Metadata practices for simulation workflows

Jose Villamar [1,2,*], Matthias Kelbling [3], Heather L. More [1,4], Michael Denker [1], Tom Tetzlaff [1], Johanna Senk [1,5], and Stephan Thober [3]

[1]Institute for Advanced Simulation (IAS-6), Jülich Research Centre, Jülich, Germany
[2]RWTH Aachen University, Aachen, Germany
[3]Department of Computational Hydrosystems, Helmholtz-Centre for Environmental Research, Leipzig, Germany
[4]Institute for Advanced Simulation (IAS-9), Jülich Research Centre, Jülich, Germany
[5]Sussex AI, School of Engineering and Informatics, University of Sussex, Brighton, United Kingdom
[*]corresponding author: Jose Villamar (j.villamar@fz-juelich.de)

ABSTRACT

Computer simulations are an essential pillar of knowledge generation in science. Understanding, reproducing, and exploring the results of simulations relies on tracking and organizing metadata describing numerical experiments. However, the models used to understand real-world systems, and the computational machinery required to simulate them, are typically complex, and produce large amounts of heterogeneous metadata. Here, we present general practices for acquiring and handling metadata that are agnostic to software and hardware, and highly flexible for the user. These consist of two steps: 1) recording and storing raw metadata, and 2) selecting and structuring metadata. As a proof of concept, we develop the *Archivist*, a Python tool to help with the second step, and use it to apply our practices to distinct high-performance computing use cases from neuroscience and hydrology. Our practices and the *Archivist* can readily be applied to existing workflows without the need for substantial restructuring. They support sustainable numerical workflows, facilitating reproducibility and data reuse in generic simulation-based research.

1 Introduction

Recent advances in high-performance computing (HPC) technology enable simulations of increasingly large and complex models, which offer huge potential for science and society but are more challenging to accurately reproduce, share, and interpret. These difficulties make user stories such as the following common among researchers:

1. *Scientist X cannot reproduce the results of scientist Y due to a lack of information on software dependencies and inconsistencies between the information provided in the article and in the code published by Y. Even personal communication with Y does not resolve these inconsistencies.*[1]

2. *Each member of a group of scientists regularly runs simulations of the same mathematical model to investigate different scientific questions. Because the data generated by each simulation is similar and could be useful to others, sharing it is desirable to minimize time and energy costs. However, the scientists have no efficient way of communicating the information necessary to understand the structure of each dataset and the details of how it was generated*[2].

3. *A group of scientists is developing simulation software. After each development cycle, the group runs a set of benchmarking experiments with different configurations and models to continuously monitor software performance ("continuous benchmarking"*[3]*). After years of development, the group has accumulated large amounts of benchmarking data for each software version. While exploring the data for an individual software version is simple, the scientists have no efficient way of comparing versions which use similar configurations or model types.*

One would assume that the above challenges should not exist in simulation science, due to our perception of full control over digital implementations[4]. However, this assumption is wrong for two reasons. First, users often are not aware of every aspect of their hardware and software systems, for example low-level hardware settings[5], implementation details of a software package, or the implications of using a specific operating system[6]. These aspects are of particular importance for HPC-enabled simulations, where highly specialized hardware and software solutions and the distribution of code among multiple processors may affect the exact simulation outcome or performance measures. Given the limited lifetime of HPC systems, reproducibility may be practically impaired. Second, users often customize aspects of their system without properly documenting their changes, for example modifying the behavior of off-the-shelf software. Therefore, despite the digital nature of simulation research, it remains difficult to acquire and organize metadata describing the details of hardware systems, software stacks,

model descriptions and simulation workflows that are necessary to replicate, understand, and share numerical experiments[1,7–9]. These metadata arise at all stages of the simulation workflow, including defining and implementing a model; preparing software; generating and executing a job; post-processing, analyzing, and visualizing data; and organizing and storing data. Successfully capturing comprehensive metadata and provenance information along the processing chain ensures reproducibility and replicability[7,8,10–12], allows assessment of simulation outcomes[1,13], and helps scientists explore, share, and reuse data[14].

Recently, several powerful tools have been developed to organize, execute, and track complex workflows, including their resulting data and metadata, such as Sumatra[10], AiiDA[15], Snakemake[16], and Datalad[17]. Although these tools are useful for new projects, integrating them into existing workflows requires major restructuring. As a consequence, scientists often write custom code for metadata handling to compromise between adding functionality and keeping an established workflow. The resulting inconsistency in metadata management makes it difficult to transfer knowledge between researchers.

Here, we propose a set of practices for flexible metadata management which can readily be integrated into existing simulation workflows without substantial refactoring. The practices are generic and apply to diverse research fields. They advise researchers about which metadata to collect at each stage of a simulation workflow, how to collect them, and how to process the metadata so that they enrich data. We further describe the Python tool *Archivist* which helps selecting and structuring metadata. Eventually, we apply the proposed practices and the *Archivist* to a minimal illustrating example, and to two real-world use cases from neuroscience and hydrology.

2 Metadata management practices

To better understand the metadata collection and handling process, we consider a generic knowledge production workflow divided into three sub-workflows (Fig. 1): a simulation experiment containing processes generating data and metadata, metadata post-processing where heterogeneous metadata is processed into meaningful formats and structures, and usage of enriched data. Although the implementation of these processes can and will vary across different use cases, this generic knowledge production workflow encompasses archetypes of the components occurring in many implemented workflows.

A generic simulation experiment comprises a sequence of steps (Fig. 1, *Simulation* section). We define these steps by abstracting the processes performed in two domain-specific workflows from neuroscience and hydrology[14,18]. Depending on the complexity of the preparation required before and after the execution of the simulation there might be additional steps, but at minimum we consider the necessary steps to be configuration of the software environment, simulation engine, and model; execution of the simulation; and storage of the generated data and metadata. For clarity we separate data and metadata storage in Fig. 1, however this is an implementation choice left to the discretion of the user and for the remainder of this paper we refer to both storages together as one collection for simplicity.

After the simulation experiment, users can analyze the resulting data and metadata collection to gain knowledge on the simulated model or simulation technology. The methods through which data can be analyzed are tightly related to the objectives and intentions of the user. Furthermore, in some disciplines, these methods may require a complex workflow and perhaps even the use of High-Performance Computing (HPC) resources. Since we cannot exhaustively define the different existing strategies for processing data, we provide an abstract description of how the generated data can be exploited (Fig. 1, *Data usage* section). For instance, data is analyzed in order to measure metrics related to model predictions or accuracy, or simulation performance. Graphical visualizations are generated for easier human interpretation. Additionally, differences between multiple versions of simulators, models or parameters are obtained by comparing with previous results. New insights are shared with peers to disseminate knowledge.

As processing the data may inadvertently remove or add specific features that alter the original information, storing the original data allows independent users to exploit individual datasets with different methods, or to process multiple datasets at once, which increases re-usability of the simulation results. Over time the data and metadata generated by different simulation experiments will be compiled and stored in a continuously growing collection. Establishing a data model is necessary to maintain the organization and integrity of this collection. A data model is an abstract representation of data objects and their relationships that organizes relevant attributes of data and standardizes how they relate to each other. This serves as a blueprint for structuring data elements and their relationships, providing a clear understanding of how data is organized, stored, and utilized within a collection. Without the data model, users may not be able to establish an overview of the actual contents and may fail to efficiently explore the collection.

To implement and perform this diversity of data usage tasks metadata is required to properly describe the simulation experiment. When exploring multiple sets of generated data, metadata categorizing each dataset allows researchers to efficiently differentiate them. Moreover, when sharing the results, metadata enhances the knowledge transfer between data users by detailing what, how, and where the experiment was performed. Most of this metadata, can only be found when performing the experiment and can exist at any step of the simulation (Fig. 1, blue *Data generation* boxes). The collection of metadata and their interpretation often varies and evolves with the systems and software used (Fig. 1, yellow *Raw metadata collection* boxes). We describe below some general practices for successfully managing the variety of metadata that arises during simulation.

