Dear Reviewer,

We sincerely appreciate your prompt and insightful review of our manuscript. Your valuable comments and suggestions have significantly improved the quality of our manuscript. Below, we address each of your points in detail and outline the corresponding changes made to the manuscript. For clarity, your comments are highlighted in blue, our responses are in black, and any newly added text appears in *italics*. All sections and line numbers refer to the revised version of the manuscript.

Reply to Reviewer 2

This paper presents the process and evaluation of re-programming of the legacy WaterGAP model into the new Python platform. While this study is of interest to the hydrological modeling community, the presentation of this manuscript needs to be substantively revised to highlight the scientific merit of the current study. In its current structure, I am afraid the manuscript does not meet the standard of scientific publication and cannot be accepted.

Thank you for the valuable feedback. The unclear objective and scope of the manuscript as well as its unsuitable structure were also mentioned by Reviewers 1 and 3. This is why we improve the manuscript in various aspects:

1) We have revised our abstract and introduction to more clearly communicate the manuscript's scope. The focus is not on presenting a new model version, with improved model output, but on presenting how the WaterGAP model was implemented in a new software that follows the principles of sustainable research software, including FAIR4RS. We are also scientifically investigating the value of reprogramming the scientific software by addressing the research question: "How can the software sustainability of a legacy scientific software be improved by reprogramming it?"
2) We revised the methods section to include a table that clearly explains the indicators used for evaluating software sustainability and FAIR principles.
3) We added a new "Lessons Learned" section based on the transition from C/C++ to Python, replacing the previous discussion section. Along with section 3 on software sustainability and FAIR principles, and section 4 on the reprogramming process, this strengthens the manuscript's focus on reporting software sustainability through reprogramming.

The abstract now reads (Lines 11-31):

"**Abstract**

Global hydrological models (GHMs) improve our understanding of water flows and storage on the continents and have undergone significant advancements in process representation over the past four decades. However, as research questions and GHMs become increasingly complex, maintaining and enhancing existing model codes efficiently has become challenging. Issues such as non-modular design, inconsistent variable naming, insufficient documentation, lack of automated software testing suites, and containerization hinder the sustainability of GHM research software as well as the reproducibility of study results obtained with the help of GHMs. Although some GHMs have been reprogrammed to address these challenges, existing literature focuses on evaluating the quality of model output rather than the quality of the reprogrammed software. To address this research gap and guide other researchers who wish to implement their existing models as sustainable research software, we describe *in detail how the most recent version of the GHM WaterGAP was reprogrammed. The reprogramming success is evaluated against numerous software sustainability criteria and the principles of findability, accessibility, interoperability, and reusability for research software (FAIR4RS), given that the objective of reprogramming was to enhance software sustainability and thus reproducibility of research results, as opposed to improving model output. Following an agile project management approach, WaterGAP was rewritten from scratch in Python with a modular Model-View-Controller architecture. Due to the switch from C/C++ in the legacy code to Python, execution time doubled. Our evaluation indicates that the reprogramming substantially improved the software's usability, maintainability, and extensibility, making the reprogrammed WaterGAP software much more sustainable than its predecessor. The reprogrammed WaterGAP software can be easily understood, applied, and enhanced by novice and experienced modellers and is suited for collaborative code development across diverse teams and locations, fostering the establishment of a community GHM. We outline six lessons learned from the reprogramming process concerning the sustainability-runtime trade-off, the applicability of the agile approach, software design patterns, variable naming, external documentation, and automation."*

The section of the introduction on the research objectives now reads (Section 1, Lines 75-89):

"To address this research gap and support the reprogramming of other legacy software, this paper provides a detailed account of the reprogramming process of GHM WaterGAP (Döll et al., 2003; Müller Schmied et al., 2024) and the characteristics of the new software. Reprogramming aimed to enhance the software's sustainability for long-term research use by a broad community and to increase the reproducibility of the computational research performed with this model. The success of the reprogramming was assessed by comparing the legacy code to the reprogrammed version according to numerous specific sustainability criteria and FAIR4RS principles. It is important to note that our goal in reprogramming WaterGAP was not to improve the model output; the reprogrammed software was to result in the same model output as the latest WaterGAP version 2.2e (Müller Schmied et al., 2024).

