a key role in streamlining software development by reducing manual effort and ensuring consistency (Wijendra and Hewagamage, 2021). We encourage developers to integrate automation into their workflows to improve efficiency. For instance, ~~using GitHub Actions,~~ developers can use GitHub Actions to automate various tasks like running test suites, generating documentation, ensuring adherence to coding standards, and managing dependencies.

Formatted: Justified

Formatted: Font: Italic

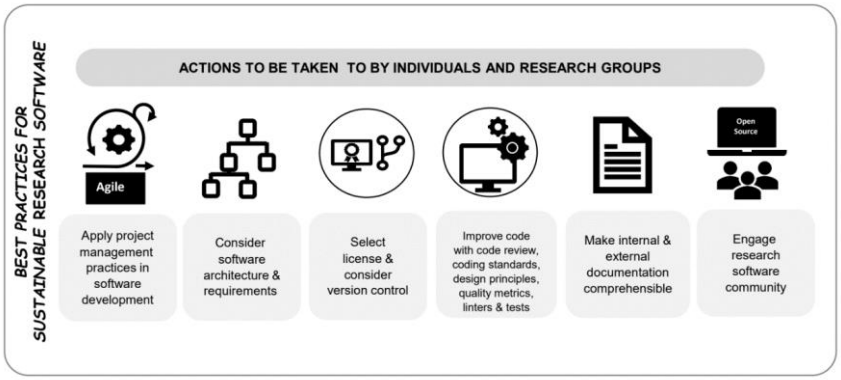

Figure 9: Actionable best practices for sustainable research software. The image summarizes the actions that modelling communities and individual developers should take, such as following project management practices, coding standards, reviews, documentation and community engagement strategies. These actions can help produce high-quality, robust, and reusable software that can be maintained.

6 Conclusion

The studied Earth system models are valuable and complex research tools that exhibit strengths and weaknesses in the use of certain software engineering practices (strengths, for example, in version control, open-source licensing, and documentation). However, notable areas remain for improvement, particularly in areas such as containerization and factors affecting code quality like comment density, modularity, and the availability of test suites. These shortcomings hinder the sustainability of such research software; they limit research reliability, reproducibility, collaboration, and scientific progress. To address this challenge, we urge all stakeholders, such as scientific publishers, funders, as well as academic and research organizations, to facilitate the development and maintenance of sustainable research software. We also propose to use best practices for the developers of research software such as using project management and software design techniques, coding reviews, documentation, and community engagement strategies. We further suggest reprogramming the legacy code of well-established models. These practices can help achieve higher-quality code that is more understandable, reusable, and maintainable.

Efficient computational science requires high-quality software. While our study primarily focuses on Earth System Sciences, our assessment method and recommendations should be applicable to other scientific domains that employ complex research software. Future research could explore additional sustainability indicators, such as user base size, code development activity

(e.g. frequency of code contributions), software adaptability and interoperability, as well as code compliance standards for various programming languages.

Code Availability

The Python scripts utilized for analysis can be accessed at https://zenodo.org/doi/10.5281/zenodo.10245636 . Additionally, the line counting tool developed by Ben Boyter is available through the GitHub repository: https://github.com/boyter/scc.

Data Availability

The results obtained from the line count analysis are accessible at https://zenodo.org/doi/10.5281/zenodo.10245636.
For convenient downloads of global impact models, links to the 32 global impact models, along with the respective dates of access, can be found in an Excel sheet named "ISIMIP_models.xlsx." present in the Zenodo repository.

Author contributions

EN and RR designed the study. EN performed the analysis and wrote the paper with significant contributions from PD, DK, and RR. RR and PD supervised EN.

Competing interests

The authors declare no competing interests.

Acknowledgements

EN, RR, and PD acknowledge support from Deutsche Forschungsgemeinschaft (DFG) (Project number 443183317)

References

Abernathey, R. P., Augspurger, T., Banihirwe, A., Blackmon-Luca, C. C., Crone, T. J., Gentemann, C. L., Hamman, J. J., Henderson, N., Lepore, C., McCaie, T. A., Robinson, N. H., and Signell, R. P.: Cloud-Native Repositories for Big Scientific Data, Computing in Science & Engineering, 23, 26–35, https://doi.org/10.1109/MCSE.2021.3059437, 2021.