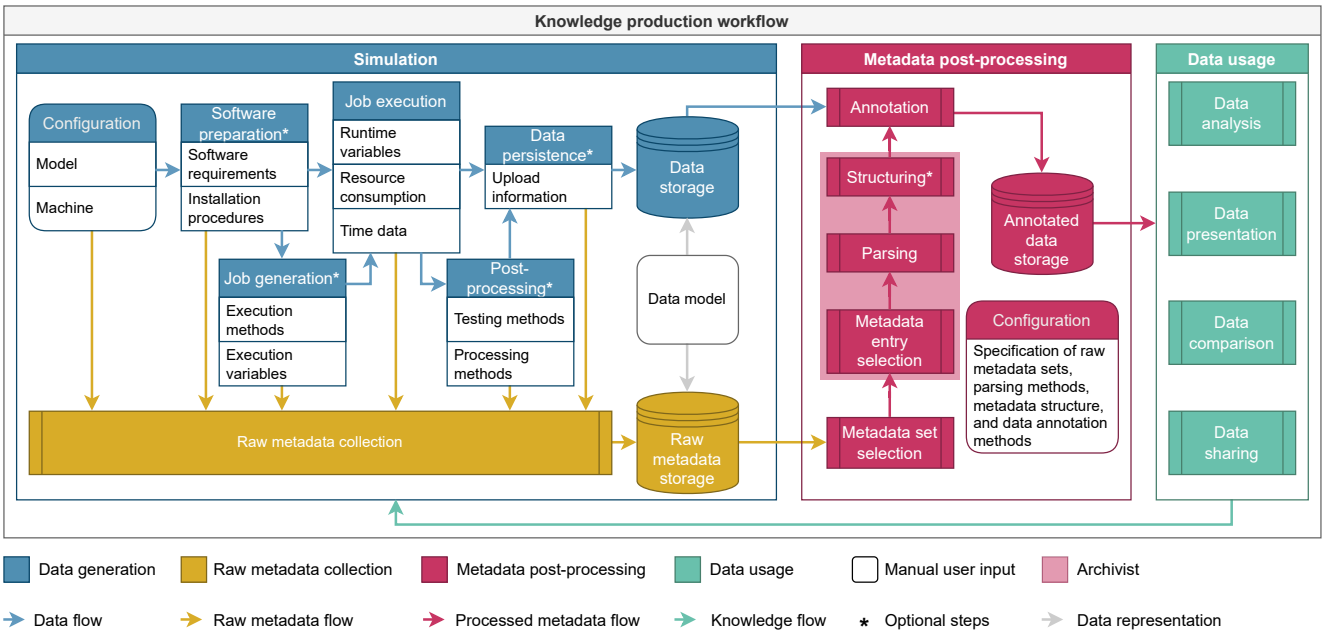

Figure 1. Metadata management in a generic knowledge production workflow. We conceptually divide the workflow into three consecutive sub-workflows: simulation, metadata post-processing, and data usage. **Simulation** collects raw metadata (yellow) at each step (blue), then stores it along with a data model describing the structures of data and raw metadata. **Post-processing** (red) selects raw metadata, then parses and structures it; annotation links this processed metadata to the data. During post-processing, user-defined configuration specifies which raw metadata to select and how to process it, ensuring the final annotated data is suitable for its application. Some of these steps (light red) can be performed by the post-processing framework presented in "Metadata post-processing framework – the *Archivist*". **Data usage** (green) analyzes data to draw conclusions through presentation and comparison; data may also be shared with others. Ultimately, the results of data usage inform subsequent simulations (green arrow). Each sub-workflow consists of autonomous steps (rectangles), user-configured steps (rounded rectangles), data storage (cylinders), and data transfer (arrows). This diagram aims to represent generic simulation-based research; specific workflows may omit some elements (asterisks).

Collect as much information as possible. Metadata are data that describe data. Here, we focus on information that can be directly collected in the environment where the experiment takes place (sometimes termed "hard" metadata[20]). However, the definition of what precise pieces of information should be represented as metadata is difficult to generalize. Each experiment generates an arbitrary amount of information – separating this information into metadata and data depends on the nature and goal of the workflow itself. For instance, if the goal of a simulation workflow is to collect and analyze the data generated by the underlying model, then the performance of the simulation (time to solution, memory consumption, etc.) would be part of the metadata. On the contrary, if the goal of the workflow is to benchmark the simulator then the data generated by the model is of secondary importance, as the recorded performance data would be the desired primary result. Limiting the amount of information collected during a simulation experiment restricts how its output can be used in the future. Therefore, to maximize the potential for reuse of shared experiment results, it is important to collect as much information as possible[10]. Yet, the full extent of this collection needs to be determined on a case by case basis. Sometimes, collecting the full breadth of information available can be redundant. For instance, when simulation models grow complex, the number of available parameters grows in volume too. Well-documented parameter sets often use default values suitable for their corresponding model; during experiments, only a subset of these may actually be modified. In this case, collecting only the modified parameters may be sufficient. Conversely, there may be instances where keeping detailed track of the differences between experiment is necessary. Experience has shown, for example, that differences in software versions may lead to discrepancies in experiment results[1]. In the case of rapidly evolving software such as the model, simulator, or their library dependencies, it is beneficial to keep track of the different versions used for each experiment. As a compromise, we propose collecting information as sustainably as possible to an extent where all currently known user needs are satisfied.

Identify metadata sources. Given the rapidly evolving software and hardware platforms available for simulation alongside user specific set-ups, it is impossible to exhaustively list all collectible metadata. Instead, we categorize the different sources of

metadata according to each step of the simulation experiment (Fig. Fig. 1, blue *Data generation* boxes):

- *Configuration*: Most steps during the workflow need some kind of configuration or parametrization, which the user often saves and edits in files beforehand. These files are a type of metadata describing each step of the simulation. This metadata includes, in particular, the configuration and parameters used by the model and simulator.
- *Software preparation*: In this step all the required software (libraries, tools, and programs) are fetched, if needed compiled, and loaded into the environment. Information to collect here includes the source of the software, the parameters related to compilation, and the environment variables defined when loading the software.
- *Job generation* (if applicable): In HPC scenarios, job schedulers are generally used to efficiently allocate and share resources of compute clusters. Collectable metadata here include job parameters: machine name, version and partition used, amount and type of resources used, resource configuration such a process binding or thread pinning, and job time limit.
- *Job execution*: The main part of the simulation, the execution involves instantiating the simulation model and running the simulation. It is at this step that recording the status of the execution environment itself is the most interesting and least redundant. In contrast to *Software preparation*, the metadata collected in this step should focus on the environment variables, libraries, and processes present during execution. Additionally, execution time, resource consumption (memory, storage, network, etc.), and system load can only be recorded at this step.
- *Post-processing* (if applicable): Depending on the goal and the implementation of the workflow, the data that is generated must be post-processed for further use, which may involve steps such as checking, cleaning, compressing or transforming the data. If post-processing is necessary, documenting the procedures performed on data is essential to have an overview of how the final result was obtained. Thus the information on the different methods used during this step must be recorded.
- *Data persistence*: In this step, information on the storing process can be recorded. For example: who generated the data and metadata to be stored? When was it generated and when is it being stored? Where was it generated and where is it being stored to e.g. a file server, a local or remote database? In the case of a file server, the address of the server and the absolute path of the stored files should be recorded. When using a database, new entries are indexed with a unique identifier; storing this identifier allows linking with future related entries and should be recorded correspondingly.

Depending on the system and software used, the metadata present in the above mentioned categories can often be found as parameter or configuration files. In particular, most system information can be obtained through commands provided by the operating system or package management solutions.

Users might be able to recognize the metadata required for a specific data analysis and collect it alongside other metadata they are aware of. However, situations could arise where reusing the generated data for unforeseen purposes is impossible due to missing context. Even after collecting all known metadata, it may be that there are still unknown pieces of metadata that need to be recorded. With rapidly evolving systems and software, this is not an uncommon scenario, and if such a situation arises then users must identify what information is lacking, and add it to the list of metadata to be recorded for future experiments.

Metadata collection does not need to be complicated. There are two possible ways of collecting metadata: retrieval by a dedicated parallel process “pulling” the relevant pieces of information from each simulation step, or autonomous “pushing” by each component to a metadata storage. The pull-based approach relies on an overarching metadata-monitoring module that is adapted to the specific simulation. The push-based approach, in contrast, does not require such infrastructure at run time. At each step, the governing process simply dumps its metadata in some arbitrary, more or less raw format, independent of other simulation steps. The push-based approach is therefore much more flexible and can readily be applied to existing simulations. With each component presenting its own metadata it is likely that the final collection of metadata contains different structures and formats. Although it is possible to standardize the presentation of metadata across all components, this process would require transformations that are dependent on the objective for which the data is being generated, and can vary from user to user, or interoperability standard that the researcher must adhere to. In some situations, transformations of metadata may even depend on other metadata, making it difficult to perform an ad-hoc transformation process. We therefore propose extracting and structuring the raw metadata in a separate step after the simulation. This presents a clear advantage by decoupling metadata collection processes from user goals and processing software. Collecting as much metadata as possible is essential as it is impossible to get the corresponding metadata after a simulation is performed. Thus, gathering the metadata regardless of format and storing it during simulation for later processing is an efficient way to keep track of the information describing the experiment. We call the collection of these unprocessed metadata files the "raw" metadata (Fig. 1, yellow *Raw metadata collection* boxes).

Post-process recorded metadata according to your goals. Transforming the heterogeneous raw metadata into a unified comprehensive format is essential in order to be able to exploit it alongside the data. Given that each workflow is run with a specific objective in mind, the collection of raw metadata contains often much more than needed. Although this is desirable to maximize possible reusability of simulation data, for a given objective the raw metadata should be filtered. For this, a dedicated read operation that extracts the relevant entries from a collection of metadata is performed according to given objective. The extracted metadata is parsed and might be immediately usable depending on the output format of the parsing operations, however a specific structure might be needed to increase interoperability between the processed metadata and other data exploitation software. Therefore, structuring the metadata according to a standard is often a beneficial step to easily and unambiguously leverage the metadata information. With this, the previously heterogeneous metadata collection is now unified, possibly structured according to a standard, and ready to be exploited.

Combine your data with metadata. Before being able to use processed metadata and data in unison these need to be combined together. Here we refer to combining as a referencing procedure in which any data entity that makes up a complete data record is cross-referenced with any metadata (item or collection) that assists in using and interpreting the data entity. Although it is possible to exploit metadata and data independently for specific purposes, combining them quickly becomes necessary when the volume and dimension of a dataset starts to grow. When amassing data from multiple simulations, each with varying degrees of different configuration and or models, it becomes apparent that simulation results need to be explored with a higher level of description. For this, "tagging" or annotating the data with the processed metadata creates an enriched dataset. When appropriately stored, users can exploit indexing engines to navigate through these enriched datasets and generate complex data projections. Again, the level of granularity with which data entities are identified and matched to metadata will need to be decided based on the objective of the workflow.

Following these practices will help in handling large heterogeneous volumes of metadata, and combining datasets with processed metadata increases their explorability and exploitability. The actual methods through which these practices can be followed or implemented depend on the technology used to manage the simulation experiment and data processing pipeline. In particular some existing technologies offer support for these practices and are addressed in the "Discussion".

3 Metadata post-processing framework – the *Archivist*

In the previous section we showed that, to cope with the heterogeneity of collected metadata, it is necessary to perform parsing and possibly structuring operations to obtain a unified, exploitable metadata representation. Although there exist tools capable of doing these kinds of operations, e.g.,AiiDA[15] parsers (https://.readthedocs.io/projects/-core/en/latest/reference/apidoc/.parsers.html, last access: 23 February 2024), DataLad[17] metadata extractors (https://docs.datalad.org/projects/metalad/en/latest/extractors.html, last access: 23 February 2024), these can pose fixed constraints on the processing of the metadata, may be an inseparable component of a larger framework and may not be used independently.

Here we propose the *Archivist* as a framework that can be used to perform parsing and structuring operations. It is capable of individually parsing generic files and combining arbitrary parsed results in a customizable structure. We note that data annotation, i.e., linking metadata records to relevant parts of the data in the post-processing pipeline (Fig. 1, red *Annotation* box), is currently not supported because annotations of data need to be heavily customized to fit the underlying structure of the data and this process is difficult to formalize. For example, while data structured according to HDF5-based[21] formats like the NIX[22] standard can make use of intrinsic mechanisms provided for linking collections of metadata entries to data, this mechanism cannot be generalized to other file formats and data models.

To illustrate the *Archivist* framework, we show a representative example implementation a workflow to parse and structure metadata in Fig. 2. The framework, coded in Python (http://www.python.org, last access: 23 February 2024), is primarily composed of four processing classes and one interface class. The processing classes perform different aspects of the metadata parsing and structuring (exploration of metadata files, parsing of metadata sources, formatting and structuring of collected metadata, and exporting to a selected format), while the central interface class orchestrates the other classes for ease of use.

Below, we introduce the functionalities of each of these classes. Source code, example implementations, and additional details can be found at https://doi.org/10.5281/zenodo.13442425 (last access: 30 August 2024).

Archivist: The *Archivist* class is a convenience interface class which instantiates and orchestrates the processing classes. As input, the class accepts a collection of raw metadata, the parsing operations to apply, and optionally a formatting schema. This interface is necessary because each processing class uses specifically structured inputs and outputs, which can be cumbersome and error-prone for users to generate themselves. We designed the interface to be configurable, giving it enough flexibility to customize the behavior of the processing classes while providing simple inputs and outputs.

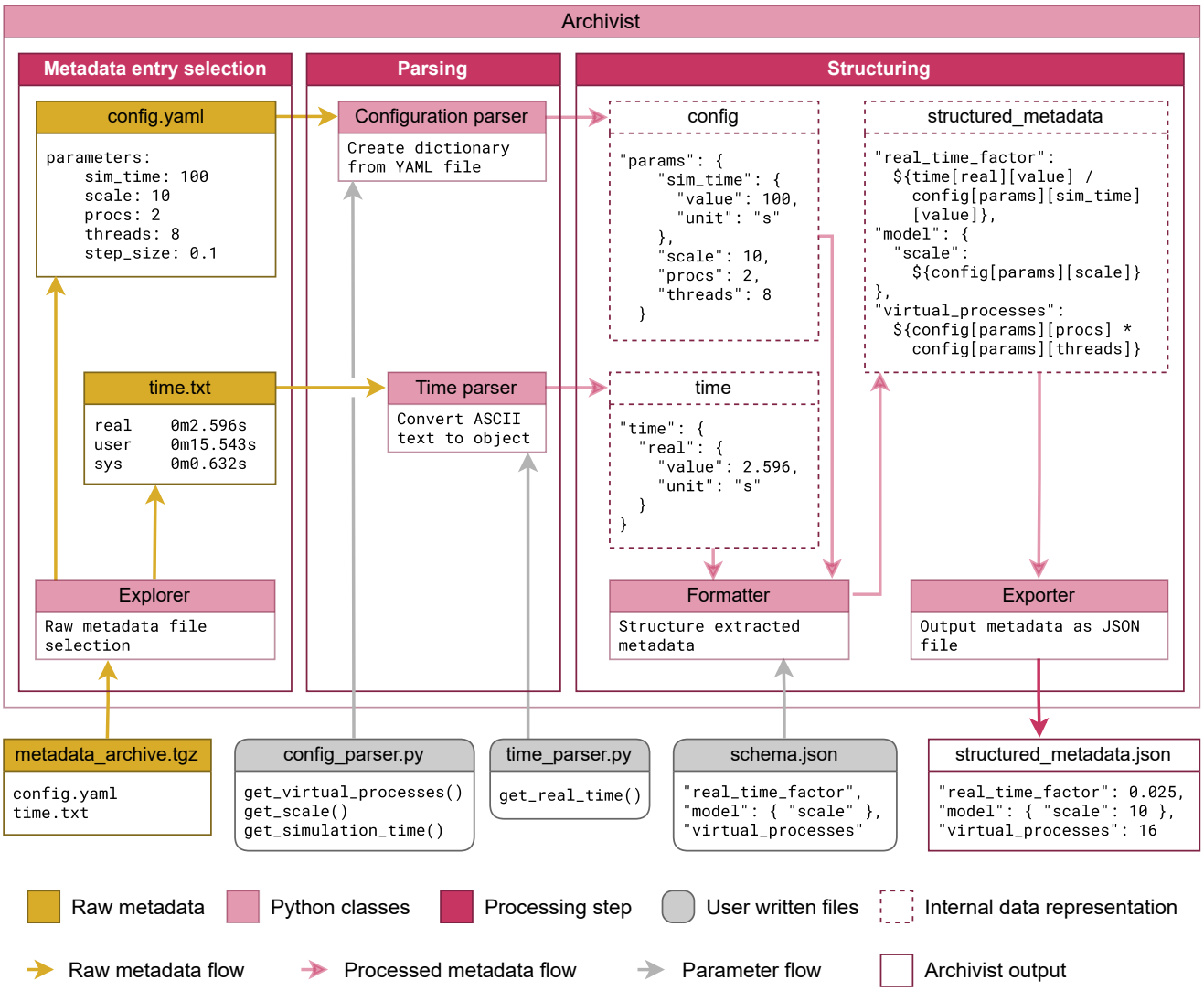

Figure 2. Implementation of an *Archivist* metadata processing pipeline with two example parsers. From the user perspective, the *Archivist* class is provided with all inputs associated with the pipeline. Internally, the *Explorer* class sorts the individual files to process from a collection of raw metadata files, and dispatches them to corresponding *Parsers* classes, here the *Configuration parser* and the *Time parser* classes. Then, each *Parser* class employs a user defined function to extract specific information from its respective files. After this, the *Formatter* class collects the parsing results. If a schema is provided, the composite result can be structured following user design. The final processed metadata is output in a format of choice by the *Exporter* class. Bottom: Legend for different components of the figure.

Explorer: The *Explorer* class processes inputs given to the *Archivist*. These can be raw metadata archives or directories containing raw metadata files. To enhance interoperability, no assumption is made on the structure and contents of the archives or directories. As such the user must define rules to identify which files to parse. To do this, the user can provide precise file names or regular expressions describing these names. We refer to these as file description rules. Using these rules, the explorer searches the input (archive or directory) for corresponding files and provides a list of files to parse.

Parser: The *Parser* is used to gather metadata from a subset of the list of files identified by the *Explorer*. For this, the user must associate one of the file description rules provided to the *Explorer* with a *Parser* instance. Depending on the rule, the *Archivist* instance dispatches each file to the respective *Parser*. The *Parser* is an abstract class designed to be extended by users for their own purposes. Such an extended instance contains a file description rule and an associated parsing method. Although each user must provide their own parsing methods, the extension mechanism allows transparent file parsing regardless of format or structure. Furthermore, with the source code, we provide examples of parsing methods we use for our own workflows and

we hope to build a user base around shared methods to foster reusability and replicability.

Formatter: The *Formatter* combines the output of the Parser instances into a unified metadata file. Although parsing the desired metadata files and listing the results in a single file might be sufficient for some workflows, in other cases it may be necessary to transform the collection of parsing results into a cohesive and comprehensive structure. For this, the user must provide a schema to match a parsing result to the desired structure. We based the format of our schema on the JSON Schema standard[23], and defined additional directives to orchestrate more complex transformations. As a simplified example in Fig. 2, given the parsed information from the configuration file and time file, the user can combine the simulation time with the real time in a single field, as the `real_time_factor`.

Exporter: The *Exporter* saves the internal data representation of the structured metadata to a file with specific output format. Like the *Parser*, though not an abstract class itself, this class was designed to be extended to enable serialization to arbitrary formats. In particular, users define the output format that enhances compatibility with their annotation method of choice. In Fig. 2, the *Archivist* employs the JSON format to export the structured metadata.

Using this class hierarchy, users only need to define the parsing functionalities they are interested in along with the file target rules, and provide a metadata collection to process. The *Archivist* class takes care of coordinating the other classes and generating the output. For more complex operations, users can provide a schema for the *Formatter* class, and extend the *Exporter* class to change the output format. With this flexibility, the *Archivist* framework provides a re-usable parsing and structuring pipeline that can be operated on existing metadata or attached to a workflow for automated post-processing.

4 Examples

To illustrate how the proposed metadata practices ("Metadata management practices") and the *Archivist* framework ("Metadata post-processing framework – the *Archivist*") can be implemented into specific simulation workflows, we here provide a minimal and two real-world examples. The first example ("Minimal example") constitutes a toy workflow applicable to generic simulations where the user conducts a parameter scanning experiment and leverages recorded metadata to identify suitable configurations. The second example ("Neuroscience use case") describes a workflow for the benchmarking and the verification of a specific neuroscience simulation architecture. The third example ("Hydrology use case") showcases a routine procedure for the calibration of a hydrological model.

4.1 Minimal example

The first example mimics a typical problem in computational science (Fig. 3): how to choose the parameters of a given model such that the accuracy of its predictions is high while the time-to-solution is small? In many applications, a typical parameter which improves the model prediction but slows down the simulation is the model size, here referred to as the `scale`. Examples are the number of elements in a finite element simulation, or the number of neurons in an artificial neuronal network. The minimal example described here illustrates how the enrichment of simulation results with processed metadata helps finding an answer to the above question. For simplicity, we assume that the example runs in a local simulation environment where the required software stack is already installed, no job manager is used, and the data is stored locally. In accordance with Fig. 1, we subdivide the entire workflow into the three components: *Simulation*, *Metadata post-processing*, and *Data usage*. In the following, we describe each of these components in detail.

The *Simulation* section consists of the configuration of the model and the simulation architecture, the simulation execution, and the collection of the generated data and metadata (Fig. 3, blue box). Here, we do not further specify the actual structure and dynamics of the underlying hypothetical model. We merely assume that it contains a parameter `scale` representing the model size. During the simulation, the model dynamics is propagated forwards in time with some time resolution (`step_size`), up to a total model time (`sim_time`). We further assume that the simulation architecture is equipped with a parallelization infrastructure, which is parameterized by the total number of processes (`procs`) and the number of computing threads used by each process (`threads`). Model and architecture parameters are stored in a configuration file (`config.yaml`). During each simulation, some primary simulation data (which we do not further specify here) is generated and stored in a data file (`results.dat`). In addition, the simulation scripts monitor the duration of each simulation run with the help of the linux *time* command (`https://www.man7.org/linux/man-pages/man1/time.1.html`, last access: 25 August 2024). It returns three different types of durations: the `real` time represents the actual wall-clock time, i.e., the total time taken by the simulation process from invocation to termination. The `user` and the `sys` times measure the cumulative CPU (thread) time spent in the user and in the kernel mode, respectively. All three times (`real`, `user`, `sys`) are stored in a text file (`time.txt`). The simulation is repeated for a range of `scale`'s, `procs`, and `threads`. At the end of each simulation run, the configuration and the time files are collected in the form of a raw metadata archive (`metadata_archive.tgz`).

During the *Metadata post-processing*, relevant configuration information and simulation times are parsed from the raw metadata archive and combined into a structured metadata file (Fig. 3, red box). For the question at hand, not all metadata

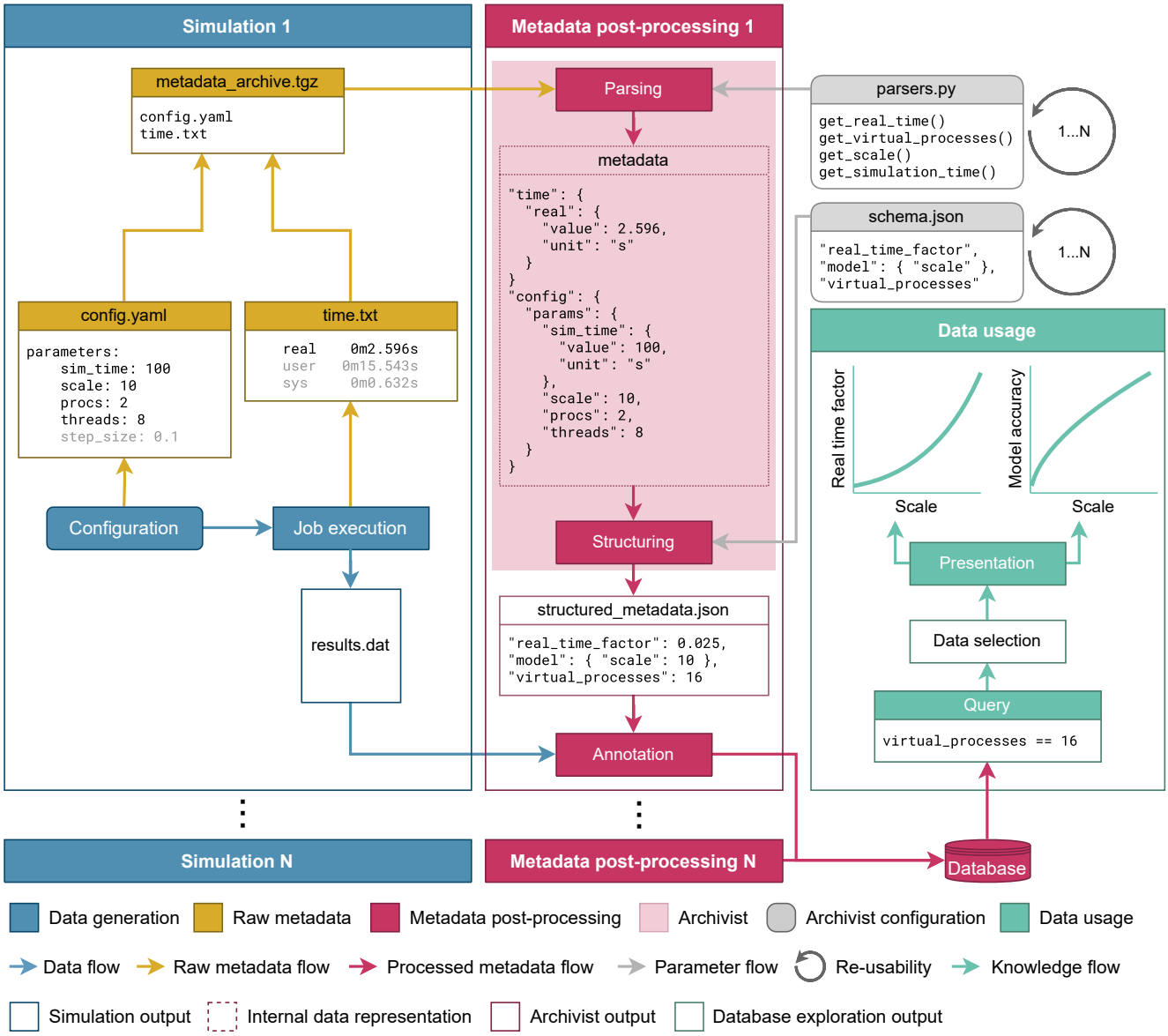

Figure 3. Minimal example. Illustration of the *Archivist*'s functionality in a simple example use case. In a parameter scanning experiment, several instances of a model with different configurations (parameters) are simulated ("Simulation 1",..., "Simulation *N*"; blue boxes on the left). During each simulation, configuration and performance information are recorded and stored in a (raw) metadata archive (yellow). After each simulation, the stored metadata is post-processed ("Metadata post-processing 1",..., "Metadata post-processing *N*; red): first, the relevant information is extracted by calling a predefined parsing script (gray box `parsers.py`). Non-relevant information is discarded (see light gray text in the raw metadata files). The extracted metadata are then structured according to a provided schema (gray box `schema.json`). Finally, the simulation results are annotated with the structured metadata and stored in a database (red cylinder). After all simulations and the metadata post-processing are finished and their results stored in the database, the annotated data can be queried and presented (green).

collected during the simulation needs to be extracted. Let's, for example, assume that the time resolution (`step_size`) was kept constant for all simulations. Moreover, we are only interested in the `real` but not in the `user` and in the `sys` times. During the parsing of the raw metadata, only a subset of metadata is therefore extracted (black items in `config.yaml` and `time.txt` in Fig. 3). In addition, physical units are added to the time quantities. The metadata extraction is performed by the *Archivist* with the help of user-defined parsing methods (`parsers.py`). The *Archivist* further structures the extracted metadata according to a user-defined schema (`schema.json`). In this example, the schema introduces the total number of virtual processes (`virtual_processes`) as the product of the number of processes and the number of threads per process,

as well as the real time factor (`real_time_factor`) as the ratio between the measured wall-clock (`real`) time and the model time (`sim_time`). Finally, the primary simulation data are annotated by the structured metadata. In this example, this annotation is performed by uploading the primary simulation data and the corresponding structured metadata as a single entry to a local data base. An implementation of the metadata post-processing performed for this example can be found at https://doi.org/10.5281/zenodo.13442425 (last access: 30 August 2024).

In the *Data usage* section, the local data base containing the accumulated annotated simulation results is queried to ultimately find model sizes where the model accuracy is sufficiently high while the time-to-solution is low (Fig. 3, green box). To this end, the parameter search is restricted to a subset of simulations where a given amount of computational ressources has been used. In this example, only those entries are extracted from the data base where the total number of virtual processes equals 16. Based on the selected primary simulation data, the user assesses the model accuracy, for example, by comparing the model predictions with some observed or experimental data (for a concrete example, see Sec. 4.3). At the same time, the user extracts the real time factor from the corresponding structured metadata. A subsequent analysis of the dependence of the model accuracy and the real time factor on the model size (`scale`), and an account of additional constraints such as the maximum acceptable real time factor or the minimal acceptable model accuracy, permits an identification of appropriate model sizes.

4.2 Neuroscience use case

Understanding how the brain "computes", what principles it employs to solve complex tasks with minimal energy consumption, how it evolves and changes during the lifetime of an organism, what the origins and effects of neurodegenerative diseases are and what possibilities of treatment exist has huge social, economical and ecological impact. The human brain consists of about 10^{11} nerve cells (neurons)[24], which form a complex network. Each neuron receives inputs from thousands of other neurons, both from the local neighborhood and from distant brain areas. The connectivity structure is highly heterogeneous and depends on the involved neuron types and brain areas. Furthermore, the connections between neurons (synapses) are not static but change depending on sensory inputs and other factors. The mathematical description of the brain's dynamics at cellular resolution therefore involves large sets of coupled differential equations. Even for a single cubic millimeter of brain tissue, this number would be on the order of at least 10^4. Neuroscience is thus dealing with complicated mesoscopic dynamical systems, which are neither small nor in the thermodynamic limit. They can not fully be understood by means of analytical mathematical methods from dynamical systems theory or statistical physics. State-of-the-art neuroscience hence relies on simulation.

Simulating brain-scale neuronal networks at cellular resolution, i.e., instantiating the corresponding models and simulating them in a reasonable time, is challenging. In particular, the investigation of slow biological processes, such as learning or brain development, requires accelerated simulations where the wall-clock times T_{wall} are substantially smaller than the duration T_{model} of the simulated time interval. Due to the large number of neurons and connections, brain-scale neuronal-network simulation requires substantial amounts of memory to store all involved state variables[25]. At each instance of time, large numbers of differential equations have to be solved simultaneously. Brain simulation thus typically employs parallel, distributed computing. One of the key challenges in distributed brain simulation, however, is the communication between neurons, which are typically located on different compute nodes. The development of efficient algorithms and simulation software that can exploit the possibilities of continuously evolving high-performance computing architectures and incorporate new insights from experimental and theoretical neuroscience is hence a fundamental activity in this field[26–34]. It depends on large teams of software developers, who coordinate their work over years of development with many and fast update cycles. Continuously monitoring the quality (correctness, usability, reproducibility) and the performance (speed, memory demands, energy costs) of the simulation code is mandatory[3,35]. Without regularly testing the simulation code, the integration of new features or new optimizations may unknowingly lead to wrong results or performance breakdowns[14]. Detecting and understanding such unwanted behavior, comparing different hardware configurations or testing procedures, and sharing test results with other developers relies on an efficient tracking and organization of the corresponding benchmarking data and metadata.

The use case presented here demonstrates how the proposed metadata managing guidelines and the *Archivist* can help in fulfilling this task. It illustrates a verification and performance benchmarking workflow for the neuronal-network simulation code NEST GPU[36,37] executed on a range of different hardware platforms (Fig. 4). It shows that the simulation code generates identical results on all tested platforms (Fig. 4B), and highlights differences in the simulation speed (Fig. 4C). For illustration, we restrict this example to

- a specific test case: a model[38] of a small piece of the mammalian cortex comprising about 80,000 neurons and 300 million synapses (Fig. 4A),
- a specific set of hardware platforms: four different GPU architectures (see axis labels in Fig. 4B,C),
- a specific verification metric: the average firing rates of neurons in different subpopulations of the network model (Fig. 4B), and

- a specific performance metric: the real time factor, i.e., the ratio between the wall-clock time and the simulated biological time Fig. 4C).

Details on each of these aspects are given in "Methods". In a real performance-benchmarking setting, the same workflow would be (and has been[14,29,37,39–43]) executed for a broader range of test cases, hardware configurations, verification metrics, and performance metrics.

The entire pipeline underlying this use case is implemented using Snakemake[16] – a lightweight yet powerful and flexible workflow manager. Individual workflow components, such as the software deployment and compilation, the simulation execution, the storage of simulation data and raw metadata, the data and metadata post-processing, as well as the data exploration and presentation, correspond to specific Snakemake rules (for details, see "Methods"). During each execution of the simulation workflow, the (probabilistic) network model is simulated with 10 different random realizations of the initial state and the network connectivity (different random-number-generator [RNG] seeds). Each simulation instance generates single-neuron activity traces (spike trains) as well as population and time averaged firing rates as the primary neuroscientific data (blue cylinder in Fig. 4A). In addition, information about the model parameterization, RNG seeds, the hard- and software configuration, as well as the wall-clock times and real time factors are stored in various files and formats as raw metadata (yellow cylinder in Fig. 4A). After the simulation, the data and raw metadata are compressed, labeled with a unique identifier (uid), and uploaded to an instance of a Mongo database (MongoDB, https://www.mongodb.com, last access: 23 February 2024). During the metadata post-processing (red gear in Fig. 4A), the *Archivist* uses specific parsers to extract the type of the computing platform, the RNG seed, and the real time factor from the raw metadata archives. Subsequently, the extracted information is structured according to a user-specific schema. The link between the data and the metadata (data annotation) is established by attaching the data uid to the structured metadata. At the end of the metadata post-processing, the structured metadata are uploaded to the database containing the data and raw metadata (red cylinder in Fig. 4A). The workflow is executed on four different GPU platforms, each sweep individually enriching the same database with its raw and structured metadata and annotated data. The database containing the accumulated data and metadata from various instantiations of the simulation and metadata post-processing workflow can be efficiently queried according to user interest (curved red arrows in Fig. 4A). For the verification and the performance assessment of the NEST GPU code, we extract the average firing rates and the real time factors, respectively, for each network realization (RNG seed) and computing platform, and plot the corresponding across-trial averages and standard deviations (Fig. 4B,C).

The source code underlying the workflow of this example can be found at https://doi.org/10.5281/zenodo.13585723 (last access: 30 August 2024).

4.3 Hydrology use case

The simulation of the hydrologic cycle is fundamental in environmental modeling. The movement and storage of water in the terrestrial system includes diverse processes. Key processes are the infiltration of precipitation into the subsurface, the storage of water in soils and the removal of water from the soil via plant transpiration and subsurface runoff. A reliable estimation of water fluxes and storages is relevant for a wide range of sectors, like drinking water supply, agriculture, forestry, energy production, and transportation. Hydrologic models simulate hydrologic state variables and fluxes and their interaction. These models are created for different purposes according to the needs of specific sectors outlined above. These range from infrastructure planning for drinking water supply to drought quantification for agriculture and to climate projections for future water management. Hydrologic models represent fundamental processes like snow accumulation and melting, soil infiltration, evaporation and plant transpiration, surface and subsurface runoff, and river routing. Typical output variables are river discharge, evapotranspiration and soil moisture among others. Input variables are precipitation and air temperature. There are a wide range of computational workflows given the diverse purposes that hydrologic models are used for. One of the most complex workflows are related to climate change projections where input variables are taken from different sources like climate model ensembles[44]. Currently, metadata tracking in such workflows is often very limited and does not contain the comprehensive settings that are necessary to execute these workflows.

Here, we show how the proposed metadata managing guidelines and the *Archivist* can be applied to a workflow using a hydrologic model and which results can be derived from these (Fig. 5). On the left, a hydrologic model is visually represented by the logo of the mesoscale Hydrologic Model (mHM[18,45,46]) in Fig. 5A. Both, output variables and metadata information represented by the blue and yellow cylinder, respectively, are stored. In the hydrology use case, it is worth noting that the model output is orders of magnitude larger in terms of size than the associated metadata. The output variables are user-specified and contain a minimal set of metadata like, unit and creation date. The metadata managing guidelines presented here does also allow us to create a set of additional metadata that provide comprehensive information about the workflow at hand (yellow cylinder in Fig. 5). For example, the entire configuration of the hydrologic model and its parameters can be recorded as well as the execution environment, version information, and all inputs and outputs. The *Archivist* is then able to process the metadata information in a user-specific way. Two potential applications of the *Archivist* are shown in Fig. 5B and Fig. 5C.

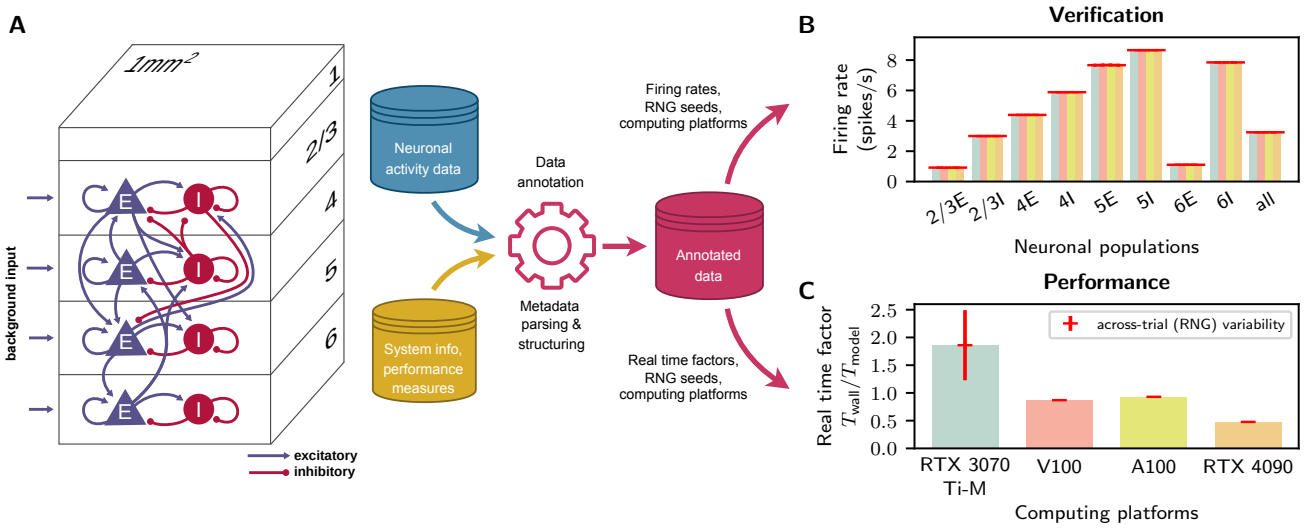

Figure 4. Neuroscience use case. A) Left: Sketch of a model[38] describing the activity dynamics generated by a local neuronal circuit of the mammalian neocortex (adapted from [40]). The network model is composed of four excitatory (E; blue triangles) and four inhibitory neuronal populations (I; red circles), distributed across four cortical layers 2/3, 4, 5 and 6, and driven by background inputs. Neurons in the network are interconnected in a cell-type and layer specific manner (blue and red arrows). Right: The model generates neuronal activity data ("spikes" and firing rates) as the primary neuroscientific data (blue cylinder). For each simulation instance, information about the model parameterization, random number generator (RNG) seeds, the hard- and software configuration, as well as the wall-clock times are stored in various files and formats as raw metadata (yellow cylinder). In a subsequent post-processing step (red gear), the metadata is parsed and structured by the *Archivist*. The simulation data is annotated with this structured metadata and stored in a database for further usage (red cylinder). The database can flexibly be queried according to user interests (curved red arrows). **B, C)** Verification (B) and performance benchmarking (C) as two exemplary types of data usage. B) Average activity level (firing rate) in each of the 8 neuronal populations 2/3E, ..., 6I depicted in panel A, obtained from simulations of the model on four different GPU platforms (see labels at horizontal axis in panel C). C) Real time factor (ratio between wall-clock time T_{wall} and simulated biological time $T_{\text{model}} = 10\,\text{s}$) for four different GPU computing platforms. Error bars (red) in B) and C) depict standard deviations across ten different model realizations (random-number generator [RNG] seeds) and simulation runs for each platform (error bars are partly too small to be visible).

In Fig. 5B, a performance analysis of the hydrologic model is shown. A routine exercise in hydrologic modeling is parameter calibration. This stems from the fact that hydrologic modeling follows the paradigm that model parameters govern model behavior. For example, water infiltration into the soil is controlled by a shape parameter that dictates how fast soil infiltration is decreasing when the soil is drying. This flexibility is fundamental to account for highly conductive soils (i.e., sandy soils) and low conductive soils (i.e., clay soil). Here, we are able to analyze how model performance depends on the particular choice of a parameter (Fig. 5B). This can be done in principle over all the simulations that have ever been done using the hydrologic model which is not possible without this comprehensive set of metadata. Performance is measured by comparing the model output, typically river discharge, with observations and calculate a performance measure from the simulated and observed time series. These performance measures are created in a way that the optimal value is at one and decreases to $-\infty$ for simulated and observed time series that are unrelated[47]. Hydrologic models are run over a set of model parameters and given the performance of these, a cumulative distribution function can be calculated (Fig. 5B)[18,48]. Fig. 5B shows that model configuration $\mathscr{P}_1$ is outperforming model configuration $\mathscr{P}_2$. Parameter set $\mathscr{P}_1$ leads to a larger frequency of performance around 0.5 and higher, while parameter set $\mathscr{P}_2$ has larger relative frequency for lower performance values of zero and less. It is important to highlight here that such an analysis can be done with all available parameter configurations, moreover subsets of parameters of interest can be created so that their performances are investigated. This allows users to obtain a deeper understanding of how combinations of model parameters are affecting model performance at a level of detail that is currently not possible.

A critical task in the evaluation of hydrologic models is the visual inspection of the simulated and observed time series of a variable of interest. Fig. 5C depicts two time series of river discharge. The time series are based on simulations (red line in Fig. 5C) and the black line is based on observations. The visual inspection then allows understanding the impact of the parameter calibration. More precisely, it allows understanding which part of the hydrograph is simulated well (e.g., high river discharge,

low river discharge values, or the transition phase between high and low values). Having an annotated database at hand, as shown in Fig. 5, allows freely selecting parameter configurations from already created simulations, not necessarily related to parameter optimization. Ongoing discussions on modern hydrologic sciences raise the need for reusability of hydrologic models results to better understand how process parametrizations are affecting model performance and behavior[49]. Such efforts largely benefit from enriching model results with metadata describing the simulation experiment. For example discrepancies in model performance can be found to not only be due to differences in model parameters, but also model configuration.

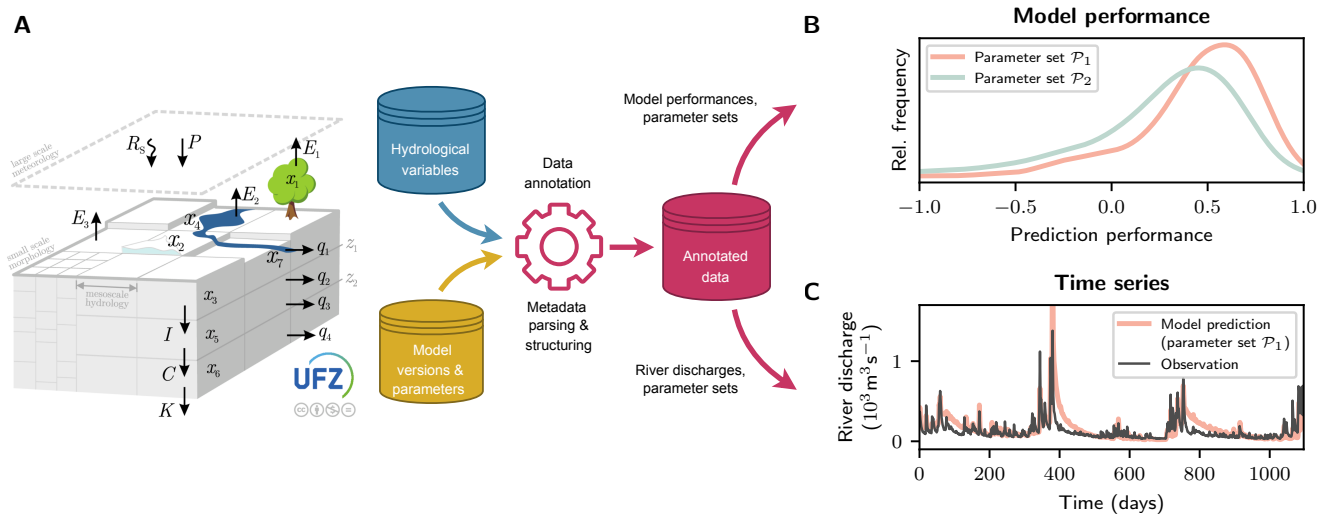

Figure 5. Hydrology use case. A) A hydrologic model, here represented by the logo of the mesoscale Hydrologic Model (mHM `https://mhm-ufz.org`, last access: 25 June 2024) creating output (i.e., hydrological variables shown by the blue cylinder) and additional metadata information (yellow cylinder). A hydrologic model simulates the water cycle at the land surface. A typical output variable is river discharge ($m^3 s^{-1}$). Hydrological data is annotated by metadata post-processed by the *Archivist*. **B)** Distribution of prediction performances of the hydrologic model across measurement stations for two parameter sets $\mathscr{P}_1$ and $\mathscr{P}_2$. Prediction performance is estimated by comparing simulated and observed river discharge for each measurement station. **C)** Time series of observed (black) and predicted river discharge (red; parameter set $\mathscr{P}_1$).

5 Discussion

Motivated by the expectation that rigorous metadata management is the key to enhance reproducibility and interpretability of scientific results, this work proposes and applies practices for handling metadata in simulation research. We derived a generic knowledge production workflow to illustrate that different types of metadata can be collected at different steps of a simulation, then be post-processed, and finally exploited alongside data To facilitate post-processing, we developed he *Archivist* framework, a Python-based tool for parsing raw metadata files and combining the extracted information into a structured file. As example implementations, we presented a conceptual example and two use cases with varying degrees of complexity. First was a hypothetical minimal example consisting of simple generic time-driven simulations where the collected metadata was used to track the performance of the simulation measured by the real time factor, and the accuracy of the model obtained by analyzing the generated results. Even if simple in nature, processing the metadata stemming from these simulations can prove challenging when dealing with larger volumes. As the goal of the experiment was to find a compromise between performance and accuracy by exploring the parameter space, a consequently large amount of data and metadata was generated. By automating metadata processing and structuring with the *Archivist*, the experiment pipeline was streamlined from data generation to data usage. Our second use case was a proof of concept of a benchmarking and verification workflow. By running several simulations of the same model[38] with NEST GPU[36], the simulation performance was compared and simulation results were statistically verified across multiple hardware platforms. Although the software setup was similar, the format of the information retrieved directly from the hardware varied between platforms. By leveraging the modularity of the *Archivist*, parsing functions specific to each platform were implemented and could be exchanged without needing to modify the experiment workflow. As the parsing output was unified through a common structure, the processed metadata was homogenized across hardware platforms and was transparently exploited. Our last use case was a routine parameter calibration procedure where multiple parameters for the mesoscale Hydrologic Model (mHM[18,45,46]) are evaluated across several simulation configurations. Due to its routine nature,

data and metadata for each procedure have accumulated over time. As information from each parameter set is collected, the collection of raw metadata is large even within a single calibration procedure. By filtering the metadata entries to specific parameter sets, the *Archivist* was used to efficiently explore the available database of results.

With the *Archivist's* modular design shown in "Metadata post-processing framework – the *Archivist*", users only need to define parsing functions to extract information from metadata entries, and use a schema to combine the results into desirable structures and formats. Both parsing functions and schemas can be shared among groups to foster reusability of implemented workflows. However, we note the need to properly document the defined functions and structures. The same information can be interpreted differently by multiple users; mistakenly one might extract information with a different interpretation than expected. A simple example would be reading a number with a large amount of decimal digits. Functions can be implemented to use the exact number with all decimals, truncate after an arbitrary significance of decimal digits, or round to the nearest integer. All of these implementations are valid interpretations and their use depends on how the information will be exploited later. Appropriate documentation solves this problem by disambiguating between implementations and increasing interpretability of the processed metadata. Much like the source code and version of used models and simulators help describing a simulation experiment, the implementation and version of parsing functions describe the post-processing operations. This is of particular importance considering that metadata processing for a given dataset is not a single final process, but can be refined for different needs. By keeping a copy and sharing the raw metadata set, data can be subsequently annotated by the same or other users even if certain metadata entries were filtered out after an initial processing and annotation step. Data is then further enriched by extending the available annotated information and can later be used for new studies. Future contributions to the *Archivist* could exploit the formalization of datasets structures provided by the data model to automatically define parsing targets without the need of user definition. Furthermore, given a detailed enough description of dataset contents, parsing function could be automatically generated too.

As mentioned in "Metadata post-processing framework – the *Archivist*", there exist alternative tools for metadata post-processing, such as the AiiDA[15] parsers or DataLad[17] metadata extractors. AiiDA is a workflow manager that can use parsing plugins to extract information from files generated during the simulation workflow. The parsed information can then be used as an output of a workflow step or attached as additional metadata for data provenance tracking. DataLad is a distributed data management system where users can define extractors to parse files present in the database and extract metadata to annotate respective datasets based on the data versioning system git-annex (`https://git-annex.branchable.com`, last access: 01 March 2024). Conceptually, both these tools perform similar operations to the *Archivist*: users define functions to extract information from files, these functions are used to parse the contents of the generated or stored data and extract metadata according to specific needs, and for further reusability, users can share these functions. The difference lies in how parsing is orchestrated and how the extracted metadata is handled. In both AiiDA and DataLad, parsing is triggered on a file-by-file basis. Although the metadata can be integrated in the knowledge base (i.e., the provenance graph for AiiDA or as annotated information for DataLad), each piece of parsed information is treated independently and is not meant to be combined into a comprehensive output. In contrast, the parsers employed by the *Archivist* operate on an arbitrary number of files automatically determined by user defined rules. Furthermore, the *Archivist* permits the use of a custom JSON Schema to structure and combine the parsed metadata, thereby fostering interoperability and reusability of the post-processing results. We note that our intention is not to create a replacement for features in AiiDA or DataLad, but to implement a standalone tool that can transparently suit specific user needs. Conversely, dedicated workflow manager tools such as Snakemake[16] or AiiDA are particularly useful for implementing a complete new workflow. In addition to increased reusability and interoperability of workflows, these tools offer software environment handling, HPC cluster job handling, and data provenance tracking among other features[50]. Data provenance in particular is useful for describing the contents of the data generated during simulations. Indeed, these solutions are not incompatible with our framework but could actually synergize by using the *Archivist* as a parsing and structuring backend. To this end, future work can be done on a combined implementation of the *Archivist* with existing tools for automated provenance tracking and enhanced storage platform. This combined implementation would feature automated experiment description, flexible data characterization, and efficient data annotation and exploration methods. Finally, further parsing and structuring methods can be added to the *Archivist* by the user community driven by their specific needs. Sharing these methods in a common repository would pave the way for interoperable metadata descriptions.

6 Conclusion

Our proposed practices for handling metadata in simulation workflows are applicable to a wide range of scientific disciplines with domain-specific workflows and metadata conventions. New workflows can benefit from these practices to ensure that metadata is handled accordingly. The practices can also be implemented in existing workflows without major restructuring of established code. In this case, our proof of concept tool, the *Archivist*, provides a flexible solution for parsing and structuring heterogeneous metadata files. Our practices and tool support sustainable numerical workflows, facilitating reproducibility and data reuse in generic simulation-based research.

7 Methods

7.1 Details on the neuroscience use case

The neuroscience use case focuses on a workflow for benchmarking and verification of a spiking neural network simulator across multiple hardware platforms. The workflow implementation can be found at https://doi.org/10.5281/zenodo.13585723 (last access: 30 August 2024).

NEST GPU The simulator in question is NEST GPU[36,37], an open library for simulation of large-scale networks of spiking neurons, written in the C++ and CUDA-C++ programming languages (source code: https://github.com/nest/nest-gpu, last access: 23 February 2024; documentation: https://nest-gpu.readthedocs.io, last access: 23 February 2024).

Hardware platforms The interest on this experiment lies on asserting whether the simulator would produce the same results across different hardware platforms. As the models simulated rely on floating point calculations, artifacts during numerical computations may arise during simulation and could potentially accumulate to observable differences in produced results. In particular, GPU architectures can vary highly between generations, which increases the chances of such divergence. For this reason we test the simulator with different GPUs models, both consumer and data center level, with varying architectures: one laptop with the consumer GPU RTX 3070 Ti Mobile with CUDA version 12.2; two compute clusters, JUSUF[51] and JURECA-DC[52], both using CUDA version 11.3 and equipped with the data center GPUs V100 and A100, respectively; and a workstation with the consumer GPU RTX 4090 with CUDA version 11.4.

Table 1. Hardware configuration of the different platforms used.. Cluster information is given on a per node basis.

System	CPU	GPU
Laptop	Intel Core i7-12800H vPro, 14 cores (6 P-cores + 8 E-cores), P-core 4.8GHz / E-core 3.7GHz	NVIDIA RTX 3070 Ti Mobile[2], 1410 MHz, 8 GB GDDR6, 5888 CUDA cores
JUSUF cluster	2× AMD EPYC 7742, 2× 64 cores, 2.25 GHz	NVIDIA V100[1], 1530 MHz, 16 GB HBM2e, 5120 CUDA cores
JURECA-DC cluster	2× AMD EPYC 7742, 2× 64 cores, 2.25 GHz	NVIDIA A100[2], 1410 MHz, 40 GB HBM2e, 6912 CUDA cores
Workstation	Intel Core i9-10940X, 14 cores, 3.30 GHz	NVIDIA RTX 4090[3], 2520 MHz, 24 GB GDDR6X, 16384 CUDA cores

[1] Volta architecture: https://developer.nvidia.com/blog/inside-volta, last access: 25 June 2024

[2] Ampere architecture: https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth, last access: 25 June 2024

[3] Ada Lovelace architecture: https://www.nvidia.com/en-us/technologies/ada-architecture/, last access: 25 June 2024

Network model The model used to evaluate the simulator is the cortical microcircuit of Potjans and Diesmann[38] which represents a 1 mm^2 patch of early sensory cortex at the biological plausible density of neurons and synapses (depicted in Fig. 4A). The model was simulated previously on different simulation platforms[36,40,53], and in a recent study dynamics and performance were evaluated using NEST GPU[37]. The availability of both dynamics and performance data make this model ideal for verification tests. Furthermore, the inherent complexity and scale of the network are good candidates to confirm whether numerical artifacts would be generated even on data center level GPUs.

Verification metrics To verify the simulated dynamics, we collect spiking activity of all neuron populations of the model, and compute their average firing rate. Simulations are performed using a time step of 0.1 ms and 500 ms of network dynamics are simulated before recording spiking activity to avoid transients. Then, we record spiking activity of the subsequent 10 s of network dynamics. As shown in previous studies[54], the average firing rate of a population is computed as the number of recorded spikes emitted by all neurons in the population, averaged by the number of neurons in the population, and normalized by the duration of the recording.

Performance metrics To measure the performance of the simulation in each hardware platform we use the *real time factor* as defined in "Minimal example". Here we use internal timers included in the model definition to compare the time needed for state propagation of model dynamics and the simulated biological time.

Data generation As a proof of concept, we devised a minimal workflow where we follow our previously defined practices on metadata collection, post-processing, and annotation to populate a database. Here, we describe the steps for software preparation, simulation execution, and metadata collection (see Fig. 1). For simplicity, we assume that all software dependencies for the simulator and for the data processing pipeline are already present in the target platform. We also assume that no job schedulers are needed for execution. The workflow, implemented using Snakemake, consists of nine rules underlying the production and storage of raw simulation data and metadata, two rules for metadata post-processing, and one rule for data exploration and usage. This implementation was designed so each rule can be dynamically configured through a dedicated file to increase flexibility and reusability for different simulation scenarios (such as different simulators and models). The workflow starts by cloning two separate repositories, one for the simulator and another for the model. Following this, the simulator is compiled using its CMake (https://cmake.org, last access: 23 February 2024) installation infrastructure. Then the model is consecutively run in a sequence of independent simulations each with different random number generation seed. The spiking activity predicted by the model is recorded and constitutes the primary data output of each simulation. Special care is also taken to monitor the simulated biological time and the wall-clock time for performance data. Information on the system environment before and after job execution is recorded through a dedicated metadata collection script. This allows us to compare the state of the system before and after setting up the simulation environment. Additional metadata produced before execution, such as configuration files, execution scripts, compilation output, shell output logs, and the internal metadata tracked by Snakemake itself, are stored as raw metadata. After each simulation run and metadata collection, a post-processing step computes the population-averaged firing rates from the recorded spike data (Fig. 4 B), as well as the real time factor (Fig. 4 C). Finally, the produced data and metadata are compressed into archives. As storage platform, we use an instance of a MongoDB. By employing its file storage system GridFS (https://www.mongodb.com/docs/manual/core/gridfs, last access 23 February 2024), we can upload the compressed archives and get a unique identifier for each.

7.2 Details on the hydroscience use case

The hydroscience use case ("Hydrology use case") presents how the metadata *Archivist* can be applied in the hydrological modeling sciences.

mHM An essential tool of this use case is the employed mesoscale hydrologic model mHM[18,45,46] (https://mhm-ufz.org, last access: 25 June 2024). mHM is a grid-based, spatially distributed hydrological model driven by daily precipitation, temperature, and potential evapotranspiration. It accounts for major hydrological processes such as snow accumulation and melt, canopy interception, soil infiltration, evapotranspiration, deep percolation, baseflow generation, and river routing. The open-source model code repository is available and is under active development and maintenance (https://git.ufz.de/mhm/mhm, last access: 12 February 2024). A general overview on the model processes and parameterization can be obtained from[45] and[46]. The model is an integral part of the German Drought Monitor (https://www.ufz.de/duerremonitor, last access: 25 June 2024). mHM was also applied and evaluated in multiple climatological regions, including Europe[55,56], West Africa[57], India[58], and the conterminous United States[48,59].

Hardware platforms mHM simulations were carried out on the High-Performance Computing (HPC) Cluster EVE, a joint effort of both the Helmholtz Centre for Environmental Research - UFZ (http://www.ufz.de/, last access: 25 June 2024) and the German Centre for Integrative Biodiversity Research (iDiv) Halle-Jena-Leipzig (https://www.idiv.de/, last access: 25 June 2024). The main compute hardware of EVE comprises a) 42 compute nodes with dual socket Intel Xeon 6348 CPUs with 512 Gigabytes of DDR4 main memory, two of these also include NVIDIA Tesla A100 GPGPUs, and b) 27 compute nodes with dual socket Intel Xeon Gold 6148 CPUs with up to 1536 Gigabytes of DDR4 main memory, two of which include NVIDIA Tesla V100 GPGPUs. The central network component of the cluster is an Intel Omni-Path 100 Series high performance interconnect, providing all compute nodes with non-blocking EDR bandwidth (100 Gigabit per second). All compute nodes share a 4.5 Petabyte IBM Spectrum Scale file system. The system performed with over 164 teraFLOPS under a High-Performance LINPACK (HPL) benchmark. For the simulations of this project, we have used CPU cores exclusively.

Model setup The mHM model is executed within the test basin provided along with the model source code, the so called "test basin". The test basin coming with the mHM source code is for the Moselle River basin upstream of Perl, a place, where the Moselle River leaves France and enters Luxembourg and Germany (Moselle Basin). The catchment area is approximately $11,500\ km^2$, altitude ranges between 150 and 1300 m. a.m.s.l. The Moselle River originates from the Vosges Mountains and is a tributary of the Rhine River. The origin of data used in the test example is provided on https://mhm-ufz.org/docs/ (last access: July 18th, 2024).

Performance metrics The hydroscience use cases makes use of performance metrics that are commonly used in hydrologic modeling. Explicitly, it is the Kling-Gupta efficiency (KGE[47]), that can be used to compare two time series. The metric combines the long-term mean, long-term variance, and Pearson correlation coefficient.

Data generation The mHM simulations have been facilitated using the ecFlow workflow manager[60] developed at the European Centre for Medium-Range Weather Forecasts (ECMWF). ecFlow is a client/server workflow package that allows users to execute any number of simulations. It is tailored to work on HPCs and allows to easily restart workflows if hardware and software failures occur. We have created a simple suite with three tasks. The first task is the compilation of mHM using CMake (`https://cmake.org`, last access: 23 February 2024). The second tasks executes the hydrologic model. The third task is a post-processing step that creates plots like the ones shown in Fig. 5B and C. We have now also added the *Archivist* to these workflows to manage the metadata.

8 Acknowledgments

The authors thank Dennis Terhorst, and Guido Trensch for constructive discussions. This project has received funding from the Initiative and Networking Fund of the Helmholtz Association in the framework of the Helmholtz Metadata Collaboration (HMC) project call (ZT-I-PF-3-026) and under project number SO-092 (Advanced Computing Architectures, ACA), the Helmholtz Joint Lab "Supercomputing and Modeling for the Human Brain", the European Union's Horizon Europe Programme under the Specific Grant Agreement No. 101147319 (EBRAINS 2.0 Project), and HiRSE_PS, the Helmholtz Platform for Research Software Engineering - Preparatory Study, an innovation pool project of the Helmholtz Association. The authors gratefully acknowledge the computing time granted by the JARA Vergabegremium and provided on the JARA Partition part of the supercomputer JURECA at Forschungszentrum Jülich (computation grant JINB33). They further acknowledge the use of Fenix Infrastructure resources, which are partially funded from the European Union's Horizon 2020 research and innovation programme through the ICEI project under the grant agreement No. 800858. The work was carried out in part within the HMC Hub Information at the Forschungszentrum Jülich.

9 Author contributions

All authors contributed to the conceptual metadata management framework, and the design of the example use cases. J.V. and M.K. implemented the metadata *Archivist*. J.V. implemented the neuroscience use case. M.K. implemented the hydrology use case. All authors wrote and reviewed the manuscript.

10 Competing interests

The authors declare no competing interests.

References

1. Pauli, R., Weidel, P., Kunkel, S. & Morrison, A. Reproducing polychronization: a guide to maximizing the reproducibility of spiking network models. *Front. Neuroinform.* **12**, 10.3389/fninf.2018.00046 (2018).
2. Leipzig, J., Nüst, D., Hoyt, C. T., Ram, K. & Greenberg, J. The role of metadata in reproducible computational research. *Patterns* **2**, 100322, 10.1016/j.patter.2021.100322 (2021).
3. Anzt, H. *et al.* Towards continuous benchmarking: An automated performance evaluation framework for high performance software. In *Proceedings of the Platform for Advanced Scientific Computing Conference*, PASC '19, 10.1145/3324989.3325719 (Association for Computing Machinery, New York, NY, USA, 2019).
4. Penders, Holbrook & de Rijcke. Rinse and repeat: Understanding the value of replication across different ways of knowing. *Publications* **7**, 52, 10.3390/publications7030052 (2019).
5. Gutzen, R. *et al.* Reproducible neural network simulations: Statistical methods for model validation on the level of network activity data. *Front. Neuroinform.* **12**, 90, 10.3389/fninf.2018.00090 (2018).
6. Glatard, T. *et al.* Reproducibility of neuroimaging analyses across operating systems. *Front. Neuroinformatics* **9**, 12, 10.3389/fninf.2015.00012 (2015).
7. Nordlie, E., Gewaltig, M.-O. & Plesser, H. E. Towards reproducible descriptions of neuronal network models. *PLOS Comput. Biol.* **5**, e1000456, 10.1371/journal.pcbi.1000456 (2009).
8. Ivie, P. & Thain, D. Reproducibility in scientific computing. *ACM Comput. Surv.* **51**, 1–36, 10.1145/3186266 (2018).
9. McDougal, R. A., Bulanova, A. S. & Lytton, W. W. Reproducibility in computational neuroscience models and simulations. *IEEE Trans. Biomed. Eng.* **63**, 2021–2035, 10.1109/TBME.2016.2539602 (2016).
10. Davison, A. Automated capture of experiment context for easier reproducibility in computational research. *Comput. Sci. & Eng.* **14**, 48–56 (2012).

11. Manninen, T., Aćimović, J., Havela, R., Teppola, H. & Linne, M.-L. Challenges in reproducibility, replicability, and comparability of computational models and tools for neuronal and glial networks, cells, and subcellular structures. *Front. Neuroinformatics* **12**, 10.3389/fninf.2018.00020 (2018).

12. Plesser, H. E. Reproducibility vs. Replicability: A Brief History of a Confused Terminology. *Front. Neuroinformatics* **11**, 76, 10.3389/fninf.2017.00076 (2018).

13. Guilyardi, E. *et al.* Documenting climate models and their simulations. *Bull. Am. Meteorol. Soc.* **94**, 623–627, 10.1175/bams-d-11-00035.1 (2013).

14. Albers, J. *et al.* A modular workflow for performance benchmarking of neuronal network simulations. *Front. Neuroinform.* **16**, 837549, 10.3389/fninf.2022.837549 (2022).

15. Huber, S. P. *et al.* Aiida 1.0, a scalable computational infrastructure for automated reproducible workflows and data provenance. *Sci. Data* **7**, 1–18, 10.1038/s41597-020-00638-4 (2020).

16. Mölder, F. *et al.* Sustainable data analysis with snakemake. *F1000Research* **10**, 33, 10.12688/f1000research.29032.1 (2021).

17. Halchenko, Y. *et al.* Datalad: distributed system for joint management of code, data, and their relationship. *J. Open Source Softw.* **6**, 3262, 10.21105/joss.03262 (2021).

18. Thober, S. *et al.* The multiscale routing model mrm v1.0: simple river routing at resolutions from 1 to 50 km. *Geosci. Model. Dev.* **12**, 2501–2521, 10.5194/gmd-12-2501-2019 (2019).

19. Alagić, S. *Data Model*, 20–63 (Springer New York, New York, NY, 1986).

20. Grewe, J., Wachtler, T. & Benda, J. A Bottom-up Approach to Data Annotation in Neurophysiology. *Front. Neuroinformatics* **5**, 16, 10.3389/fninf.2011.00016 (2011).

21. Koranne, S. *Hierarchical Data Format 5: HDF5*, 191–200 (Springer US, 2010).

22. Adrian, S., Christian, K., