Geosci. Model Dev., 17, 8593–8611, 2024
https://doi.org/10.5194/gmd-17-8593-2024

Geoscientific Model Development
Open Access
EGU

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 and iSchool, University of Illinois Urbana-Champaign, Urbana, IL 61801, USA
[4]Institute of Geography, Johannes Gutenberg University Mainz, 55128 Mainz, Germany

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

Received: 22 May 2024 – Discussion started: 5 June 2024
Revised: 20 September 2024 – Accepted: 14 October 2024 – Published: 5 December 2024

Abstract. Research software for simulating Earth processes enables the estimation of 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 that is, in this sense, non-sustainable. Here we evaluate the sustainability of global-scale impact models across 10 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 an open-source license. Only 16 % provide the software in a 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 % of the GIMs, insufficient modularity in 88 % of the GIMs, 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 PEP8 coding standards, with the rest showing low compliance. To improve the sustainability of GIMs 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 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. The central role of impact models can be seen in model intercomparison efforts of the Inter-Sectoral Impact Model

Intercomparison Project (ISIMIP) (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, and the computational effort needed for simulation, as well as their spatio-temporal resolution and the 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 practices (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, is not easily (re)usable, is difficult to modify and maintain, has scarce internal documentation (code comments) and external documentation (e.g. manuals, guides, and tutorials), and has 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 maintainable, extensible, and 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/, last access: 29 November 2024) 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.

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 and 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 was thus certainly sustained during that period, while at the same time, it does not meet some of the sustainability requirements in 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: findable, accessible, interoperable, and reusable (FAIR) 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 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 following 10 research fields (or impact sectors): agriculture, biomes, fire, fisheries, health, lakes, water (resources), water quality, groundwater, 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/, last access: 2 December 2024), which provides essential information, such as model ownership, names, 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 models 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 Google's 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 papers. However, it is 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 could not 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 best practice in software engineering and four indicators of source code quality (Table 1).

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, and tutorials that explain the usage and functionality of the software, as well as 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 file ISIMIP_models.xlsx in the Supplement). Therefore, in addition to the reference model paper, we checked for available manuals, guides, README files, and tutorials. We regard 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 SourceForge. The host name 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 "SourceForge". 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 an in-depth 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 open-source license. We determined the existence of open-source licenses by checking license files within repositories or official websites against licenses approved

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 life cycle and enables collaboration between developers and 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	Prevents single points of failure in the development process and makes software development and 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. PEP8)	Improves code quality and 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 reusability (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).

by the Open Source Initiative (OSI) (https://opensource.org/licenses, last access: 2 December 2024). Specifically, we looked for licenses that conform to the definition of open source, 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 allow for minimal restrictions on how the software can be used (e.g. providing attribution), and copyleft licenses, like GPL, which 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.

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 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 project's codebase within the period of 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/, last access: 2 December 2024) and Apptainer (https://apptainer.org/, last access: 2 December 2024). There are also cloud-supported container solutions, such as Binder (https://mybinder.org/, last access: 2 December 2024), with the capacity to execute a model with 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 an "Apptainer definition file (.def file)". In addition, we also searched through source code repositories to identify the previously

stated images or image-building files. Lastly, we utilized 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 analyse 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 assigns a maximum score of 10 as perfect compliance but has no lower bound (Molnar et al., 2020). We consider scores below 6 as indicative of weak compliance as the code contains 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

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

Here, the total lines of code (TLOC) metric includes both comments and source lines of code (SLOC) (SLOCCount, 2024). The SLOC metric is defined as the physical non-blank, non-comment lines in a source file. Arafat and Riehle (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).

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 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 of research software sustainability. Modular programming is an approach where source codes are organized 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 programme-specific tools. For instance, in Python, a module that contains a significantly higher TLOC metric than usual (here a TLOC value of over 1000) 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 1000 lines to prevent potential performance issues such as crashes or slow programme execution with some integrated development environments (IDEs) (Fowler, 2019; McConnell, 2004). IDEs are software applications that provide tools like code editors and 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 a TLOC per file value of 10–1000 indicates adequate modularity. We also varied the upper bounds of the total lines of code to 5000 and 500 to investigate how modularity changes across models and sectors.

2.3 Source code counter

To count the SLOC, comment lines, and TLOC of computational models, the counting tool developed by Ben Boyter (https://github.com/boyter/scc, last access: 2 Decem-

ber 2024) was used (Boyter, 2024). This tool builds on the industrial standard source code counter tool called Source Lines of Code Count (SLOCCount) (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 value of 35 000) to optimize the parameters of the COCOMO regression model (Sachan et al., 2016). Effort in person months is estimated following Eq. (2):

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

where total lines of code are expressed in a TLOC value of 1000 (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 programming 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, a large user community, and a rich ecosystem of packages, including those supporting parallel computing. R and C++ follow with five models and C with four (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, 2021a).

We find that 24 (75 %) of the readily accessible 32 GIMs are 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).

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., 2023). This might be the case for other models as well.

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 web pages for documentation, and 5 included comprehensive manuals (see file ISIMIP_models.xlsx in the Supplement). 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 lacks documentation or does not have publicly available documentation (Table 2).

Version control.
We find that 81 % (26 out of 32) of GIMs use 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.

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 most 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 use a license which is not OSI-approved (one GIM each with a creative commons license and user agreement) (Fig. 2). This ambiguity in or absence of licensing details can deter potential users and contributors, as it raises uncertainties about the permissions and restrictions associated with the software.

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 those without it 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, respectively (Fig. 3). These values correlate with the size of GIM source codes, as evidenced by TLOC (282 722 for ParFlow, 33 236 for CWatM, 136 002 for LPJmL, and 29 477 for GOTM). However, models such as WAYS, VIC, BioScen1.5-MEM, and CGMS-WOFOST

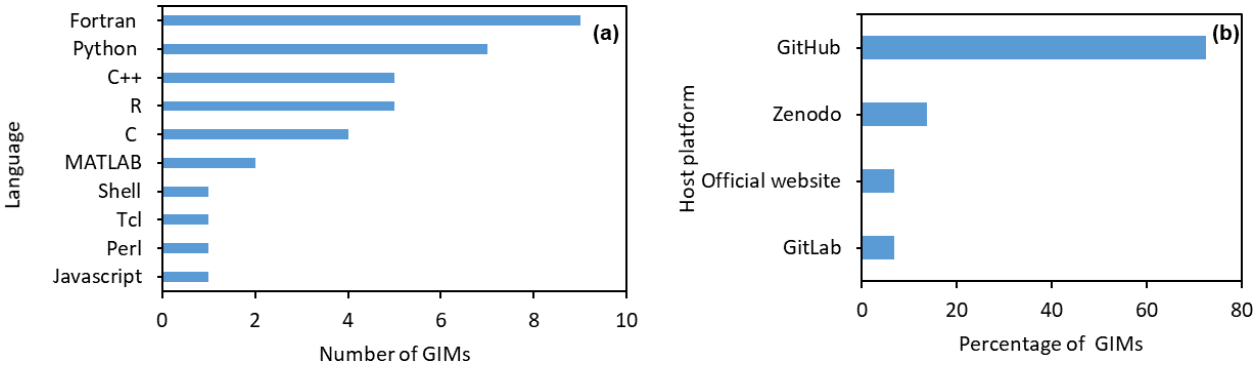

Figure 1. Programming languages for model development and model accessibility. **(a)** Bar plots showing programming languages used for developing 32 global impact models. **(b)** Bar plot showing open-source hosting platforms where 32 global impact models were accessed.

Table 2. Availability of documentation, version control, open-source licenses, test suites, and containers for 32 global impact models across 10 sectors in Earth system science. "+", "–", "not valid", and "no info" represent the availability of information, the unavailability of information, a license that is not OSI-approved, and the 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	G^3M	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	BioScen1.5-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

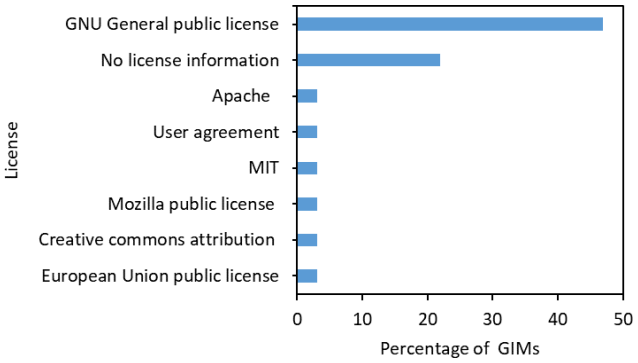

Figure 2. License distribution for 32 global impact models across 10 sectors. Eight (25 %) GIMs lack license information, and two (6 %) GIMs have licenses that are not OSI-approved.

had no active developers during the considered period of 2020 to 2024 (Fig. 3).

Containerization.
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.

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 the challenges described in Sect. 2.2. Among the 10 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.

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 commenting practice is generally low across the studied research fields.

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 a TLOC per file value of between 10 and 1000 (Fig. 6). The remaining 28 GIMs either had at least one file with a TLOC value exceeding 1000, which likely could be divided into smaller modules with distinct functionality, or had at least one file with a TLOC value of less than 10, 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 a TLOC per file value of 5000 and 500 with the same lower limit of a TLOC value of 10. Nine simulation models (LPJmL, MC2-USFS-r87g5c1, EcoTroph, ZooMSS, BioScen1.5-SDM, BioScen1.5-MEM, H08, HydroPy, and DynQual) meet the 5000-line criterion, and two models (EcoTroph, ZooMSS) meet 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 an SLOC value of 1000 instead of a TLOC value of 1000 with an SLOC value of 10. Three GIMs (ZooMSS, BioScen1.5-SDM, and HydroPy) meet the criterion of an SLOC value of 10–1000 (see Fig. S1 in the Supplement).

3.3 Cost of GIM software development

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 Sect. 2.4) in a scenario of "What if we hired a commercial software company to develop the source code of the global impact models?" This cost estimate does not include the costs of developing the sci-

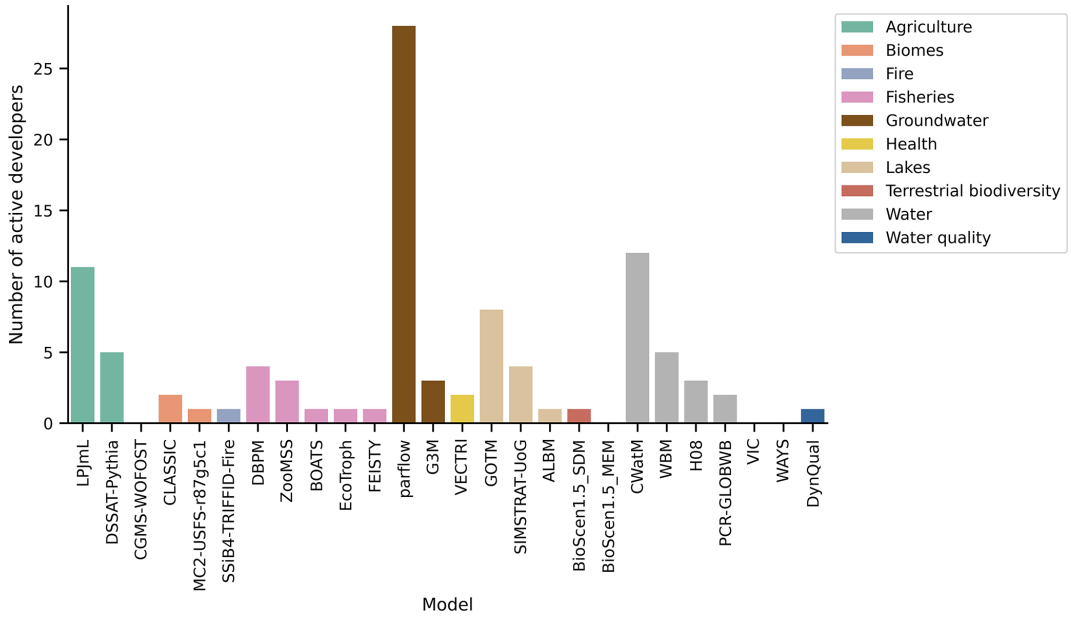

Figure 3. Number of active developers within 5 years (2020–2024) for 26 global impact models across 10 sectors. The results for the six remaining GIMs could not be measured since version control information could not be found. A value of zero means there were no active developers within the 5-year period. The models are sorted within each sector by the number of active developers.

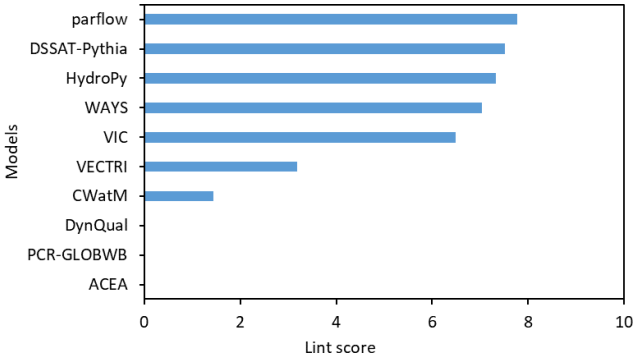

Figure 4. Lint scores of GIMs containing Python code.

ence (e.g. concepts, algorithms, and input data) or the costs of documenting, running, and maintaining the software; it only includes the implementation of code. We assume that the COCOMO model is transferable to research software as the NASA projects used in the cost model contain software that is similar to research software. As the TLOC of the impact model codes range from a TLOC value of 262 to 500 000 (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 $+0.1$ (Fig. S2 in the Supplement).

The results suggest that these complex research software programmes 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 in developing and maintaining sustainable research software in various domains (Carver et al., 2013, 2022; van 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).