Alexander, K. and Easterbrook, S. M.: The software architecture of climate models: a graphical comparison of CMIP5 and EMICAR5 configurations, Climate and Earth System Modeling, https://doi.org/10.5194/gmdd-8-351-2015, 2015.

ISIMIP: https://www.isimip.org/, last access: 23 March 2024.

Anzt, H., Bach, F., Druskat, S., Löffler, F., Loewe, A., Renard, B., Seemann, G., Struck, A., Achhammer, E., Aggarwal, P., Appel, F., Bader, M., Brusch, L., Busse, C., Chourdakis, G., Dabrowski, P., Ebert, P., Flemisch, B., Friedl, S., Fritzsch, B., Funk, M., Gast, V., Goth, F., Grad, J., Hegewald, J., Hermann, S., Hohmann, F., Janosch, S., Kutra, D., Linxweiler, J., Muth, T., Peters-Kottig, W., Rack, F., Raters, F., Rave, S., Reina, G., Reißig, M., Ropinski, T., Schaarschmidt, J., Seibold, H., Thiele, J., Uekermann, B., Unger, S., and Weeber, R.: An environment for sustainable research software in Germany and beyond: current state, open challenges, and call for action [version 2; peer review: 2 approved], F1000Research, 9, https://doi.org/10.12688/f1000research.23224.2, 2021.

Arafat, O. and Riehle, D.: The comment density of open source software code, in: 2009 31st International Conference on Software Engineering - Companion Volume, 195–198, https://doi.org/10.1109/ICSE-COMPANION.2009.5070980, 2009.

Azmi, E., Ehret, U., Weijs, S. V., Ruddell, B. L., and Perdigão, R. A. P.: Technical note: "Bit by bit": a practical and general approach for evaluating model computational complexity vs. model performance, Hydrology and Earth System Sciences, 25, 1103–1115, https://doi.org/10.5194/hess-25-1103-2021, 2021.

Barker, M., Chue Hong, N. P., Katz, D. S., Lamprecht, A.-L., Martinez-Ortiz, C., Psomopoulos, F., Harrow, J., Castro, L. J., Gruenpeter, M., Martinez, P. A., and Honeyman, T.: Introducing the FAIR Principles for research software, Scientific Data, 9, 622, https://doi.org/10.1038/s41597-022-01710-x, 2022.

Barton, C. M., Lee, A., Janssen, M. A., van der Leeuw, S., Tucker, G. E., Porter, C., Greenberg, J., Swantek, L., Frank, K., Chen, M., and Jagers, H. R. A.: How to make models more useful, Proceedings of the National Academy of Sciences, 119, e2202112119, https://doi.org/10.1073/pnas.2202112119, 2022.

Boehm, B. W.: Software engineering economics, Prentice-Hall, Englewood Cliffs, N.J, 57–96, 1981.

Boyter Ben: https://github.com/boyter/scc., last access: 3 March 2024.

Burek, P., Satoh, Y., Kahil, T., Tang, T., Greve, P., Smilovic, M., Guillaumot, L., Zhao, F., and Wada, Y.: Development of the Community Water Model (CWatM v1.04) – a high-resolution hydrological model for global and regional assessment of integrated water resources management, Geoscientific Model Development, 13, 3267–3298, https://doi.org/10.5194/gmd-13-3267-2020, 2020.

Capiluppi, A., Boldyreff, C., Beecher, K., and Adams, P. J.: Quality Factors and Coding Standards – a Comparison Between Open Source Forges, Electronic Notes in Theoretical Computer Science, 233, 89–103, https://doi.org/10.1016/j.entcs.2009.02.063, 2009.

Carver, J., Heaton, D., Hochstein, L., and Bartlett, R.: Self-Perceptions about Software Engineering: A Survey of Scientists and Engineers, Comput. Sci. Eng., 15, 7–11, https://doi.org/10.1109/MCSE.2013.12, 2013.

Carver, J. C., Weber, N., Ram, K., Gesing, S., and Katz, D. S.: A survey of the state of the practice for research software in the United States, PeerJ Computer Science, 8, e963, https://doi.org/10.7717/peerj-cs.963, 2022.

Chue Hong, N. P., Katz, D. S., Barker, M., Lamprecht, A.-L., Martinez, C., Psomopoulos, F. E., Harrow, J., Castro, L. J., Gruenpeter, M., Martinez, P. A., Honeyman, T., Struck, A., Lee, A., Loewe, A., van Werkhoven, B., Jones, C., Garijo, D., Plomp, E., Genova, F., Shanahan, H., Leng, J., Hellström, M., Sandström, M., Sinha, M., Kuzak, M., Herterich, P., Zhang, Q., Islam, S., Sansone, S.-A., Pollard, T., Atmojo, U. D., Williams, A., Czerniak, A., Niehues, A., Fouilloux, A. C., Desinghu, B., Goble, C., Richard, C., Gray, C., Erdmann, C., Nüst, D., Tartarini, D., Ranguelova, E., Anzt, H., Todorov, I., McNally, J., Moldon, J., Burnett, J., Garrido-Sánchez, J., Belhajjame, K., Sesink, L., Hwang, L., Tovani-Palone, M. R., Wilkinson, M. D., Servillat, M., Liffers, M., Fox, M., Miljković, N., Lynch, N., Martinez Lavanchy, P., Gesing, S., Stevens, S., Martinez Cuesta,

S., Peroni, S., Soiland-Reyes, S., Bakker, T., Rabemanantsoa, T., Sochat, V., Yehudi, Y., and WG, R. F.: FAIR Principles for Research Software (FAIR4RS Principles), https://doi.org/10.15497/RDA00068, 2022.

Colazo, J. and Fang, Y.: Impact of license choice on Open Source Software development activity, Journal of the American Society for Information Science and Technology, 60, 997–1011, https://doi.org/10.1002/asi.21039, 2009.

Döll, P., Sester, M., Feuerhake, U., Frahm, H., Fritzsch, B., Hezel, D. C., Kaus, B., Kolditz, O., Linxweiler, J., Müller Schmied, H., Nyenah, E., Risse, B., Schielein, U., Schlauch, T., Streck, T., and van den Oord, G.: Sustainable research software for high-quality computational research in the Earth System Sciences: Recommendations for universities, funders and the scientific community in Germany, https://doi.org/10.23689/fidgeo-5805, 2023.

Editorial: Does your code stand up to scrutiny?, Nature, 555, 142–142, https://doi.org/10.1038/d41586-018-02741-4, 2018.

Editorial: Giving software its due, Nat Methods, 16, 207–207, https://doi.org/10.1038/s41592-019-0350-x, 2019.

Eeuwijk, S. van, Bakker, T., Cruz, M., Sarkol, V., Vreede, B., Aben, B., Aerts, P., Coen, G., Dijk, B. van, Hinrich, P., Karvovskaya, L., Ruijter, M. K., Koster, J., Maassen, J., Roelofs, M., Rijnders, J., Schroten, A., Sesink, L., Togt, C. van der, Vinju, J., and Willigen, P. de: Research software sustainability in the Netherlands: Current practices and recommendations, Zenodo, https://doi.org/10.5281/zenodo.4543569, 2021.

Fowler, M.: Refactoring, 2nd ed., Addison Wesley, Boston, MA, 2019.

Frieler, K. and Vega, I.: ISIMIP & ISIpedia - Inter-sectoral impact modeling and communication of national impact assessments, 2019.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J.: Design patterns, Addison Wesley, Boston, MA, 1994.

Guaman, D., Delgado, S., and Perez, J.: Classifying Model-View-Controller Software Applications Using Self-Organizing Maps, IEEE Access, 9, 45201–45229, https://doi.org/10.1109/ACCESS.2021.3066348, 2021.

Hannay, J. E., MacLeod, C., Singer, J., Langtangen, H. P., Pfahl, D., and Wilson, G.: How do scientists develop and use scientific software?, in: 2009 ICSE Workshop on Software Engineering for Computational Science and Engineering, 1–8, https://doi.org/10.1109/SECSE.2009.5069155, 2009.

Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M. H., Brett, M., Haldane, A., del Río, J. F., Wiebe, M., Peterson, P., Gérard-Marchant, P., Sheppard, K., Reddy, T., Weckesser, W., Abbasi, H., Gohlke, C., and Oliphant, T. E.: Array programming with NumPy, Nature, 585, 357–362, https://doi.org/10.1038/s41586-020-2649-2, 2020.

He, H.: Understanding Source Code Comments at Large-Scale, in: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, New York, NY, USA, event-place: Tallinn, Estonia, 1217–1219, https://doi.org/10.1145/3338906.3342494, 2019.

Hofmann, H., Wickham, H., and Kafadar, K.: Letter-Value Plots: Boxplots for Large Data, Journal of Computational and Graphical Statistics, 26, 469–477, https://doi.org/10.1080/10618600.2017.1305277, 2017.

Jay, C. and Haines, R.: Reproducible and Sustainable Research Software, in: Web Accessibility: A Foundation for Research, edited by: Yesilada, Y. and Harper, S., Springer, London, 211–221, https://doi.org/10.1007/978-1-4471-7440-0_12, 2019.

Jiménez, R. C., Kuzak, M., Alhamdoosh, M., Barker, M., Batut, B., Borg, M., Capella-Gutierrez, S., Hong, N. C., Cook, M., Corpas, M., Flannery, M., Garcia, L., Gelpí, J. L., Gladman, S., Goble, C., Ferreiro, M. G., Gonzalez-Beltran, A., Griffin, P. C., Grüning, B., Hagberg, J., Holub, P., Hooft, R., Ison, J., Katz, D. S., Leskošek, B., Gómez, F. L., Oliveira, L. J., Mellor, D., Mosbergen, R., Mulder, N., Perez-Riverol, Y., Pergl, R., Pichler, H., Pope, B., Sanz, F., Schneider, M. V., Stodden, V., Suchecki, R., Vařeková, R. S., Talvik, H.-A., Todorov, I., Treloar, A., Tyagi, S., Gompel, M. van, Vaughan, D., Via, A., Wang, X., Watson-Haigh, N. S., and Crouch, S.: Four simple recommendations to encourage best practices in research software, https://doi.org/10.12688/f1000research.11407.1, 13 June 2017.

JuliaReachDevDocs: https://juliareach.github.io/JuliaReachDevDocs/latest/guidelines/, last access: 11 September 2024.

Katz, D. S.: Research Software: Challenges & Actions. The Future of Research Software: International Funders Workshop, Amsterdam, Netherlands., https://doi.org/10.5281/zenodo.7295423, 2022.

Kemp, L., Xu, C., Depledge, J., Ebi, K. L., Gibbins, G., Kohler, T. A., Rockström, J., Scheffer, M., Schellnhuber, H. J., Steffen, W., and Lenton, T. M.: Climate Endgame: Exploring catastrophic climate change scenarios, Proceedings of the National Academy of Sciences, 119, e2108146119, https://doi.org/10.1073/pnas.2108146119, 2022.

Long, J.: Understanding the Role of Core Developers in Open Source Development, Journal of Information, Information Technology, and Organizations (Years 1-3), 1, 075–085, 2006.

McConnell, S.: in: Code Complete, Second Edition, Microsoft Press, USA, 565–596, 2004.

McKiernan, E. C., Barba, L., Bourne, P. E., Carter, C., Chandler, Z., Choudhury, S., Jacobs, S., Katz, D. S., Lieggi, S., Plale, B., and Tananbaum, G.: Policy recommendations to ensure that research software is openly accessible and reusable, PLOS Biology, 21, 1–4, https://doi.org/10.1371/journal.pbio.3002204, 2023.

Merow, C., Boyle, B., Enquist, B. J., Feng, X., Kass, J. M., Maitner, B. S., McGill, B., Owens, H., Park, D. S., Paz, A., Pinilla-Buitrago, G. E., Urban, M. C., Varela, S., and Wilson, A. M.: Better incentives are needed to reward academic software development, Nat Ecol Evol, 7, 626–627, https://doi.org/10.1038/s41559-023-02008-w, 2023.

Molnar, A.-J., Motogna, S., and Vlad, C.: Using static analysis tools to assist student project evaluation, in: Proceedings of the 2nd ACM SIGSOFT International Workshop on Education through Advanced Software Engineering and Artificial Intelligence, ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual USA, 7–12, https://doi.org/10.1145/3412453.3423195, 2020.

