The Geological Society of America
Special Paper 558

Improving reproducibility of geoscience models with Sciunit

Raza Ahmad
School of Computing, College of Computing and Digital Media, DePaul University, 243 South Wabash Avenue, Chicago, Illinois 60604, USA

Young Don Choi
Jonathan L. Goodall
Department of Engineering Systems and Environment, University of Virginia, 151 Engineers Way, P.O. Box 400747, Charlottesville, Virginia 22904, USA

David Tarboton
Ayman Nassar*
Utah Water Research Laboratory, Utah State University, 8200 Old Main Hill, Logan, Utah 84322, USA

Tanu Malik
School of Computing, College of Computing and Digital Media, DePaul University, 243 South Wabash Avenue, Chicago, Illinois 60604, USA

ABSTRACT

For science to reliably support new discoveries, its results must be reproducible. Assessing reproducibility is a challenge in many fields—including the geosciences—that rely on computational methods to support these discoveries. Reproducibility in these studies is particularly difficult; the researchers conducting studies must agree to openly share research artifacts, provide documentation of underlying hardware and software dependencies, ensure that computational procedures executed by the original researcher are portable and execute in different environments, and, finally, verify if the results produced are consistent. Often these tasks prove to be tedious and challenging for researchers.

Sciunit (https://sciunit.run) is a system for easily containerizing, sharing, and tracking deterministic computational applications across environments. Geoscience applications in the fields of hydrology, solid Earth, and space science have actively used Sciunit to encapsulate, port, and repeat workflows across computational environments. In this chapter, we provide a comprehensive survey of geoscience applications that have used Sciunit to improve sharing and reproducibility. We classify the applications based on their reproducibility requirements and show how Sciunit accommodates relevant interfaces and architectural components to support reproducibility requirements within each application. We aim to provide these applications as a Sciunit compendium of use cases for replicability, benchmarking, and improving the conduct of reproducible science in other fields.

**Erratum*: In the first version of this chapter published online, "*Ayman* Nassar" was misspelled as "*Ayam* Nassar." GSA sincerely regrets this error.

Ahmad, R., Choi, Y.D., Goodall, J.L., Tarboton, D., Nassar, A., and Malik, T., 2022, Improving reproducibility of geoscience models with Sciunit, *in* Ma, X., Mookerjee, M., Hsu, L., and Hills, D., eds., Recent Advancement in Geoinformatics and Data Science: Geological Society of America Special Paper 558, https://doi.org/10.1130/2022.2558(07).

1. INTRODUCTION

During this decade, the scientific method has become increasingly computational, involving large quantities of data, complex data manipulation tasks, and large and often distributed software stacks. To assess the reproducibility of the computation-based scientific method, scientists are increasingly encouraged to make their studies FAIR: findable, accessible, interoperable, and reusable (Wilkinson et al., 2016). Scientists generally comply with FAIR by uploading their computational artifacts to online repositories such as GitHub (https://github.com), Figshare (https://figshare.com), HydroShare (https://hydroshare.org), and Zenodo (http://zenodo.com). These repositories store and index code, data, and documentation. They also assign digital object identifiers (DOIs; https://www.doi.org) and provide interfaces for finding (F) and accessing (A) artifacts. Tools for creating interoperable (I) and reusable (R) artifacts, however, are lacking, which is acutely evident when reviewers attempt to reproduce a scientific study.

Consider the following scenario: Alice, a geoscientist, has recently published a journal article related to hydrology on the declining rate of evaporation in the Carolinas. Her model and analysis are available as part of five Jupyter (https://jupyter.org) notebooks, whereas both code and data are available via the HydroShare (https://hydroshare.org) portal. In the article, the author's model predicts the rate of evaporation as 5 mm/d; Bob, a reviewer, believes it is close to 3.5 mm/d. He wants to change a few methods and use alternate data sets in the model to test the sensitivity of the evaporation rate. Bob downloads the notebooks and initially faces issues in setting up the environment. He fixes the environment by downloading the necessary packages, but his result is ~6 mm/d, which is surprising. He wonders if the notebook is using the same data set as was used in the earlier experiment. The data set used is large and requires careful comparisons. He is aware of alternate data sets on HydroShare with which he would like to experiment. However, using them will engage him in a drawn out, iterative process as the model sensitivity parameters of the previous data set are unknown.

In the above example, the scientists *did* adopt notebooks and data hubs that make computational artifacts findable and accessible. However, verifying reproducibility continues to be both time-consuming and challenging due to the lack of tools that create reusable artifacts and enable their interoperability with new methods and data. Consequently, researchers mostly implement the first two, but not all, of the FAIR practices. In a recent study, Stagge et al. (2019) found that of 360 articles from six leading hydrology and water resources journals, 49% made some materials available online; only 6% made code, data, and directions publicly available; and only 1.1% of sampled articles could be fully reproduced.

There are several known methods for creating reusable and interoperable artifacts. Two of the popular methods are virtualization and containerization. Virtualization isolates computing environments and creates reusable artifacts. However, the maintenance cost of virtualization is high, and it is only supported by cloud computing platforms like AWS (https://aws.amazon.com) and Infrastructure as a service platform such as Jetstream (https://jetstream-cloud.org) and XSEDE (https://www.xsede.org). In containerization, an application is built from well-known packages (https://docker.com) and is popularly used in DevOps to encapsulate standard applications such as database servers, web servers, and compilers. However, containerization continues to require significant training and education to create Dockerfiles, resolve versions, determine appropriate image identifiers, and manage the resulting Docker containers on the server-side. Commercial systems such as Binder (https://mybinder.org) create Docker containers but restrict the languages in which workflows are developed, and they do not equip users to create, share, and manage containers.

We recently proposed an alternative method for creating containers that uses the reference execution of an application to automatically create a container with application provenance to ensure repeatability across environments (Pham et al., 2013). Sciunit (Yuan et al., 2018), the resulting tool, simplifies container creation and maintenance. To examine whether Sciunit can be directly used by scientists and efficiently integrated with the underlying cyberinfrastructure used by them, we teamed with geoscientists as part of the National Science Foundation's EarthCube-funded projects "GeoDataspace,"[1] "GeoTrust,"[2] and "ReproBench."[3] These projects resulted in some significant outcomes:

- Scientists created containers, and, thus, reusable objects, using Sciunit with a single command. These containers are unlike the commercial containerization tools available and have simplified the containerization process.
- Scientists downloaded and reran the resulting reusable artifacts without additional server or client maintenance. This simplified repeatability.
- Sciunit is easily integrated with findable and accessible data portals such as HydroShare, which is used by hydrologists. This made it easy to build and share reusable artifacts over the web.
- Sciunit included additional interfaces—such as integration with Jupyter notebooks—to efficiently support the complex reproducibility requirements of geoscience computational models.