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 GIMs, such as HydroPy, (Stacke and Hagemann, 2021a, b) have been reprogrammed, while others (e.g. WaterGAP2.2e, Nyenah et al., 2023) are in the process of being reprogrammed. We compared the global hydrological legacy 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 lowers 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 (Edi-

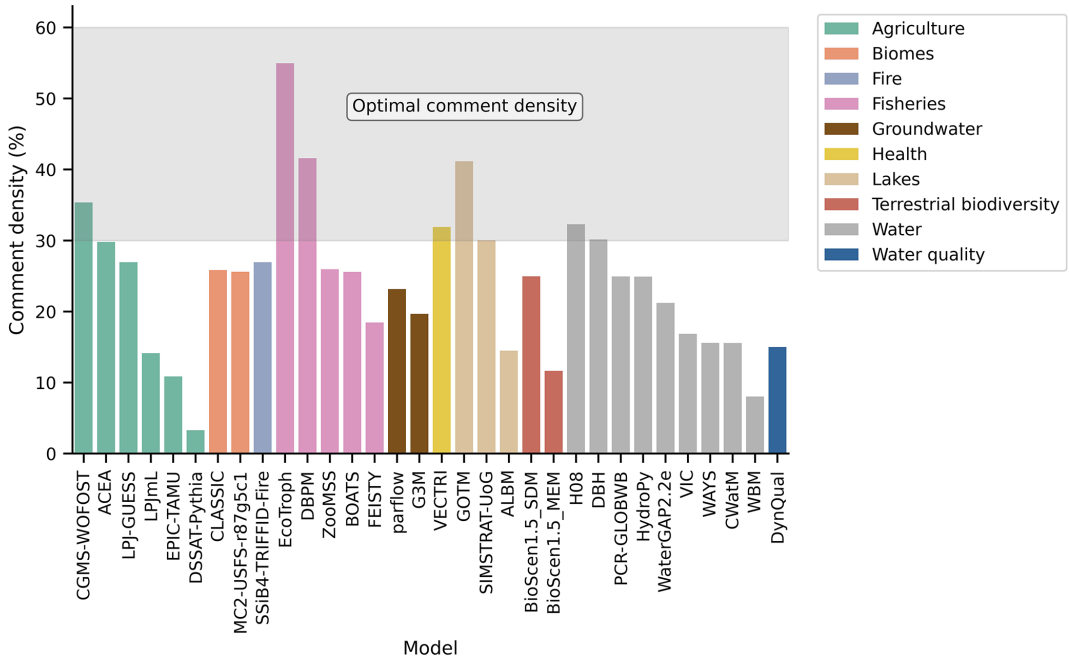

Figure 5. Comment density per model across 10 sectors. The grey zone denotes the optimal comment density (Arafat and Riehle, 2009; He, 2019). Models are sorted within each sector by decreasing comment density.

torial, 2019). This research software refers users to its associated publication for a detailed model description, as well as to instructions on Zenodo for setting up and running HydroPy. A software testing suite and container have not been made available yet.

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" (Tobias Stacke, personal communication, 2023). MPI-HM has more comments (49 %, Fig. 8b) because of its legacy Fortran code that limits variable names to a maximum length of eight 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 the question "Is the comment density threshold metric still valid if a code is highly readable and comprehensive?" 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.

4 Limitations

Our study has limitations in the following regards. In the interest of timely analysis, we did not contact the developers of models that were not readily available. This means that older software, particularly 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 biome 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.