Nangia, U. and Katz, D. S.: Track 1 Paper: Surveying the U.S. National Postdoctoral Association Regarding Software Use and Training in Research, https://doi.org/10.6084/m9.figshare.5328442.v3, 2017.

Nüst, D., Sochat, V., Marwick, B., Eglen, S. J., Head, T., Hirst, T., and Evans, B. D.: Ten simple rules for writing Dockerfiles for reproducible data science, PLoS Comput Biol, 16, e1008316, https://doi.org/10.1371/journal.pcbi.1008316, 2020.

Nyenah, E., Reinecke, R., and Döll, P.: Towards a sustainable utilization of the global hydrological research software WaterGAP, pico, https://doi.org/10.5194/egusphere-egu23-4453, 2023.

Obermüller, F., Bloch, L., Greifenstein, L., Heuer, U., and Fraser, G.: Code Perfumes: Reporting Good Code to Encourage Learners, in: The 16th Workshop in Primary and Secondary Computing Education, WiPSCE '21: The 16th Workshop in Primary and Secondary Computing Education, Virtual Event Germany, 1–10, https://doi.org/10.1145/3481312.3481346, 2021.

Plösch, R., Bräuer, J., Körner, C., and Saft, M.: Measuring, Assessing and Improving Software Quality based on Object-Oriented Design Principles, Open Computer Science, 6, 187–207, https://doi.org/10.1515/comp-2016-0016, 2016.

Prinn, R. G.: Development and application of earth system models, Proceedings of the National Academy of Sciences, 110, 3673–3680, https://doi.org/10.1073/pnas.1107470109, 2013.

Rashid, M., Clarke, P. M., and O'Connor, R. V.: A systematic examination of knowledge loss in open source software projects, International Journal of Information Management, 46, 104–123, https://doi.org/10.1016/j.ijinfomgt.2018.11.015, 2019.

Reinecke, R., Trautmann, T., Wagener, T., and Schüler, K.: The critical need to foster computational reproducibility, Environmental Research Letters, 17, https://doi.org/10.1088/1748-9326/ac5cf8, 2022.

Research Software Alliance: Amsterdam Declaration on Funding Research Software Sustainability, , https://doi.org/10.5281/ZENODO.8325436, 2023.

Sachan, R. K., Nigam, A., Singh, A., Singh, S., Choudhary, M., Tiwari, A., and Kushwaha, D. S.: Optimizing Basic COCOMO Model Using Simplified Genetic Algorithm, Procedia Computer Science, 89, 492–498, https://doi.org/10.1016/j.procs.2016.06.107, 2016.

Sarkar, S., Kak, A. C., and Rama, G. M.: Metrics for Measuring the Quality of Modularization of Large-Scale Object-Oriented Software, IIEEE Trans. Software Eng., 34, 700–720, https://doi.org/10.1109/TSE.2008.43, 2008.

Satoh, Y., Yoshimura, K., Pokhrel, Y., Kim, H., Shiogama, H., Yokohata, T., Hanasaki, N., Wada, Y., Burek, P., Byers, E., Schmied, H. M., Gerten, D., Ostberg, S., Gosling, S. N., Boulange, J. E. S., and Oki, T.: The timing of unprecedented hydrological drought under climate change, Nature Communications, 13, https://doi.org/10.1038/s41467-022-30729-2, 2022.

Sauer, I. J., Reese, R., Otto, C., Geiger, T., Willner, S. N., Guillod, B. P., Bresch, D. N., and Frieler, K.: Climate signals in river flood damages emerge under sound regional disaggregation, Nat Commun, 12, 2128, https://doi.org/10.1038/s41467-021-22153-9, 2021.

Schmidhuber, J. and Tubiello, F. N.: Global food security under climate change, Proceedings of the National Academy of Sciences, 104, 19703–19708, https://doi.org/10.1073/pnas.0701976104, 2007.