The paper is structured as follows: Section 2 introduces the WaterGAP model and the legacy software. Sustainability criteria for research software and methods relevant to this study are presented in Section 3. After describing the reprogramming process in Section 4, we present the architecture and new features of the reprogrammed software in Section 5. In Section 6, we evaluate the new WaterGAP software against selected sustainability criteria and the FAIR4RS principles. We also demonstrate that the reprogrammed and legacy software yield very similar model outputs (Section 7) and share lessons learned for others undertaking similar efforts (Section 8). Our conclusions follow in Section 9."

We also added a new table to the method section that more clearly communicates the metrics we utilized to assess the software sustainability of the reprogrammed and legacy research software (Section 3, Lines 148-150):

"Table 1: Sustainability indicators used for the assessment of the legacy and reprogrammed research software.

No	Indicators	Description
Best practices in software engineering		
1	External documentation	Effective use and ease of software maintainability rely on clear and extensive external documentation (Nyenah et al., 2024;

		Wilson et al., 2014). We evaluate the availability and extensiveness of external documentation by analyzing the following components: installation guide, tutorials, user guide, reference guide (in-depth descriptions of the model processes and the governing equations), glossary, contributor guide, and frequently asked questions (FAQs).
2	Version control and automation.	Version control facilitates change tracking and supports collaboration (Wilson et al., 2014). We evaluate the use of version control considering the choice between public and private repositories, which significantly affects the repository's transparency and accessibility. We also checked the automation practices, focusing on automated testing, linting, and documentation to ensure consistent quality and maintainability.
3	Use of an open-source license	We determine the presence of open-source licenses by reviewing license files within repositories and comparing them with licenses approved by the Open Source Initiative (OSI) (https://opensource.org/licenses) (Nyenah et al., 2024).
4	Number of active developers	This indicates the capacity for ongoing software development and maintenance (Nyenah et al., 2024). We measured this by counting individuals who made commits to the codebase of the legacy and the reprogrammed code within the past two years (2023–2024).
5	Containerization	Containerization packages software with its full runtime environment, ensuring consistent execution across different systems (Nüst et al., 2020). This helps overcome reproducibility issues caused by variations in operating systems or dependencies. We simply check whether a containerization solution is provided.
Source code quality		
6	Public availability of an (automated) testing suite	We adopted the approach proposed by Nyenah et al. (2024), in using the public availability of an (automated) testing suite as a proxy for the ability to test software functionality. While test coverage is the ideal metric, current coverage tools do not support Python functions with Numba decorators, which

		compile Python functions into machine code for performance (GitHub issues, 2025; Lam et al., 2015; Stack Overflow, 2025).
7	Compliance with coding standards	Coding standards are industry best practices that guide software development for consistency and quality (Wang et al., 2008). To assess compliance, we used CLion static analysis for the legacy C/C++ code, which flags issues (including errors, typos, and warnings) based on the C/C++ Core Guidelines but does not provide a score to interpret results. A higher issue count generally indicates lower reliability or maintainability. For the reprogrammed code, we used Pylint to check compliance with PEP-8 conventions. Pylint assigns a score up to 10 for perfect compliance, with no lower bound (Molnar et al., 2020; Nyenah et al., 2024).
8	Comment density	We compute comment density as the ratio of the number of lines of comments to the total lines of code (TLOC). TLOC refers to the sum of source lines of code (SLOC) and comment lines. SLOC, in turn, represents the non-blank, non-comment lines within a source file. We regard a comment density of 30% to 60% as optimal (Arafat and Riehle, 2009; He, 2019; Nyenah et al., 2024).
9	Modularity	We evaluate the modularity of the software by the TLOC per file metric, with an ideal range of 10 to 1,000 TLOC per file (Nyenah et al., 2024). This metric reflects the organization of source codes into manageable modules, each focusing on a specific functionality. Modules within this range are typically easier to read, modify, and reuse.

"

The new lesson learned section (Section 8, Lines 497 - 565) now reads:

"***8 Lesson learned***
When reprogramming WaterGAP, we made six key observations that we hope can guide others in their efforts to improve the sustainability of their research software.

8.1 Improved readability, maintainability, and adaptability may negatively impact model runtime
Considering sustainable research software indicators and the FAIR4RS principles in the reprogramming of the legacy code has enhanced the software quality, extensibility, reproducibility, and long-term

sustainability of WaterGAP. Unfortunately, the transition from C/C++ in the legacy software to Python, an interpreted language, has approximately doubled the WGHM runtime. This is to be expected as numerical computations in Python can be 3-10 times slower compared to C/C++ (Cai et al., 2005). The average runtime for a standard run on an AMD EPYC 7543 processor with 3.7 GHz is about 7-8 minutes per simulated year for the reprogrammed software, compared to 3-4 minutes for the legacy software. Considering that this may lead to critical run time-related constraints for model calibration and ensemble methods, e.g., used for sensitivity analysis and ensemble forecasts, is the choice of Python justifiable?

To reach this runtime, we already utilized the optimization library Numba, which compiles parts of the Python code. Python is generally slower in terms of runtime performance compared to C/C++ since it uses interpretation instead of compilation (Cai et al., 2005). Compiled code is translated into machine code by a compiler before execution, resulting in a standalone executable file that can be run directly by the processor. On the other hand, interpreted code is executed line by line by an interpreter during runtime, meaning the code must be interpreted every time it is run. Compiled code generally executes faster but often requires a separate compilation step and may be less portable. In contrast, interpreted code is typically more portable but executes more slowly. The pure Python implementation of the GHM HydroPy model is three times slower than the version with a routing scheme written in Fortran (Stacke and Hagemann, 2021).

However, Python generally produces more readable, less error-prone, and more maintainable code than C++, primarily due to its simpler syntax, dynamic typing, automatic memory management, and higher-level abstractions (Balreira et al., 2023; Johnson, 2025; Prechelt, 2000). These features reduce the likelihood of errors and allow developers to express complex ideas more concisely. Python's extensive standard library and ecosystem further enhance maintainability by reducing the need for custom code. In contrast, C++'s more complex syntax and manual memory management can lead to more errors and harder-to-maintain code. Most scientists lack the necessary skills to produce high-quality C++ code and are unlikely to follow any best practices (Reinecke et al., 2022). We believe that the benefits of Python regarding code quality outweigh the runtime increase. The switch from C/C++ to Python makes it easier for scientists, particularly those with restricted programming experience, to understand, modify, extend, and maintain a complex model. Slow code can always be made fast with better hardware, but hardware cannot fix bad code and unsustainable software.

8.2 Implementing an agile process benefits reprogramming also in an academic setting

The agile development process, along with the use of user stories, was essential to our reprogramming effort, enabling iterative improvements through continuous feedback. Agile principles offer significant benefits in academic software development. Specifically, Agile supports flexibility in incorporating evolving research questions and enables effective progress tracking. Tools such as task boards and backlogs provide transparency and help manage workflows efficiently. This is particularly important in academic settings where timelines are often constrained and team composition can change frequently, such as in PhD and Postdoc projects. Agile's emphasis on regular communication helps align the efforts of diverse contributors, including students, researchers, and supervisors (the "project owners"), ensuring everyone stays informed and coordinated throughout the project. User stories helped ensure that the software features matched the scientific requirements of WaterGAP. However, estimating the time needed to complete user stories was difficult, and coordinating an agile process with only a few developers in an academic setting was somewhat challenging. Despite this, we recommend this approach for other reprogramming projects as it supports timely and user-focused development and helps the team stay updated on progress.

*8.3 **Carefully applying established software design patterns throughout the process, ideally with expert input, yields highly modular software***
Defining software architecture and its modular design is an iterative process that benefits greatly from the input of software experts. Architectural decisions play a critical role in determining how easily a model can be extended or modified without affecting other software components. For example, implementing each storage compartment as an independent Python module enabled targeted test development and comprehensive testing before integration. Guidance on software design patterns can be found, for example, in Gamma et al. (1994) (Gamma et al., 1994). A modular design also leads to improved readability, as single components of a project (e.g., code files) are more concise in their purpose.

8.4 Consistent variable naming is paramount for code readability and maintainability
Establishing meaningful and consistent variable names is also an iterative process that requires collaborative effort among developers and domain experts. Clear and logical naming significantly enhances code readability and maintainability. Importantly, naming conventions need to be documented to guide future model development.

8.5 Documentation should be written in parallel with code development
We strongly recommend writing model documentation alongside code development rather than leaving it until the end. This approach helps to capture critical assumptions, such as those embedded in algorithms, while they are still fresh in the developers' minds. Peer review of documentation improves its quality and clarity.

8.6 Automation is key to ensuring efficient development and high software quality.
Automating the generation of the documentation reduces manual work and helps to keep it up to date. Automating linting and testing ensures that the code functions correctly, without the need for constant manual checks ”

Specific comments

1. My major concern of this manuscript is its structure and organization. The manuscript contains sections with too many (10) main sections and levels of subsections. In addition, some subsections contain overlapping information (across different main sections). This tedious structure makes the general reading of the manuscript very difficult. In addition, Sections 5.1 and 5.2 contain subsections with identical titles (Architecture and new features), which should be avoided or simply merged together to reduce the level of subsections. In particular, Section 5.2.1 contains a single sentence with a reference to the Supplement and a GitHub link, which is completely unnecessary to make a separate subsection.

We appreciate the reviewer’s feedback regarding the structure and organization of our manuscript. We have reduced the number of subsections and revised the manuscript throughout to convey a more concise message and improve readability. Furthermore, we moved the user survey section to the supplementary because the scope and depth of the survey do not justify a standalone section in the manuscript, as pointed out by Reviewer 1. This change reduces the number of main sections from ten to nine. We also reviewed the manuscript carefully to eliminate overlapping content. For example, Section 3.1, which initially described the software evaluation against sustainability criteria and FAIR principles, has been condensed into a concise table (see Table 1) that clearly summarizes how each criterion was assessed. This revision prevents redundancy with Section 4.3, which focuses on how these criteria were applied during software development. We also revised Section 5 by merging the previously separate subsections 5.1 and 5.2 (see line 359-427).

2. In general, the authors should either avoid using abbreviations in section titles, or spell out their full names. Examples including FAIR4RS (Section 3.1.3 and 6 title), WGHM (5.1), GWSWUSE (5.2).

We have replaced all abbreviations in the revised manuscript with their full names.

3. Section 3.1.1 “Indicators for best practices in software engineering”: this section seemingly contains information that overlaps with Section 4.3.1 “Best practices for code development”, such as document and version control and automation. Please consider re-structure these parts to make the presentation more precise by, e.g. merging redundant information.

Thank you for the valuable feedback. Please see our response to your specific comment 1.

4. Figure 2: the right panel of the figure contains major user stories that are purely text. Such information can be better presented using a table or in-text description rather than a graphic presentation (figure).

Thank you for the suggestion. We tested a table in a previous version of the manuscript and found that it is more challenging to connect stories to the chart shown in Figure 2 (left panel). Discussing all complete user stories in the main manuscript would increase the text size, which likely would impact the readability of the already extensive paper. We thus chose to keep the representation as is, as we believe that through the representation as one figure, readers will be able to identify and connect the major implemented user stories (right panel) with the cumulative plot showing the remaining user stories and the corresponding working hours (left panel). Together, this provides an overall picture of what was implemented during the reprogramming process. For the convenience of readers who would like to see a table, we included it in the supplementary material (progress_taking.xlsx) and now reference it in the figure description.

5. Section 4.3 title: get rid of the period after “Code development”.

We have revised the said title accordingly.

6. Section 4.3.2: narrative in this section contains many hyperlinks to GitHub repositories, which impedes the flow of the text and makes reading difficult. Authors may consider documenting these links in supplementary information (SI) and making reference to SI.

Thank you for the comment. We have replaced all hyperlinks to GitHub repositories in revised manuscript as bibtex citations with a “last accessed” field.

7. Table 2: A graphic presentation (e.g. bar plots) is recommended to better show the difference between the outputs of the reprogrammed and legacy software.

Thank you for the suggestion, however, we believe that Table 3 (previously Table 2) already provides a clear and comprehensive summary of the differences in the global-scale water balance components (in $km^3\ yr^{-1}$) between the reprogrammed and legacy WaterGAP global hydrological models.

8. Sections 9 does not contain much in-depth discussion, but rather recap of what has been presented in previous sections. Consider either merging this into concluding remarks (Section 10).

Thank you for your valuable comment. In response, we have removed the previous discussion section and included a dedicated "Lessons Learned" section. We refer you to the response to major points raised by you, where we elaborate further on the "Lessons Learned" section (pages 4-6, before specific comments).