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.

To improve our manuscript, we have

- revised key aspects of the Introduction and Methods sections (Sections 1 and 3, respectively). In particular, we have reformulated the manuscript's objectives.
- simplified the structure of the manuscript, moving the user survey to the supplement.
- enhanced the sections describing the programming process (Section 4) to avoid repetition with the Methods section.
- included a "Lessons learned" section (section 8) to benefit others undertaking similar efforts.

Reply to Reviewer 1: Rolf Hut

The authors present their work on reprogramming the WaterGAP model from its legacy code in C/C++ into python. The main purpose of this activity is to enhance the reproducibility of science done with WaterGAP, to make the science more transparent (FAIR) and overall reduce the effort required to maintain such a large code-base.

This is a worthwhile effort that will greatly help the (hydrological) scientific community in general and the WaterGAP user base in particular. The python version of WaterGAP has potential for bigger uptake, easier collaboration and is generally a better piece of research software than the legacy C/C++ version.

Thank you for highlighting the importance and quality of our paper.

I do, however, struggle with this publication and its place in the academic literature. I find it hard both to judge what the intention of the authors is with the publication and if they success in that.

Sidenote: reporting on the progress of software projects within academia is a struggle. The classic "report on results of work done so others can build on it"-structure of academic articles doesn't fit on reporting on developing new software, because the software in and of itself is not a scientific result. I would argue, however, that in the current age where academic credit is almost solely awarded based on "publications" a form of reporting on important software projects is needed. Both for informing the academic community on the new software (availability) and for rewarding / acknowledging those working on building that software. We had exactly the same problem when writing our eWaterCycle papers, where the first eWaterCycle paper never made it past peer review because it lacked "scientific results". In the second paper we focused on providing use cases to illustrate the platform and did manage to publish the work. I think this illustrates that GMD as a journal is accepting more and more software-like contributions.

Below I will list the different purposes I identified in the manuscript and provide feedback and tips for each different purpose to optimize the paper towards that purpose. I leave it to the combined team of authors and the editors of GMD to decide on which purpose they want to prioritize in the manuscript (or maybe even split in different manuscripts, for different audiences, using different platforms?)

Announcing reprogrammed version of WaterGAP for potential users

The new WaterGAP seems to me like an amazing tool for hydrologists to work with. Sections 2.1 (Model description), 5 (architecture), 6 (eval against sustainability criteria), 7 (Difference between output of C/C++ and Python versions) are essential for communicating this. For this focus I strongly suggest to add a few case studies that demonstrate the capabilities and user friendly-ness of the new WaterGAP.

Reporting on the process of reprogramming a legacy model

For those that contemplate reprogramming a legacy model, lessons learned from the transition from C/C++ version of WaterGAP to Python are very valuable. Section 3.1 on software evaluation against (FAIR) criteria and section 4 on the reprogramming process are very valuable here. I would add a paragraph on "lessons learned" that give pointers for others that set out to undertake a similar effort.

Reporting on user experience with the new WaterGAP codebase

The overview gathered from the survey conducted at EGU (section 3.3 and 8) gives some preliminary info on the perception of (potential) users of the new WaterGAP code towards its quality. This is in principle a valuable addition to the literature, but the width and execution of the survey is slim for a stand-alone publication. The selective response and low number of respondents make generalizing claims from this survey hard. I would strongly advice to present the results in terms of absolute numbers instead of percentages, so "10 people thought it was easy" versus "15% of people thought it was easy". For a full report on how outside users experience the new WaterGAP I suggest additional work, including for example a focus group session where users working with the new WaterGAP are observed.

Thank you very much for your thoughtful and constructive feedback. We appreciate your recognition of the value of our work and your insights into the challenges of publishing software-focused contributions in academic literature. Our goal, in fact, was not to announce the reprogrammed version of the WaterGAP but to report on the process of reprogramming a legacy model into a sustainable research software and to scientifically investigate the value of reprogramming scientific software. The main intention of the manuscript is to provide guidance for those who wish to reprogram the legacy code of the scientific model into sustainable software that can be easily maintained and improved, thus improving the reproducibility of the computational research done with this software. The comments provided by you and Reviewer 2, however, highlight that we did not succeed in conveying that in our initial manuscript.