Simmons, A. J., Barnett, S., Rivera-Villicana, J., Bajaj, A., and Vasa, R.: A large-scale comparative analysis of Coding Standard conformance in Open-Source Data Science projects, in: Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), ESEM '20: ACM / IEEE International Symposium on Empirical Software Engineering and Measurement, Bari Italy, 1–11, https://doi.org/10.1145/3382494.3410680, 2020.

Stacke, T.: Personal communication, 2023.

Stacke, T. and Hagemann, S.: HydroPy (v1.0): a new global hydrology model written in Python, Geoscientific Model Development, 14, 7795–7816, https://doi.org/10.5194/gmd-14-7795-2021, 2021.

Stacke, Tobias and Hagemann, Stefan: Source code for the global hydrological model HydroPy, 2021.

Ståhl, D. and Bosch, J.: Modeling continuous integration practice differences in industry software development, Journal of Systems and Software, 87, 48–59, https://doi.org/10.1016/j.jss.2013.08.032, 2014.

Stamelos, I., Angelis, L., Oikonomou, A., and Bleris, G. L.: Code quality analysis in open source software development, Information Systems Journal, 12, 43–60, https://doi.org/10.1046/j.1365-2575.2002.00117.x, 2002.

Trisovic, A., Lau, M. K., Pasquier, T., and Crosas, M.: A large-scale study on research code quality and execution, Sci Data, 9, 60, https://doi.org/10.1038/s41597-022-01143-6, 2022.

Turk, D., Robert, F., and Rumpe, B.: Assumptions Underlying Agile Software-Development Processes, Journal of Database Management (JDM), 16, 62–87, https://doi.org/10.4018/jdm.2005100104, 2005.

Van Snyder, W.: Scientific Programming in Fortran, Scientific Programming, 15, 3–8, https://doi.org/10.1155/2007/930816, 2007.

Wagener, T., Gleeson, T., Coxon, G., Hartmann, A., Howden, N., Pianosi, F., Rahman, M., Rosolem, R., Stein, L., and Woods, R.: On doing hydrology with dragons: Realizing the value of perceptual models and knowledge accumulation, WIREs Water, 8, e1550, https://doi.org/10.1002/wat2.1550, 2021.

Wan, W., Döll, P., and Zheng, H.: Risk of Climate Change for Hydroelectricity Production in China Is Small but Significant Reductions Cannot Be Precluded for More Than a Third of the Installed Capacity, Water Resources Research, 58, e2022WR032380, https://doi.org/10.1029/2022WR032380, 2022.

Wang, Y., Zheng, B., and Huang, H.: Complying with Coding Standards or Retaining Programming Style: A Quality Outlook at Source Code Level, Journal of Software Engineering and Applications, 1, 88–91, https://doi.org/10.4236/jsea.2008.11013, 2008.

Warszawski, L., Frieler, K., Huber, V., Piontek, F., Serdeczny, O., and Schewe, J.: The Inter-Sectoral Impact Model Intercomparison Project (ISI–MIP): Project framework, Proceedings of the National Academy of Sciences, 111, 3228–3232, https://doi.org/10.1073/pnas.1312330110, 2014.

SLOCCount: https://stuff.mit.edu/iap/debian/solutions/sloccount-2.26/sloccount.html., last access: 4 March 2024.

Wijendra, D. R. and Hewagamage, K. P.: Software Complexity Reduction through the Process Automation in Software Development Life Cycle, in: 2021 Fourth International Conference on Electrical, Computer and Communication Technologies (ICECCT), 2021 Fourth International Conference on Electrical, Computer and Communication Technologies (ICECCT), 1–7, https://doi.org/10.1109/ICECCT52121.2021.9616781, 2021.

Wilson, G., Aruliah, D. A., Brown, C. T., Hong, N. P. C., Davis, M., Guy, R. T., Haddock, S. H. D., Huff, K. D., Mitchell, I. M., Plumbley, M. D., Waugh, B., White, E. P., and Wilson, P.: Best Practices for Scientific Computing, PLOS Biology, 12, e1001745, https://doi.org/10.1371/journal.pbio.1001745, 2014.

Zhou, N., Zhou, H., and Hoppe, D.: Containerisation for High Performance Computing Systems: Survey and Prospects, IIEEE Trans. Software Eng., 49, 2722–2740, https://doi.org/10.1109/TSE.2022.3229221, 2023.