In this chapter, we demonstrate these outcomes through geoscience applications to model development and their reproducibility requirements. In these applications, the objective is to containerize the application and create a reusable research object using Sciunit. We describe additional features of the Sciunit interface and architecture to create a reusable and interoperable research object. We primarily focus on demonstrating Sciunit commands within the geoscience applications and the specific

[1]https://nsf.gov/awardsearch/showAward?AWD_ID=1722152
[2]https://nsf.gov/awardsearch/showAward?AWD_ID=1639759
[3]https://nsf.gov/awardsearch/showAward?AWD_ID=1928288; https://nsf.gov/awardsearch/showAward?AWD_ID=1928369; https://nsf.gov/awardsearch/showAward?AWD_ID=1928315

ways in which they ease the reproducibility requirements and provide seamless integration with the underlying community cyberinfrastructure—such as HydroShare—used by the application. Each resulting research object is available via HydroShare and its corresponding link is provided. Sufficient documentation is provided to enable users to reuse the available sciunits.

The contributions of our work in this chapter are summarized as follows:

(1) We describe six characteristics of geoscience applications based on their reproducibility requirements. All applications focus on one or more aspects of computational modeling. While some applications may run in a parallel or distributed mode, we focus on a single-node operation.
(2) We explain five use cases that have motivated changes in the architecture and functionality of Sciunit. Each functionality is described with the relevant code samples.
(3) We introduce four new functionalities in Sciunit to satisfy FAIR-based reproducibility requirements for geoscience computational workflows. These functionalities are *import*, *export*, *content-based diff*, and *provenance-based diff*.

The rest of the chapter is organized as follows. We present the characteristics of geoscience application in Section 2. Section 3 explains the terminologies and concepts from the field of computer science used in this chapter. Section 4 provides an overview of the Sciunit software. We then present our geoscience use cases in Section 5, each of which has influenced the features and architecture of Sciunit and resulted in a declarative Sciunit command line to create a reusable research object. We conclude in Section 6.

2. CHARACTERISTICS OF GEOSCIENCE WORKFLOWS

We describe the characteristics of geoscience applications that focus on computational model development. We consider the entire workflow underlying these applications, which consists of data preparation, model description, execution, and analysis. We describe how these characteristics make it harder to satisfy FAIR-based reproducibility requirements.

(1) Workflows are often developed in a variety of programming languages. Modelers frequently use additional software to prepare inputs for a specific model (pre-processing software) and analyze outputs generated by the model (post-processing software). Consequently, models are often a combination of smaller software modules or components contributed over time by many individuals and groups. There is usually no consistent standard for the choice of a programming language in developing these software modules. However, to reproduce large experiments that depend on these modules, it would be preferable to reproduce each module in the same transparent manner rather than choosing a language-specific method for reproducing each module.

(2) Workflows change environments, data sets, and methods. A functional reproducibility requirement is the ability to obtain consistent results. Such results are usually obtained by testing and experimenting with a different implementation of the model, i.e., new code but with the same underlying equations or principles. A test could evaluate whether the code and solvers implement the equations accurately and interpret any differences in results.

(3) Workflows analyze model sensitivity. Modelers often spend weeks or months building, calibrating, and validating their models. Steps in this process are rarely automated, and the methods for completing these steps generally involve tacit knowledge that is difficult to automate. The exact parameters to vary may be mentioned in research articles or hard-coded in scripts, but are not easily captured in a tool used for reproducible analysis.

(4) Workflows use different interfaces. Model workflows that ease the sharing of code and data are increasingly being developed in notebooks, which operate in their specialized environments, but sharing and reproducing notebooks that were developed in other environments is a challenge. A seamless notebook sharing environment is essential for developing models using notebooks and subsequent collaboration using them.

(5) Workflows interact with services. To ease data discovery and access, facilities often make data products available through public repositories. Models usually rely on such remote data collections, and their scripts typically include web requests and queries to fetch data from these collections. However, the content of remote repositories may change over time, which affects the reproducibility of any analyses that depend on the downloaded data.

(6) Workflows perform result validation. Execution of the model is not the end; it is often necessary to validate whether the model used exactly the same sequence of instructions and produced the same set of output files with the same content.

3. BACKGROUND

In this section, we explain the various terminologies and concepts from the field of computer science used throughout this chapter. Readers familiar with these concepts can move on to the following sections.

API: Application programming interface (API) is a software interface that defines the interactions between various software applications.
CI: Continuous integration (CI) is a practice in software engineering in which frequent, independent changes to software code are integrated and tested continuously.
Cloud computing: Cloud computing refers to the on-demand availability of various computing services over the internet, especially processing power and data storage.
Computational environment: A computational environment includes the configuration files, shell environment, software packages, and other program dependencies.
Container: A container is an independent software package that contains everything that is required to run an application successfully.
Dependency: A dependency for a computer program is a piece of code or data that is required to run that program.

DevOps: DevOps stems from development operations and refers to the set of practices and tools that increases an organization's ability to deliver high-quality applications and services.

Docker: Docker is a popular tool for packaging software into isolated containers. A Docker container can be created automatically using a set of instructions in a text document called a Dockerfile.

Infrastructure as a service: Infrastructure as a service is the most basic category of cloud computing services, where enterprises rent or lease servers for compute and storage in the cloud.

Notebook: A notebook is an interactive, online environment in which users write computer code and perform data analysis in a read–evaluate–print–loop (REPL) style.

Package: A software package refers to a collection of computer programs in a portable form that allows them to be widely distributed.

Research artifact: A research artifact refers to computational artifacts such as code, binaries, libraries, scripts, data sets, documentation, and other files that have contributed to specific research output.

Research object: A reusable research object is defined as an aggregation of research artifacts. A research object must adhere to FAIR policies.

Sandbox: A sandbox is an isolated computational environment that replicates an actual computational environment to run code and perform analysis.

Solver: A solver is a piece of code that solves a mathematical problem.

System call: A system call refers to a request made by the software program to the operating system for accessing a computer system resource, such as a network, hard disk, or other devices.

Virtual environment: A virtual environment is a self-contained and isolated computational environment that enables developers to work on multiple projects with different sets of dependencies at the same time on a computer.

Virtualization: Virtualization is the process of creating independent virtual environments.

4. Sciunit

Sciunit (Pham et al., 2013; Ton That et al., 2017; Yuan et al., 2018; Ahmad et al., 2020) is a tool that creates research objects that are shareable and reusable in other Unix-based computational environments. The research object created by Sciunit aggregates code, binaries, scripts, libraries, data sets, and environments; the research object is containerized and reusable in isolation. Sciunit observes a program and captures the trace of its execution using system utilities. It stores the program lineage comprising the sequence of system calls executed as part of the program, as well as the input and output data content used by that program.

Sciunit typically runs in two modes: an audit mode to create a container and an execution mode to re-run a container. In the audit mode, a container of a user application is created as the user executes the application. In the context of auditing, such execution is termed a *reference execution*. We describe the audit process assuming that the application is running on a Linux machine.