We have revised the manuscript based on your feedback and our own reflections to focus explicitly and strongly on "Reporting on the process of reprogramming a legacy model". As you highlighted, the lessons learned from transitioning the WaterGAP model from C/C++ to Python are indeed valuable for undertaking similar efforts. In response, we have removed the previous discussion section and have included a dedicated "Lessons learned" section. This new section, along with sections 3 on software sustainability and the FAIR principles, and section 4 on the reprogramming process, significantly streamlines the manuscript's intention. We furthermore rephrased the abstract and introduction to clarify the focus of this paper.

Additionally, the section on the user survey has been moved to the supplementary material. We agree that the width and execution of the survey are slim for a standalone publication. Furthermore, the survey results are now presented in absolute numbers rather than percentages. The results of the user survey are now used as a supporting statement regarding the quality of the external documentation, code readability, and code modifiability in Section 6.

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"

Minor remarks

Independent of the direction chosen I have a few smaller remarks on the current version of the text:
- Line 61, add a citation to Wilkinson 2016 for FAIR.

We added a citation to Wilkinson 2016 for FAIR (Section 1, Lines 63-65).
"Additionally, applying FAIR (Findable, Accessible, Interoperable, and Reusable) principles for research software (FAIR4RS) improves research software reusability, reproducibility as well as transparency (Barker et al., 2022; Wilkinson et al., 2016)."

- Line 65: the unsuspecting reader might conclude that "to improve … … were reprogrammed" was done as part of the aforementioned eWatercycle project

Thank you for the comment. We have revised the section to better highlight the broader problem statement, rather than implying a direct connection with the eWatercycle initiative. The revised section (Section 1, Lines 69–73) now reads:
"*Efforts to improve comprehension, usage, maintenance, extension, and collaborative development have led to the reprogramming of several models, including the global land surface model CLASSIC (Melton et al., 2020) and GHMs such as HydroPy (Stacke and Hagemann, 2021) and PCR-GLOBWB (Sutanudjaja et als., 2018). However, the publications on these reprogrammed software focus on evaluating the performance of the model output and lack a detailed account of the reprogramming process and an evaluation of the success of the reprogramming effort*"

- Line 181 and continuing: I would avoid as much as possible using URLs as citations and use bibtex citations to websites (preferably with a "last accessed" field).

Thank you for the comment. We have replaced the majority of URLs in the revised manuscript as bibtex citations with a "last accessed" field.

- Section 4.1.2. I always learned that agile is a project management tool for when you have a fixed amount of hours, or people, to work on something, but the end goal is not fixed, just "create as much value as quickly as possible" (ideal for start-up culture). This is ideal when project goals are allowed to fluctuate during the project. In most scientific large projects the end goal is quite fixed (reprogram this model into python). Therefore, I would invite the authors to highlight why they settled on Agile as a project management practice for this project (there might well be very good reasons!)

Thank you for the valuable comment. We agree that Agile is often associated with projects where the end goal is flexible, which is common in start-up environments. However, we believe that certain Agile principles can also 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, ensuring everyone stays informed and coordinated throughout the project.

We now include this in the new lesson learned (Section 8, Lines 531-542)

" ***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.

"

- Line 288: "senior developers" are introduced as a role, but not previously explained.

We have explained the term senior developers in the revised manuscript (Section 4, lines 189-191).

"*After the writing and approval of the project proposal, a preliminary meeting was held in November 2021 among six senior developers. These are late-stage PhDs, PostDocs, and Professors with extensive expertise in the WaterGAP model and are also actively involved in developing and maintaining the software.*"

- Line 291: same for "software development advisor"

We have explained the term software development advisor in the revised manuscript (Section 4, lines 207-208).
"*The software development advisor guides the developers on best practices, architecture, and code quality to ensure robust and sustainable software.* "

Concluding
I really like the new python version of WaterGAP. (I like it so much that I would invite the authors to work together to add support for WaterGAP in eWaterCycle to make it even more accessible to hydrologists!). I hope the above suggestions will help in choosing a focus direction for the manuscript and emphasizing those parts throughout. I am happy to review an updated version of this manuscript.

Thank you for the thoughtful and encouraging feedback, and we are delighted to hear that you liked the Python version of WaterGAP. We appreciate your suggestion regarding integration with eWaterCycle. This is an exciting idea, and we would be very open to exploring such a collaboration in the future.