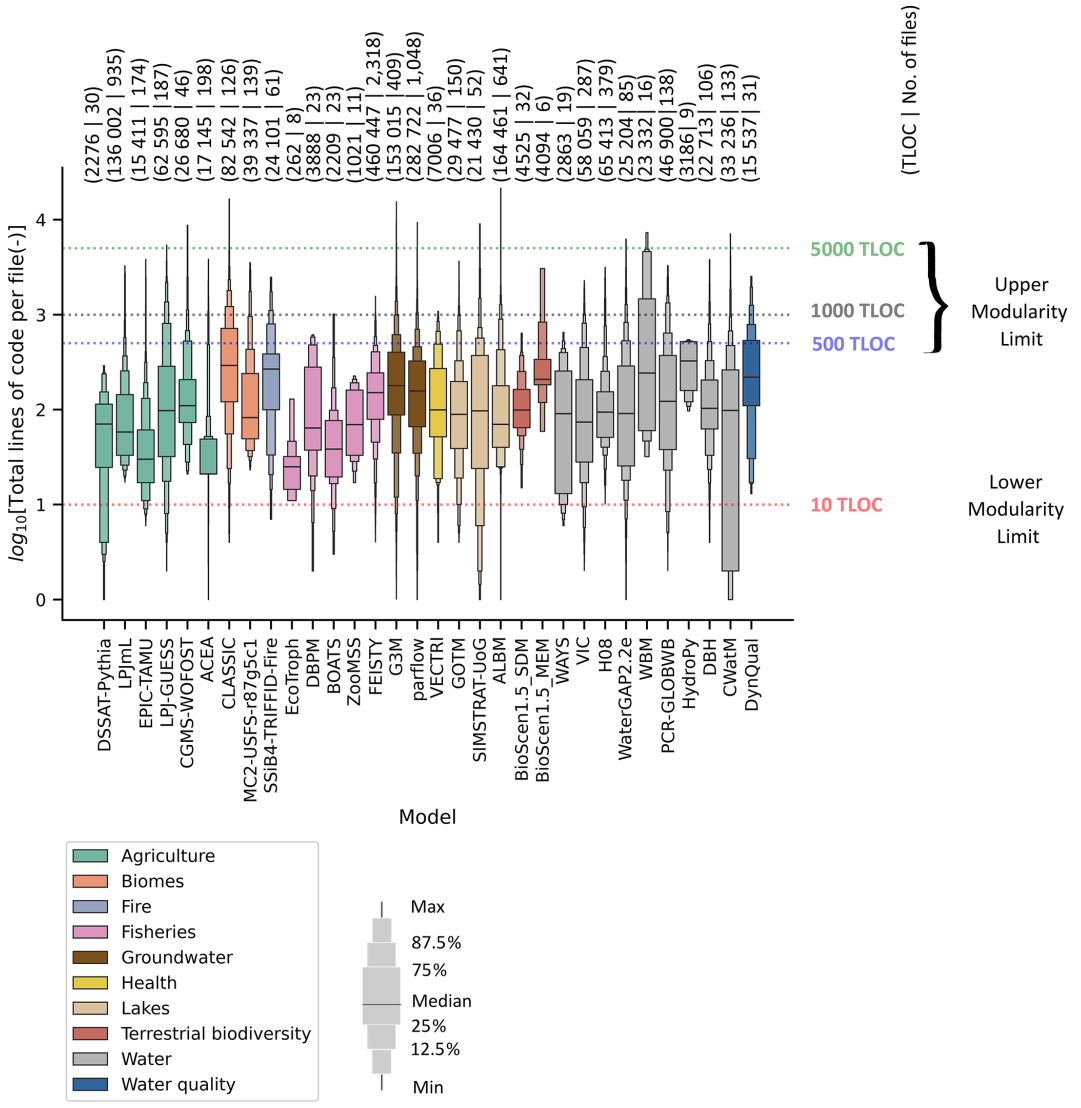

Figure 6. Letter value plot (Hofmann et al., 2017) of the total lines of code (TLOC) per file (logarithmic scale) of 32 global impact models across 10 sectors. The dotted blue, black, and green lines show upper modularity limits, and the dotted red line shows the lower limit. The values (x | y) in the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

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 and date of last update or version), number of publications and citations, coupling and cohesion, information content of comments, software adaptability to user requirements, 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 may 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 are 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.

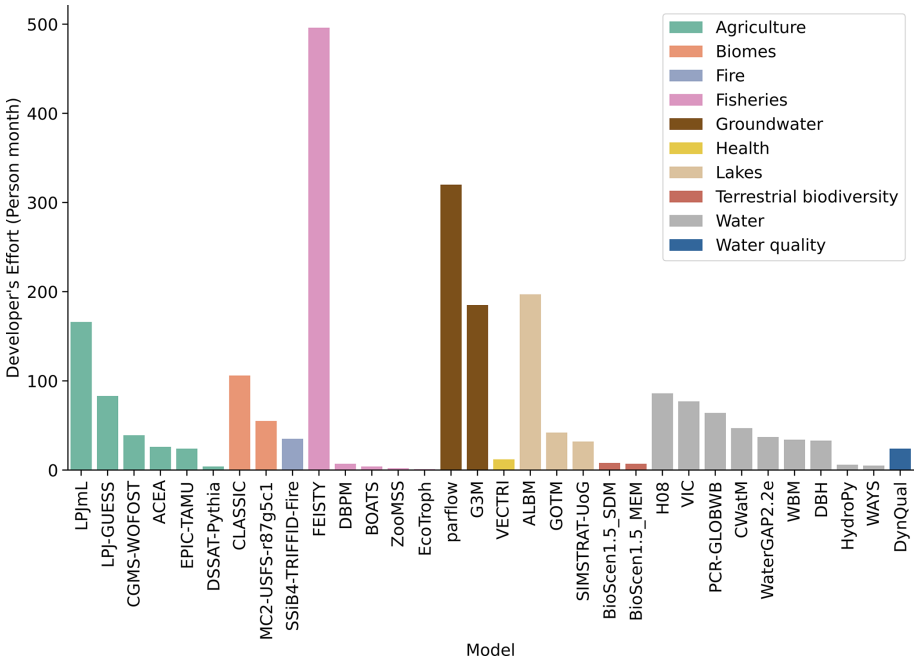

Figure 7. Effort estimates of 32 global impact models across 10 sectors. Models are sorted within each sector by decreasing the amount of the developer's effort.

In addition to the previously discussed limitations, the indicators analysed in this study are quantitative metrics that can be measured. Factors such as the 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.

Moreover, 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) do 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 have 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 programmes 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; van Eeuwijk et al., 2021; Merow et al., 2023). This software should be treated as a citable academic contribution and included, for instance, in PhD theses (Merow et al., 2023).

To assess the current state of these practices in Earth 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 im-

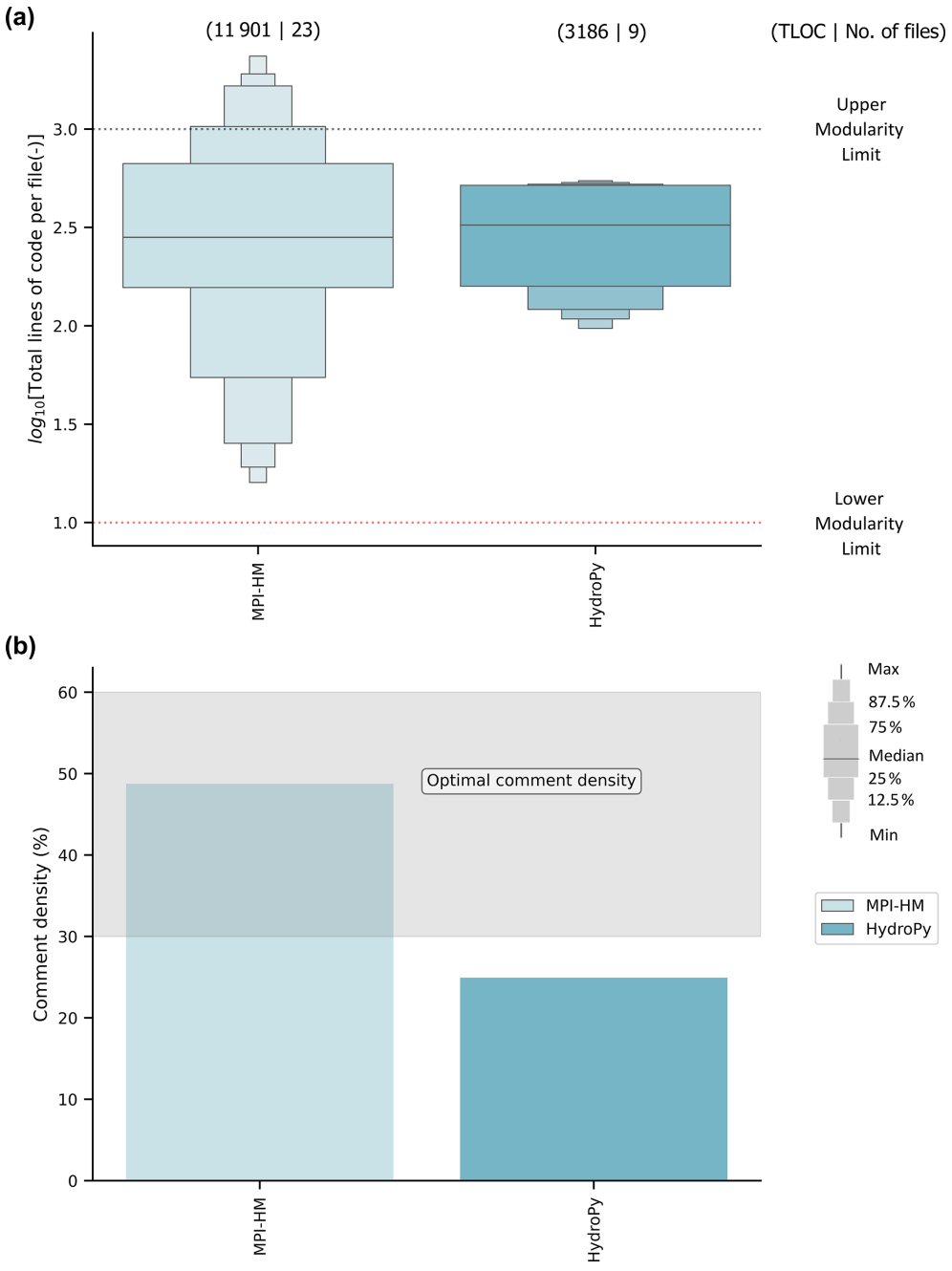

Figure 8. Modularity and commenting practice of a legacy (MPI-HM) and reprogrammed (HydroPy) global simulation model. **(a)** Letter value plot of the total lines of code per file (logarithmic scale) of each model. The dotted black (red) line shows the upper (lower) modularity limit defined as the maximum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

proved software development practices in Earth 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 (summarized in Fig. 9):

– *Choose project management practices that align with your institutional environment, culture, and project requirements.* 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 also

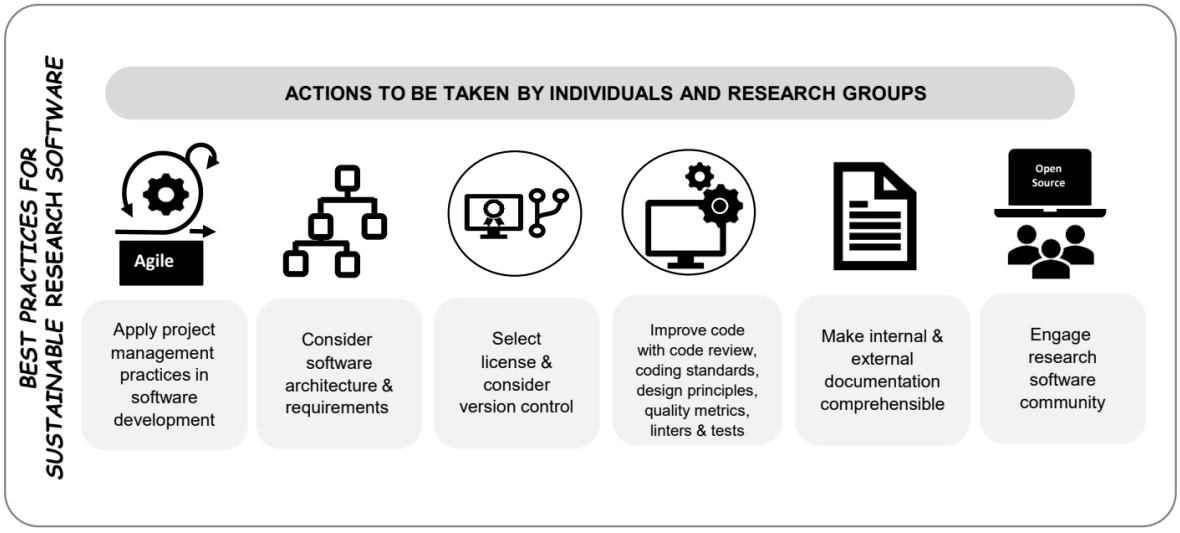

Figure 9. Actionable best practices for sustainable research software. The image summarizes the actions that modelling communities and individual developers should take, such as following project management practices, coding standards, reviews, documentation, and community engagement strategies. These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

help identify and mitigate risks, manage 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 the incorporation of changing research questions and hence software modifications or extensions, improving responsiveness to new developments (Turk et al., 2005).

– *Consider software architecture (organization 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–view–controller, Guaman et al., 2021) and user requirements helps to design a software system that has a clear and coherent structure, a 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, and protect your intellectual property rights (Anzt et al., 2021; Carver et al., 2022). Accessible software is crucial for reducing 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 (Jiménez et al., 2017). Platforms like GitHub and GitLab are commonly used for this purpose. However, it is 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 promote adherence 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 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 materials 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 to get feedback, support, and advice; promote collaboration and contributions; and obtain 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, developers can use GitHub Actions to automate various tasks like running test suites, generating documentation, ensuring adherence to coding standards, and managing dependencies.

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 of improvement remain, 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, and academic and research organizations, to facilitate the development and maintenance of sustainable research software. We also propose using 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 and data availability. The Python scripts utilized for analysis can be accessed at https://doi.org/10.5281/zenodo.10245636 (Nyenah et al., 2024). Additionally, the line-counting tool developed by Ben Boyter is available through the GitHub repository: https://github.com/boyter/scc (Boyter, 2024).

The results obtained from the line count analysis are accessible at https://doi.org/10.5281/zenodo.10245636 (Nyenah et al., 2024).

For the convenient download of the global impact models, links to the 32 global impact models, along with their respective dates of access, can be found in an Excel sheet named "ISIMIP_models.xlsx.", present in the Zenodo repository.

Supplement. The supplement related to this article is available online at: https://doi.org/10.5194/gmd-17-8593-2024-supplement.

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

Competing interests. The contact author has declared that none of the authors has any competing interests.

Acknowledgements. Emmanuel Nyenah, Robert Reinecke, and Petra Döll acknowledge support from the Deutsche Forschungsgemeinschaft (DFG) (grant no. 443183317).

Financial support. This research has been supported by the Deutsche Forschungsgemeinschaft (grant no. 443183317).

This open-access publication was funded by Goethe University Frankfurt.

Review statement. This paper was edited by Fabien Maussion and reviewed by Rolf Hut and Facundo Sapienza.

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, Comput. Sci. Eng., 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, Geosci. Model Dev., 8, 1221–1232, https://doi.org/10.5194/gmd-8-1221-2015, 2015.

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, 295, 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, 16–24 May 2009, Vancouver, BC, Canada, 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, Hydrol. Earth Syst. Sci., 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, Sci. 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, P. Natl. Acad. Sci. USA, 119, e2202112119, https://doi.org/10.1073/pnas.2202112119, 2022.

Boehm, B. W.: Software engineering economics, Prentice-Hall, Englewood Cliffs, NJ, 57–96, ISBN 0138221227, 1981.

Boyter, B.: boyter/scc, GitHub [code], 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, Geosci. Model Dev., 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 RDA FAIR4RS WG: 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, J. Am. Soc. Inf. Sci. Tec., 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: Recom-

mendations 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.

Fowler, M.: Refactoring, 2nd edn., Addison Wesley, Boston, MA, ISBN 0134757599, 2019.

Frieler, K. and Vega, I.: ISIMIP & ISIpedia – Inter-sectoral impact modeling and communication of national impact assessments, Bonn Climate Change Conference, 19 June 2019, session SBSTA 50, https://unfccc.int/documents/197148 (2 December 2024), 2019.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J.: Design patterns, Addison Wesley, Boston, MA, ISBN 0201633612, 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, 23 May 2009, Vancouver, BC, Canada, 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, 26–30 August 2019, 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, J. Comput. Graph. Stat., 26, 469–477, https://doi.org/10.1080/10618600.2017.1305277, 2017.

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

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., van Gompel, M., 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, the 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, P. Natl. Acad. Sci. USA, 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.: A Practical Handbook of Software Construction, in: Code Complete, 2nd edn., Microsoft Press, USA, 565–596, ISBN 0735619670, 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 Biol., 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, 9 November 2020, USA, 7–12, https://doi.org/10.1145/3412453.3423195, 2020.

Müller Schmied, H., Trautmann, T., Ackermann, S., Cáceres, D., Flörke, M., Gerdener, H., Kynast, E., Peiris, T. A., Schiebener, L., Schumacher, M., and Döll, P.: The global water resources and use model WaterGAP v2.2e: description and evaluation of modifications and new features, Geosci. Model Dev. Discuss. [preprint], https://doi.org/10.5194/gmd-2023-213, in review, 2023.

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, EGU General Assembly 2023, Vienna, Austria, 24–28 Apr 2023, EGU23-4453, https://doi.org/10.5194/egusphere-egu23-4453, 2023.

Nyenah, E., Döll, P., Katz, D. S., and Reinecke, R.: Software sustainability of global impact models (Dataset and analysis script), Zenodo [data set], https://doi.org/10.5281/zenodo.10245636, 2024.

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, 18–20 October 2021, 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, P. Natl. Acad. Sci. USA, 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, Int. J. Inform. Manage., 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, Environ. Res. Lett., 17, 4, 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 Comput. Sci., 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, IEEE T. 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, Nat. Commun., 13, 3287, 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, P. Natl. Acad. Sci. USA, 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, 5–9 October 2020, Bari, Italy, 1–11, https://doi.org/10.1145/3382494.3410680, 2020.

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

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

Stacke, T. and Hagemann, S.: Source code for the global hydrological model HydroPy, Zenodo [code], https://doi.org/10.5281/zenodo.4541381, 2021b.

Ståhl, D. and Bosch, J.: Modeling continuous integration practice differences in industry software development, J. Syst. 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, Inform. Syst. J., 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, J. Database Manage., 16, 62–87, https://doi.org/10.4018/jdm.2005100104, 2005.

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

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 Resour. Res., 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, P. Natl. Acad. Sci. USA, 111, 3228–3232, https://doi.org/10.1073/pnas.1312330110, 2014.

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), 15–17 September 2021, Erode, India, 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 Biol., 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, IEEE T. Software Eng., 49, 2722–2740, https://doi.org/10.1109/TSE.2022.3229221, 2023.