Audit mode: During execution, the Linux *strace* utility is used to monitor the running application process. *strace* internally attaches itself to the process using the *ptrace* system call to monitor all of the system calls of the running process. It intercepts each system call to determine the state of the running process and the arguments to the system call. For example, when a process accesses a file or a library using the system call *fopen()*, the *fopen()* call is intercepted. The intercepted system call is "paused" to examine input arguments and the process control block. For instance, in *fopen()*, the file path parameter is extracted. By intercepting all system calls, the auditing determines the contents of the research object, i.e., all program binaries, libraries, scripts, and environmental variables upon which a user program depends. Inclusion of data files is optional, which the user may or may not want to package based on the size of the data set.

During the pause, the identified dependencies are used in two ways: first, to create a "sandbox" application container that includes all identified dependencies, and second, to create an interaction log of the reference execution. The sandbox container is named with a package hash and placed in a special "root path." It contains all the dependencies that were identified during the audit of reference execution. These dependencies are placed at the same path within the special root path where they were originally identified. The interaction log generated during the audit phase contains interactions between the two processes when they are created through the *fork* or *exec* system calls, or between processes and files when files are opened or closed. The log also stores the logical range of the number of times the processes interact with other processes or files. The interaction log is topologically sorted to obtain a low-fidelity provenance graph.

Execution mode: In the execution mode, the application is executed from the container itself by monitoring its processes with *strace*, interrupting application system calls, and then extracting their path. Figure 1 illustrates the audit and execution phases. Sciunit turns the package (*Pkg* in the figure) into a self-contained, lightweight research object. It could be shared across different environments with collaborators or members of a publication review committee to reproduce their research at any time.

Sciunit operates in the user space and tracks programs with the help of utilities provided by the underlying operating system. It has a simple installation process (https://sciunit.run/install) and provides an easy-to-use command line interface and a Python API for end users. Sciunit is available and easily installed on a Linux system using the package manager pip for Python 3 as follows:

```
pip install-user sciunit2
```

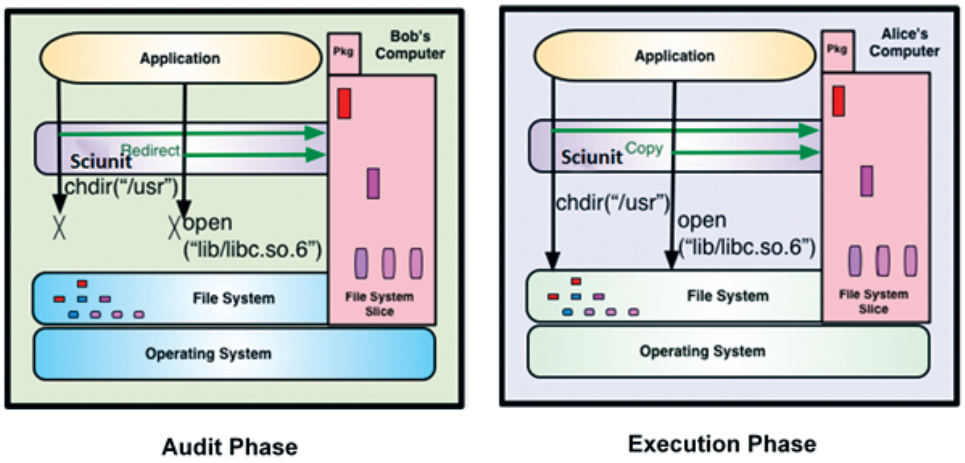

Figure 1. Diagram gives an overview of reproducibility using Sciunit. During the audit phase, Sciunit captures the execution of a program on Bob's computer into a container. During the execution phase, that container is taken to Alice's computer and repeated exactly.

Detailed installation instructions for different versions of operating systems and the list of dependencies can be found on the official Sciunit website (https://sciunit.run).

5. GEOSCIENCE APPLICATIONS

We discuss the reproducibility requirements of geoscience applications for five use cases from the domain. As we discuss each one of these, we will highlight the areas where reproducibility is an essential requirement. We attempt to address these reproducibility requirements by describing the changes made to the interface and architecture of Sciunit.

5.1. Geoscience Application 1: Variable Infiltration Capacity

The variable infiltration capacity (VIC) model is a macro-scale hydrologic model that applies water and energy balances to simulate terrestrial hydrology at a regional level (Liang et al., 1994). It is used for several applications, including water management in reservoirs, studying climate change, and simulating streamflow. The VIC model has been applied to many river basins around the world.

The input to the VIC model comes from several different sources that include hydrometeorological data, soil map, wetland features, land cover map, vegetation properties, and digital elevation model (DEM). The input data are prepared through a complex data pre-processing phase that requires significant time and effort. This pre-processing step is composed of several programs written in different languages. The output of the VIC model consists of different files that contain information on river discharge and the intermediate hydrologic processes. Figure 2 illustrates various components of the data pre-processing pipeline.[4] Here is a high-level overview of its workflow:

(1) Process precipitation and air temperature data sets.
(2) Process the wind speed data set.
(3) Process the land surface data sets including topography, soil, and vegetation data.
(4) Create the final input files for meteorological data sets.

5.1.1. Support for Various Programming Languages

In the absence of Sciunit, a geoscientist would have to enumerate all dependencies of VIC, which include code written in C, C++, FORTRAN 77, Python, and shell scripts. In total, VIC consists of 97 scripts and binaries, 11,481 data files, and 357 dependency files for a total of 2.3 GB. Also, some of these dependencies are not readily available. As an example, one of VIC's FORTRAN dependencies at the time of experimentation was installed on a single machine and not widely available over the internet. Therefore, executing and reproducing the entire VIC pipeline successfully is an arduous and time-consuming task.

For VIC, we create a Sciunit project as follows:

```
sciunit create vic
```

The four workflow steps of VIC were run from a single script. To use Sciunit, the user ran the script on the bash shell terminal using the Exec command:

```
sciunit exec VICscript
```

Sciunit captures the complete trace of the execution of VICscript and creates a research object that includes all input data, binaries, dependencies, scripts, and output files that were part of the execution. The execution of this script is stored in the Sciunit database and assigned a label of the form *en*, where *n* is a monotonically increasing positive integer. The List command shows the list of all programs executed in the VIC project:

```
> sciunit list
e1 May 9 2021 12:00 VICscript
```

Details for each execution can be viewed using the Show command:

```
> sciunit show e1
id: e1
sciunit: vic
command: VICscript
size: 2.3 GB
started: May 9 2021 12:00
```

[4]Complete code for the VIC pre-processing workflow and its sciunit is described here: https://www.hydroshare.org/resource/c7619f345a364b8bb27e87c0b9213a75.

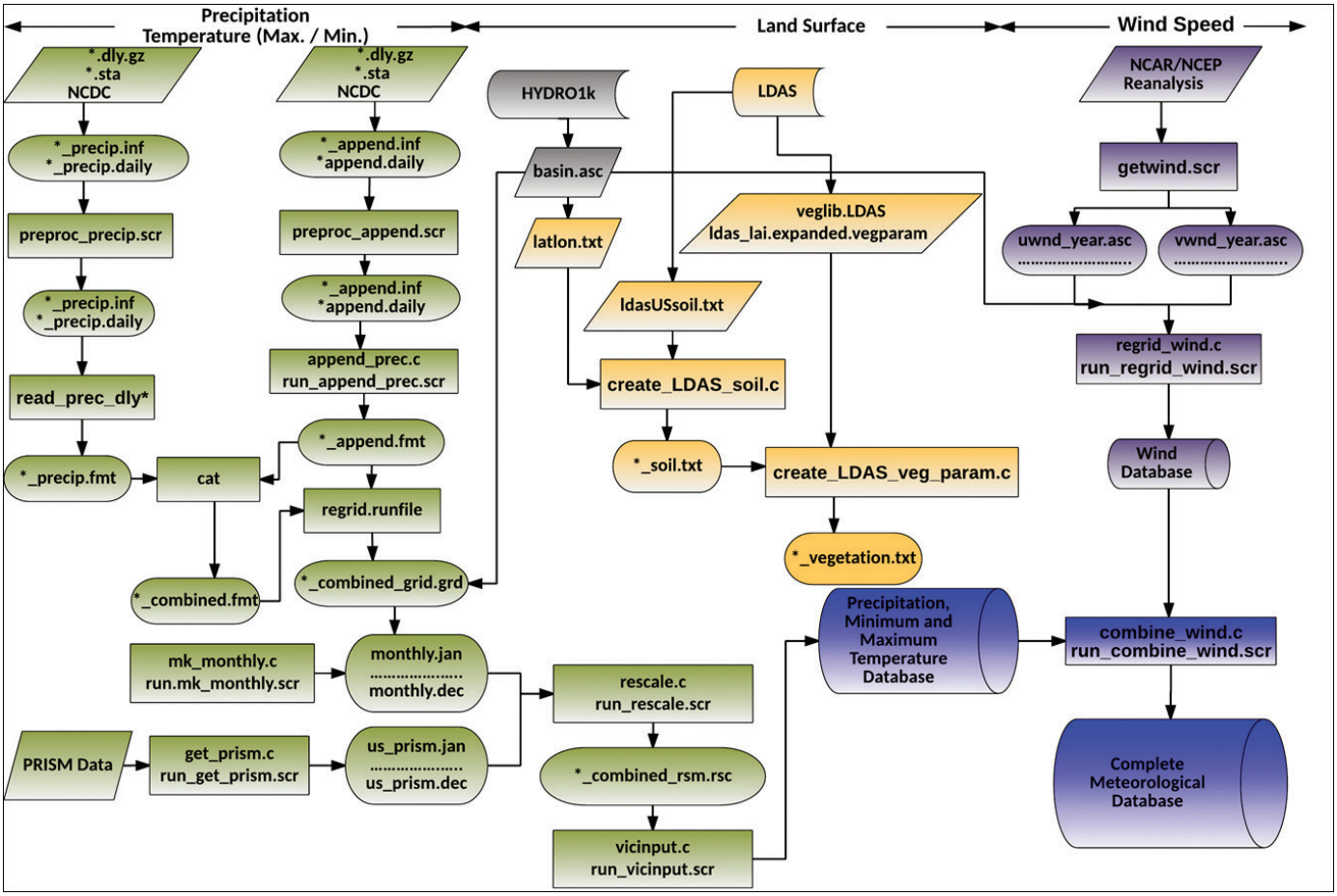

Figure 2. Various steps in the data pre-processing workflow for the variable infiltration capacity (VIC) model are shown; adapted from Billah et al. (2016). NCAR/NCEP—National Centers for Environmental Prediction/National Center for Atmospheric Research; LDAS—Land Data Assimilation Systems; PRISM—Pliocene Research, Interpretation and Synoptic Mapping; NCDC—National Climatic Data Center.

To share this Sciunit project and repeat its executions on another machine, the Copy command is used. It stores the project temporarily on a remote server and assigns it a unique token:

```
> sciunit copy
#k87A9a2e
```

The token *#k87A9a2e* generated above is temporary and in this demonstration is used on another machine to recreate the Sciunit project and all its executions there:

```
> sciunit open #k87A9a2e
opened sciunit project at ~/sciunit/vic
```

The user can again list the executions in this project using the List command. The same execution is now repeated using the Repeat command:

```
> sciunit repeat e1
```

This instruction reuses the research object and repeats the execution of VICscript. It uses the same input files and follows the same sequence of instructions. Since VIC is a deterministic program, it generates the same outputs.

5.2. Geoscience Application 2: MODFLOW

MODFLOW (McDonald and Harbaugh, 1988) is the U.S. Geological Survey's (USGS) three-dimensional, finite-difference flow model for solving the groundwater flow equation that is used by hydrogeologists to simulate the flow of groundwater through aquifers. The model is open source and primarily written in FORTRAN programming language. USGS has released multiple versions of MODFLOW, but MODFLOW-2005 (Harbaugh, 2005) is the most widely used. MODFLOW-NWT (Niswonger et al., 2011) is a version of MODFLOW-2005 that uses a Newton-Raphson formulation to improve the solution of unconfined groundwater-flow problems. MODFLOW-NWT is a stand-alone program for solving problems involving drying and rewetting nonlinearities of the unconfined groundwater flow equations.

We consider a use case where MODFLOW is used to model the shallow groundwater flow in the James River watershed upstream of Richmond, Virginia, USA (Essawy et al., 2018).

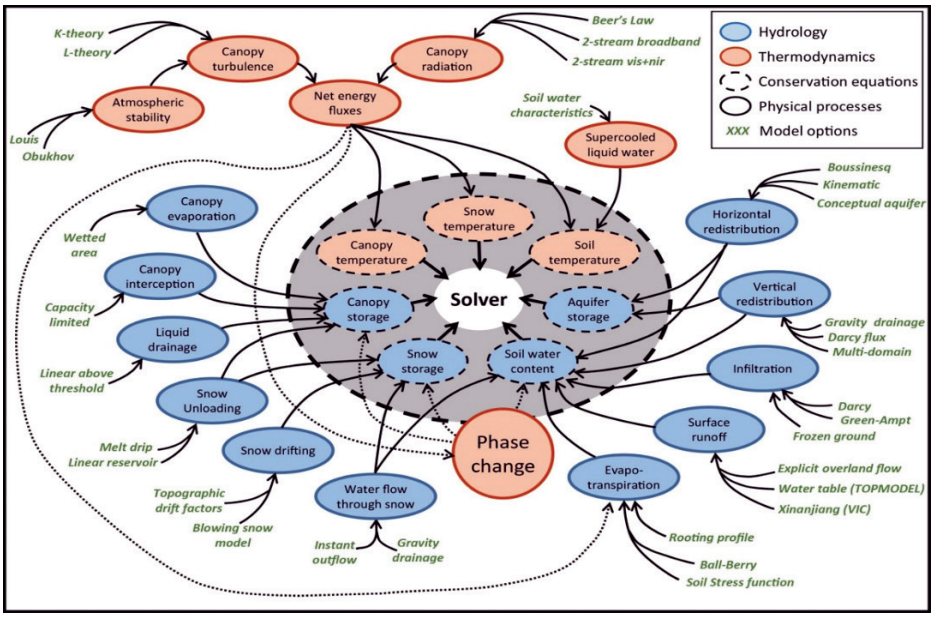

Figure 3. An illustration of the Structure for Unifying Multiple Modeling Alternatives (SUMMA) framework is used to describe the application of multiple process parameterizations with conservation equations and a numerical solver (Clark et al., 2015a, p. 2505; used with permission of the American Geophysical Union).

The model includes recharge to the water table, subsurface flow through the saturated zone, and baseflow discharge to surface water bodies including the James, Rivanna, and Hardware Rivers, as well as several smaller-order streams. The model is created using FloPy: a Python library for implementing various steps in MODFLOW model development, execution, and analysis (Bakker et al., 2016). A high-level overview of the code[5] follows:

(1) Download raw input data from HydroShare.
(2) Define and prepare input files for the model.
(3) Specify model output format.
(4) Define the various model attributes and properties for the use case.
(5) Run the model simulations.

5.2.1. Workflows Have Different Versions

Sciunit allows users to execute programs that span multiple programming and scripting languages. For the above use case, the individual components are written in Python, and the entire model workflow is then packaged into a shell script. Sciunit executes the model simulations and generates the output files. For different versions of MODFLOW, a user can execute their workflows individually using Sciunit as follows:

```
sciunit create modflow
sciunit exec modflow_2005.sh
sciunit exec modflow_nwt.sh
...
```

The workflows of different MODFLOW versions are executed separately as part of the same project in Sciunit, which will distinguish the versions but store them compactly in its deduplication engine. The deduplication engine only stores the data blocks that are unique across the files of all program executions in a project. This reduces the disk space required to store the project executions. All executions in the MODFLOW project are listed using the List command:

```
> sciunit list
e1 May 9 2021 12:00 modflow_2005.sh
e2 May 9 2021 13:00 modflow_nwt.sh
...
```

5.3. Geoscience Application 3: SUMMA

Structure for Unifying Multiple Modeling Alternatives (SUMMA) is a hydrologic model that enables the systematic and controlled evaluation of multiple model representations and provides insight into the advanced unified modeling framework (Clark et al., 2015a). It gives users the flexibility to evaluate the interplay between the model and process parameters and provides the capabilities to experiment with different numerical solvers. Figure 3 illustrates the construction of the SUMMA model. It consists of a solver with outer branches and produces a numerical solution with a conservation equation from water and energy.

We use a case study of Clark et al. (2015b), which describes a set of modeling experiments that explores various hydrologic modeling scenarios using SUMMA. We run the code in a Jupyter notebook inside the *Anaconda* virtual environment.[6] A high-level overview of the code follows:

(1) Download SUMMA model instance from HydroShare.
(2) Configure various settings of the SUMMA model execution.

[5]The MODFLOW-NWT sciunit is accessible as a Hydroshare resource from here: https://www.hydroshare.org/resource/06fe2d66751b4c6e86838ecebb8c552d.

[6]The referenced PySUMMA notebooks are available from here: https://www.hydroshare.org/resource/28e3bb9c42ec4fca9b841afbfa660764.The notebooks generate the corresponding sciunits as described in this usecase.

(3) Execute SUMMA for different stomatal resistance methods.
(4) Calculate total evapotranspiration (ET).
(5) Visualize total ET against different times of the day.

A cross-section of the code for step 3 above is shown in Figure 4. It explores the impact of three different stomatal resistance parameterizations on total evapotranspiration: the *simple soil resistance* method, the *Ball Berry* method, and the *Jarvis* method.

5.3.1. Changing Data Set and Methods

Scientists running this experiment run their code multiple times with different stomatal resistance methods for each one. It allows them to reuse their code and analyze the differences across executions. The code snippet shown in Figure 4 runs the SUMMA simulation (Choi et al., 2021) using its Python API, PySUMMA, and writes the configuration and output files on the disk. The user provides the stomatal resistance method *stomResist* as a command line argument. We run this program with Sciunit in the following manner with a simple resistance method:

```
sciunit exec pysumma_example.py
simpleResistance
```

This executes pysumma_example.py, which runs the SUMMA simulation with the simpleResistance method. After successful execution, Sciunit stores it in its database and labels it e1. This is viewed by running the List command, which displays the output as below:

```
> sciunit list
e1 Jan 29 2021 12:01 pysumma_example.py
simpleResistance
```

We now run the same program with a different stomatal resistance method using the Given command as follows:

```
sciunit given BallBerry repeat e1
```

The original execution e1 now runs in the same manner, except that it uses the *BallBerry* method instead of *simpleResistance*. This repeated execution of the program using the Given command is not automatically saved to the database since it is a repetition of the execution e1 but with a changed input parameter. To store it as a separate execution, we manually save it using the Commit command:

```
sciunit commit
```

The above statement commits the most recent unsaved execution to the Sciunit database.

5.3.2. Notebook Interaction

As mentioned earlier, the PySUMMA code was run in a Jupyter notebook in the *Anaconda* virtual environment. Notebooks make the task of programming more streamlined, interactive, and customizable. They combine the code, images, videos, graphs, and other graphical user interface components in a single place. However, they also make it challenging to structure, test, and version the code. It is also difficult to run a long-running, asynchronous task on Jupyter. Moreover, sharing and reproducing notebooks developed in other environments is also a challenge. Thus, in our experience, after initial development and experimentation, scientists often need to move their code to another computational environment for execution, sharing, and reproducibility.

We ran PySUMMA inside the *Anaconda* virtual environment that was instantiated in the Jupyter notebook. Virtual environments like *Anaconda* and *Virtualenv* allow users to manage multiple isolated computational environments without interfering with each other. Each environment has its own set of dependencies and their specific versions. These environments are often constructed seamlessly for each project according to their requirements.

We believe that if a transition of code becomes necessary, the transportation of the computational environment should be universal, so that reproducibility is easy and efficient. The new environment must contain all of the dependencies of the original environment, including when it was using a virtual environment—such as one created by *Anaconda* or *Virtualenv*—for dependency management.

Sciunit provides two functionalities for interacting with notebooks and the virtual environments: *import* and *export*. In import, scientists import or fetch the dependencies into a computational environment from another environment. This allows them to run certain programs within the same environment, which did not have their dependency requirements already satisfied. The export functionality creates a new computational environment from an existing one. For a given program, Sciunit

```
18  # Create pySUMMA Simulation Object
19  S = ps.Simulation(executable, file_manager)
20
21  # set the simulation start and finish times
22  S.manager['simStartTime'].value = "2021-01-01 00:00"
23  S.manager['simEndTime'].value = "2021-01-30 00:00"
24  # set simple soil resistance method and run the model
25  # with the output prefix from {"simpleResistance", "BallBerry", "Jarvis"}
26  S.decisions['stomResist'].value = stomResistMethod
27
28  # Save configiuration to disk
29  S._write_configuration()
30  S.run(run_option="local", run_suffix=stomResistMethod)
31  S.output.to_netcdf("model.output")
```

Figure 4. Cross-section of the code for the PySUMMA model, which takes a user argument for different *stomResist* methods, is shown.

tracks all of the dependencies and the complete trace of its instruction sequence, which enables Sciunit to reproduce the program as and when required by the user. The export functionality allows the user to export these captured dependencies outside of Sciunit into another computational environment. The output of Export command describes the computational environment that was contained within Sciunit, which could then be used to create another virtual environment using *Anaconda* or *Virtualenv* or an isolated computational environment like a Docker image or a virtual machine. The Export command is issued from the Sciunit interface as shown below:

```
sciunit export <execution id>
```

The output of the above command is a requirements.txt file that contains all of the dependencies of the given execution. A small cross-section of the output file for PySUMMA is shown in Figure 5. Each line in the file contains the name of the library file and its version used by the execution. Notice that the program has a total of 146 Python libraries listed as dependencies that were used during its execution. A few notable ones include *pysumma*, *sklearn*, and *numpy*.

```
121  wrapt==1.11.2
122  cligj==0.5.0
123  backcall==0.2.0
124  prompt_toolkit==3.0.5
125  HeapDict==1.0.1
126  more_itertools==8.4.0
127  bleach==3.1.5
128  entrypoints==0.3
129  numpy==1.19.1
130  memory_profiler==0.57.0
131  mccabe==0.6.1
132  pyzmq==19.0.1
133  pyepsg==0.4.0
134  decorator==4.4.2
135  pycparser==2.20
136  astroid==2.4.2
137  sklearn==0.0
138  pysumma==3.0.0
139  MarkupSafe==1.1.1
140  PyYAML==5.3.1
141  zipp==3.1.0
142  zict==2.0.0
143  pytest==5.4.3
144  lazy_object_proxy==1.4.3
145  oauthlib==3.1.0
146  pyproj==2.6.1.post1
```

Figure 5. A cross-section of the output file requirements.txt for the Export command executed on the PySUMMA code is shown.

The requirements.txt file generated above is useful in many ways. We list a few of them below:

- The user can analyze and inspect the list of requirements to keep track of all of the dependencies and their versions. Developers keep making changes in actively used packages, which can potentially break applications if a specific package version is not available. Bookkeeping in the form of a requirements file prevents unexpected changes.
- The requirements file generated for a given program is also useful for recreating the computational environment for that program on another machine. For example, we can use the Python package manager pip as follows:
  `pip install -r requirements.txt`
- The installation of a specific set of requirements is used to instantiate a fresh computational environment, which only contains the dependencies for the given program. This currently is achieved in two different ways:
  (1) A new virtual environment is created using environment managers like *Virtualenv*, *Pipenv*, and *Anaconda*.
  (2) A virtual machine or container is created using programs like Docker. The Export command provides an option for generating a Dockerfile in addition to the requirements file. The resultant Docker image is a very thin container with just enough resources installed to run the given program.

5.4. Geoscience Use Case 4: HAND

Height Above the Nearest Drainage (HAND) is a terrain model that normalizes topography according to the local heights relative to the drainage network (Rennó et al., 2008; Nobre et al., 2011). It has been used in the modeling of flood inundation and in the computation of reach-scale hydraulic properties for flood modeling (Zheng et al., 2018; Garousi-Nejad et al., 2019). TauDEM software provides a vertical distance down function for the computation of HAND using the D-Infinity flow model employed in this use case (Tesfa et al., 2011).

We use a case study that demonstrates the use of the HAND model for the Little Bear River near Paradise, Utah, USA. The workflow for computing HAND in this example follows:

(1) Hydro-condition the Digital Elevation Model (DEM) by removing pits (filling internally draining grid cells).
(2) Delineate stream network.
(3) Compute D-Infinity flow direction.
(4) Calculate HAND.

5.4.1. Result Validation

During computation of the drainage network, a minimum contributing area *threshold* is used to identify the channel beginning. With a lower threshold value, the density of the resulting drainage network increases. Scientists running this experiment are interested in finding out how the threshold makes a difference in the execution and result of the HAND model. These differences could lie in the files being read by the model, configuration

```
# Use 5000 for 30 m grid cells and 50000 for 10 m grid cells
threshold = threshold_value
# TauDEM HAND procedur
!mpiexec -n 4 pitremove -z {DEM} -fel fel.tif
!mpiexec -n 4 d8flowdir -fel fel.tif -p p.tif -sd8 sd8.tif
!mpiexec -n 4 aread8 -p p.tif -ad8 ad8.tif
!mpiexec -n 4 threshold -ssa ad8.tif -thresh {threshold} -src srctemp.tif
!mpiexec -n 4 moveoutletstostrm -p p.tif -src srctemp.tif -o {outlet} -om Outletmv.shp -md 50
!mpiexec -n 4 aread8 -p p.tif -ad8 ad8o.tif -o Outletmv.shp
!mpiexec -n 4 threshold -ssa ad8o.tif -thresh {threshold} -src src.tif
!mpiexec -n 4 streamnet -fel fel.tif -p p.tif -ad8 ad8o.tif -src src.tif -ord ord.tif \
                -tree tree.txt -coord coord.txt -net net.shp -w w.tif -o Outletmv.shp -sw
!mpiexec -n 4 dinfflowdir -fel fel.tif -ang ang.tif -slp slp.tif
!mpiexec -n 4 dinfdistdown -fel fel.tif -ang ang.tif -src src.tif -dd hand.tif -m ave v
```

Figure 6. Cross-section of the code for the Height Above the Nearest Drainage (HAND) model, which takes a user argument for different levels of *threshold* value, is shown.

and dependency files it uses during its execution, and the output files it generates. Finding these differences is often crucial for interpreting and analyzing a model.

Figure 6 shows a cross-section of the code from the TauDEM HAND procedure[7] given in the steps above. The *threshold* parameter is provided by the user. We run execute the HAND model with Sciunit using two different threshold values of 5000 and 50,000, each with a different Python script. As a result, we get two separate executions: namely, e1 and e2. Figure 7 shows a simplified cross-section of the result of the Diff command on these executions. The files different in e1 and e2 are their respective code files having different *threshold* values. Notice that the output files *hand.tif*, *src.tif*, and *srctemp.tif* are present in both executions but differ in their content due to the change in *threshold* value. *cde.log* and *provenance.cde-root.1.log* are some of the internal files utilized by Sciunit for containerizing these programs. The ellipses in the output represent parts of the output suppressed for readability.

5.4.2. Service Interactions

Before the program for the HAND model in Figure 6 is executed, it downloads a resource from HydroShare to run the experiment. The resource contains the Little Bear River data set to use in this experiment. This behavior is common in modeling experiments as they often rely on data and other resources residing on other hosts. This is shown in the first few lines of our code for the HAND model experiment in Figure 8.

The content of remote repositories may change over time, affecting the reproducibility of any analyses that depend on the downloaded data. In our scenario, let us consider that when the program is executed the first time, the network is accessible and the data set is downloaded successfully. After downloading the data set, the rest of the HAND code is executed. After some time, if the user decides to repeat the initial execution, the data set may not be available, or the network itself may not be accessible. In that case, a fresh copy of the data set would not be downloaded. However, the code for calculating HAND would continue to use the stale data set from the initial execution that still resides on the local machine. All of this would occur smoothly for the user running this program, and no error or warning would be generated.

Sciunit traces the entire execution of a program and stores its provenance. Sciunit then performs a *provenance-based diff*

```
> sciunit diff e1 e2

Difference in e1 and e2...
Files only in e1:
HANDWorkflow_threshold_5000.py

Files only in e2:
HANDWorkflow_threshold_50000.py
...

Files with changed size:
cde.log
provenance.cde-root.1.log
hand.tif
src.tif
srctemp.tif
...

Files with changed modified time:
...

Files with changed permissions:
None
```

Figure 7. The output of the Diff command used in the Height Above the Nearest Drainage (HAND) model is shown, using different values of thresholds. e1 is the execution with threshold 5000, and e2 is the execution with threshold 50,000.

[7]The HAND resource and sciunit is accessible from here: https://www.hydroshare.org/resource/ce0f48aff42142b3a12147e8c5113dac/.

```
!pip install git+https://github.com/hydroshare/hsclient.git
from hsclient import HydroShare
hs = HydroShare()
hs.sign_in()
# Get the resource you want to download using its identifier
res_id = "64209b842b5d45b3867a194806b29207"
res = hs.resource(res_id)
res.download()
!unzip {res_id}.zip
!cp {res_id}/data/contents/LittleBear.zip .
```

Figure 8. The first few lines of code of the Height Above the Nearest Drainage (HAND) model show the HydroShare resource being downloaded from the internet.

operation: it compares the original and the repeat executions using the provenance information collected for both. In the example described above for downloading the data set, the sequence of system calls for the original and repeat executions would differ since the data set was not downloaded and stored successfully in the repeat execution. Hence, the provenance trace for these executions diverges. Sciunit captures this difference and alerts the user that the repeat execution of the given program was not completely successful.

5.5. Geoscience Use Case 5: RHESSys

Regional Hydro-Ecological Simulation System (RHESSys) is a GIS-based terrestrial hydro-ecological modeling framework designed to simulate carbon, water, and nutrient fluxes (Tague and Band, 2004). It models the temporal and spatial variability of ecosystem processes and interactions at the watershed scale. RHESSys combines the physically based process models and a methodology for partitioning and parametrizing the landscape.

We demonstrate the end-to-end workflow of the RHESSys model using PyRHESSys, which is an object-oriented Python wrapper for RHESSys. PyRHESSys enables users to create and manipulate model input, execute the model, and analyze the model outputs. This model workflow demonstrates the application of an open and interoperable containerization approach from a hydrologic modeler's perspective. The steps are as follows:

(1) We create an end-to-end workflow for RHESSys using a Jupyter notebook.
(2) We encapsulate the workflow and create configurations that include lists of encapsulated dependencies generated using Sciunit.
(3) We create two GitHub repositories to share dependency files for evaluating the reproducibility of the immutable and interoperable Sciunit container.
(4) We create a HydroShare resource to share the Sciunit container.
(5) Finally, we evaluate the reproducibility of Sciunit using the two GitHub repositories in MyBinder.

To model the workflow, we first download raw GIS and time-series data. GIS data include data for DEM, landcover, soil, etc., and time-series data include climate and streamflow observations, etc. An overview of the code is as follows[8]:

(1) Download raw GIS and time-series data.
(2) Set up GRASS data set and environment.
(3) Prepare spatial input and time-series model input.
(4) Define RHESSys model input and parameters.
(5) Execute RHESSys model using Sciunit.
(6) Create two GitHub repositories and a HydroShare resource.
(7) Create MyBinder environment.

We successfully run this model using Sciunit, which generates an execution e1. To create the MyBinder environment in step 7, we use the `Export` command to generate the requirements .txt file for all Python dependencies for modeling the end-to-end RHESSys workflow:

```
> sciunit export e1
   Exported dependencies of e1 into
requirements.txt
```

Binder resources are configured through an environment.yml file that uses the requirements.txt file generated by Sciunit above.

6. CONCLUSIONS

In this paper, we described the reproducibility requirements of geoscience applications with a focus on model development and analysis. We first outlined the initial design of Sciunit, which has the necessary system capabilities to support the encapsulation and containerization of geoscience applications. However, it lacked the necessary interfaces and components to readily address the reproducibility requirements of geoscience applications. We then described the relevant changes in the interface and architecture of Sciunit to achieve those reproducibility requirements. Though a larger user-based survey of these interfaces is still outstanding, the immediate response from the community has been positive. Scientists have commented on the ease of creating, sharing, and repeating containers with Sciunit. Sciunit could be easily installed and used in HydroShare compute environments (Consortium of Universities for the Advancement of Hydrologic Science, Inc.

[8]The RHESSys end-end workflow and sciunit is accessible from here: https://www.hydroshare.org/resource/d2a469fe56714715bad849a5dfc380bc/.

[CUAHSI] JupyterHub and CyberGIS Jupyter for Water). Like other software, it only requires an occasional upgrade for server maintenance. Sciunit is generic in its design and has also been used in space and solid Earth science applications.

Finally, we would like to emphasize that the resources provided as part of this paper include both geoscience applications and sciunits. Sciunit software guarantees the reproducibility of workflows executed by the Sciunits that are included. However, the reproducibility of geoscience applications used to create the Sciunit containers is dependent on the libraries and configuration available on the computing platform being used. While we have tested all the applications presented in this paper on the CyberGIS Jupyter for Water platform linked to HydroShare, successful re-execution of parts of the code that were used to create the Sciunit containers—linked in this paper for documentation and pedagogical purposes—cannot be guaranteed for computational environments that differ from CyberGIS Jupyter for Water as it was configured at the time of the publication. Indeed, containerizing dependencies of the computational environment is one of the problems Sciunit was developed to solve. While code outside of Sciunit may no longer execute correctly, the use cases presented in this paper demonstrate the reproducibility guaranteed by Sciunit through the repeatability of Sciunit containers.

ACKNOWLEDGMENTS

The authors acknowledge support for this work from the National Science Foundation under grants NSF ICER-1928288, ICER-1928369, ICER-1928315, ICER-1639759, ICER-1661918, ICER-1440327, and ICER-1343816. They also acknowledge support from the U.S. Department of Energy Better Scientific Software Fellowship, DePaul Seed Grants. Finally, they thank Binata Roy and Iman Maghami at the University of Virginia for their help with testing some of the use cases.

REFERENCES CITED

Ahmad, R., Deeds, M., Nakamura, Y., Ton That, D.H., and Malik, T., 2020, Explaining and replaying containers using provenance, *in* Proceedings, International Provenance and Annotation Workshop, https://bitbucket.org/depauldbgroup/pw20-sciunitdemo/src/master/.

Bakker, M., Post, V., Langevin, C.D., Hughes, J.D., White, J.T., Starn, J.J., and Fienen, M.N., 2016, Scripting MODFLOW model development using Python and FloPy: Ground Water, v. 54, p. 733–739, https://doi.org/10.1111/gwat.12413.

Billah, M.M., Goodall, J.L., Narayan, U., Essawy, B.T., Lakshmi, V., Rajasekar, A., and Moore, R.W., 2016, Using a data grid to automate data preparation pipelines required for regional-scale hydrologic modeling: Environmental Modelling & Software, v. 78, p. 31–39, https://doi.org/10.1016/j.envsoft.2015.12.010.

Choi, Y.-D., Goodall, J.L., Sadler, J.M., Castronova, A.M., Bennett, A., Li, Z., Nijssen, B., Wang, S., Clark, M.P., Ames, D.P., Horsburgh, J.S., Yi, H., Bandaragoda, C., Seul, M., Hooper, R., and Tarboton, D.G., 2021, Toward open and reproducible environmental modeling by integrating online data repositories, computational environments, and model Application Programming Interfaces: Environmental Modelling & Software, v. 135, https://doi.org/10.1016/j.envsoft.2020.104888.

Clark, M.P., Nijssen, B., Lundquist, J.D., and 10 others, 2015a, A unified approach for process-based hydrologic modeling: 1. Modeling concept: Water Resources Research, v. 51, p. 2498–2514, https://doi.org/10.1002/2015WR017198

Clark, M.P., Nijssen, B., Lundquist, J.D., and 12 others, 2015b, A unified approach for process-based hydrologic modeling: 2. Model implementation and case studies: Water Resources Research, v. 51, p. 2515–2542, https://doi.org/10.1002/2015WR017200.

Essawy, B.T., Goodall, J.L., Zell, W., Voce, D., Morsy, M.M., Sadler, J., Yuan, Z., and Malik, T., 2018, Integrating scientific cyberinfrastructures to improve reproducibility in computational hydrology: Example for HydroShare and GeoTrust: Environmental Modelling & Software, v. 105, p. 217–229, https://doi.org/10.1016/j.envsoft.2018.03.025.

Garousi-Nejad, I., Tarboton, D.G., Aboutalebi, M., and Torres-Rua, A.F., 2019, Terrain analysis enhancements to the height above nearest drainage flood inundation mapping method: Water Resources Research, v. 55, no. 10, p. 7983–8009, https://doi.org/10.1029/2019WR024837.

Harbaugh, A.W., 2005, MODFLOW-2005, The U.S. Geological Survey modular ground-water model—The ground-water flow process: U.S. Geological Survey Techniques and Methods 6-A16, https://doi.org/10.3133/tm6A16.

Liang, X., Lettenmaier, D., Wood, E., and Burges, S., 1994, A simple hydrologically based model of land surface water and energy fluxes for general circulation models: Journal of Geophysical Research: Atmospheres, v. 99, p. 14,415–14,428, https://doi.org/10.1029/94JD00483.

McDonald, M.G., and Harbaugh, A.W., 1988, A Modular Three-Dimensional Finite-Difference Ground-Water Flow Model: U.S. Geological Survey Techniques of Water-Resources Investigations 06-A1, https://doi.org/10.3133/twri06a1, 586 p.

Niswonger, R.G., Panday, S., and Ibaraki, M., 2011, MODFLOW-NWT, A Newton formulation for MODFLOW-2005: U.S. Geological Survey Techniques and Methods 6-A37, 44 p.

Nobre, A.D., Cuartas, L.A., Hodnett, M., Rennó, C.D., Rodrigues, G., Silveira, A., Waterloo, M., and Saleska, S., 2011, Height above the nearest drainage—A hydrologically relevant new terrain model: Journal of Hydrology, v. 404, p. 13–29, https://doi.org/10.1016/j.jhydrol.2011.03.051.

Pham, Q., Malik, T., and Foster, I., 2013, Using provenance for repeatability, *in* Proceedings, USENIX Workshop on the Theory and Practice of Provenance, 5th, Lombard, Illinois: Berkeley, California, USENIX, article 2, p. 1–4, https://dl.acm.org/doi/10.5555/2482949.2482952

Rennó, C., Nobre, A.D., Cuartas, L.A., Vianei Soares, J., Hodnett, M.G., Tomasella, J., and Waterloo, M.J., 2008, HAND, a new terrain descriptor using SRTM-DEM: Mapping terra-firme rainforest environments in Amazonia: Remote Sensing of Environment, v. 112, p. 3469–3481, https://doi.org/10.1016/j.rse.2008.03.018.

Stagge, J.H., Rosenberg, D.E., Abdallah, A.M., Hadia Akbar, H., Attallah, N.A., and James, R., 2019, Assessing data availability and research reproducibility in hydrology and water resources: Scientific Data, v. 6, p. 1–12, https://doi.org/10.1038/sdata.2019.30.

Tague, C.L., and Band, L.E., 2004, RHESSys: Regional Hydro-Ecologic Simulation System—An object-oriented approach to spatially distributed modeling of carbon, water, and nutrient cycling: Earth Interactions, v. 8, no. 19, p. 1–42, https://doi.org/10.1175/1087-3562(2004)8<1:RRHSSO>2.0.CO;2.

Tesfa, T.K., Tarboton, D.G., Watson, D.W., Schreuders, K.A.T., Baker, M.E., and Wallace, R.M., 2011, Extraction of hydrological proximity measures from DEMs using parallel processing: Environmental Modelling & Software, v. 26, no. 12, p. 1696–1709, https://doi.org/10.1016/j.envsoft.2011.07.018.

Ton That, D.H., Fils, G., Yuan, Z., and Malik, T., 2017, Sciunits: Reusable research objects, *in* Proceedings, IEEE International Conference on e-Science, 13th, Auckland, New Zealand: New York, IEEE, p. 374–383, https://doi.org/10.1109/eScience.2017.51.

Wilkinson, M., et al., 2016, The FAIR guiding principles for scientific data management and stewardship: Scientific Data, v. 3, 160018, https://doi.org/10.1038/sdata.2016.18.

Yuan, Z., Ton That, D.H., Kothari, S., Fils, G., and Malik, T., 2018, Utilizing provenance in reusable research objects: Informatics, v. 5, no. 1, p. 1–14, https://doi.org/10.3390/informatics5010014.

Zheng, X., Tarboton, D.G., Maidment, D.R., Liu, Y.Y., and Passalacqua, P., 2018, River channel geometry and rating curve estimation using Height Above the Nearest Drainage: Journal of the American Water Resources Association, v. 54, no. 4, p. 785–806, https://doi.org/10.1111/1752-1688.12661.

Manuscript Accepted by the Society 17 March 2022
Manuscript Published Online 22 December 2022
Printed in the USA