Dear Ting Sun,

Thank you very much for handling our manuscript. We appreciate your prompt and helpful feedback. 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.

With regards, on behalf of all authors,

Emmanuel Nyenah

**Response to Editor**

Thank you for submitting your work to GMD. I am pleased to inform you that your manuscript is acceptable for publication subject to minor revisions. Your work provides a valuable contribution documenting the transformation of WaterGAP into sustainable research software.

Thank you for highlighting the importance of our contribution to GMD with this manuscript.

However, two issues need attention:

1. Reference to Supplementary Materials

Replace all instances of "see Supplement" with proper citations:

- Archive supplementary files (software_requirement_WaterGAP.pdf, progress_taking.xlsx, etc.) in Zenodo

- Obtain DOIs

- Cite them properly in the manuscript

Thank you very much for spotting these issues. We agree that a generic reference to “see Supplement” is not helpful. We improved the manuscript in multiple places by providing more specific references, such as “see software specification document in the Supplement” or “see Excel file in the Supplement,” to guide the reader more effectively.

We chose not to follow your suggestion to upload them to Zenodo because, according to the submission guidelines (https://www.geoscientific-model-development.net/submission.html), we are permitted to upload files smaller than 50 MB as supplementary materials, which will be assigned a DOI during publication by GMD. Our supplementary files mainly consist of supporting PDFs—such as the questionnaire, response documents, software requirements, and a table of user stories. We believe uploading to a different platform does not improve the manuscript and could instead confuse readers. Since we meet GMD's requirements for supplementary data, we would appreciate it if this choice also receives your final approval.

We further removed the “supplementary information for” title from our main supplemental file as stated in the guidelines as well.

2. Imbalanced Section 7 (Lessons Learned)

Currently imbalanced with lesson 7.1 at ~500 words while lessons 7.3-7.6 are only 50-80 words each. Please consolidate into 4 lessons:

- Keep: 7.1 Runtime trade-offs (already well-developed)

- Keep: 7.2 Agile process (expand to ~400 words with specific examples)

- Combine: 7.3 + 7.4 → "Code architecture and design practices" (~400 words)

- Combine: 7.5 + 7.6 → "Sustainable development practices" (~400 words)

Each lesson should include: challenge → solution → example → recommendation.

Thank you very much for the suggestion. We have revised Section 7.2 to include examples, and we have combined Sections 7.3 and 7.4, as well as Sections 7.5 and 7.6. We focused on ensuring that each lesson follows a clear structure as suggested, while remaining true to the lessons that resulted from our project.

The revised text for the new 7.2 to 7.4 now reads (Section 7, lines 508-565):

“

***7.2 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. For example, tracking progress allowed us to monitor the number of user stories completed within each sprint, the time invested, and the remaining tasks to be tackled in the upcoming sprint. Such tracking helped us assess whether we were on track to complete the overall project within the required timeframe. 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.*

*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. Through sprint reviews and retrospective meetings (see Fig. 1), we collaboratively reviewed various user stories to assess whether changes in conceptualization and hence algorithms are needed. For example, we revised the algorithm governing surface water demand satisfaction due to the limited documentation available. During these meetings, we also discuss efficient technical solutions for some user stories, such as improving the runtime of the snow module and enhancing the overall runtime of the WGHM software. These discussions not only enabled conceptual alignment but also allowed us to find efficient technical solutions, incorporating input from product owners and the software development advisor. Despite these advantages, we faced several*

*challenges. Estimating the time needed to complete user stories proved difficult, and coordinating an agile process with only a few developers in an academic setting was somewhat challenging. Nevertheless, we recommend this approach for other reprogramming projects, as it supports timely and user-focused development and helps the team stay updated on progress.*

### *7.3 Code architecture and design practices are paramount for readable, maintainable and 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). A modular design also leads to improved readability, as single components of a project (e.g., code files) are more concise in their purpose.*

*Furthermore, good software engineering practice can further improve readability and maintainability, such as establishing meaningful and consistent variable names, which is also an iterative process that requires collaborative effort among developers and domain experts. For large projects, it is common for different developers to use various names for the same underlying concept, which can lead to confusion and subtle bugs if only one instance of a variable is updated (McConnell, 2004). For example, suppose a variable is referred to as both "storage_canopy" and "canopy_storage" in different parts of the code. In that case, an update to one variable may not automatically propagate to the other, resulting in inconsistencies in model output. Importantly, naming conventions need to be documented and enforced through the product owners and the active development team to guide future model development and reduce the risk of errors associated with ambiguous or inconsistent variable names.*

### *7.4 Sustainable development practices such as documentation and automation ensures efficient software development and high software quality*

*Without sufficient documentation, it is challenging to comprehend the underlying concepts of algorithms and to modify, extend, or utilize the resulting software. An example is highlighted in Section 5, where we revised the algorithm governing surface water demand satisfaction and its impact on return flows to groundwater. The legacy code lacked sufficient in-code and external documentation, making it hard to understand and to re-implement. As a result, we developed an improved and consistent algorithm, accompanied by in-depth documentation. Throughout the project, we did not copy old documentation from the previous model; instead, we wrote it from scratch, keeping in mind the sustainability of the new software. We strongly recommend writing model documentation alongside code development rather than leaving it until the end. This approach helps 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.*

*Manually updating documentation, running tests, and linting can be time-consuming, especially for large software projects. Developers may even forget to perform these tasks after modifying or extending the software, which can lead to buggy or broken code. Automating documentation generation reduces manual effort and helps keep the documentation up to date. Similarly, automating linting and testing ensures that the code functions correctly without the need for constant manual*

*checks. Automation is key to efficient development and high software quality, and we strongly recommend adopting this practice.”*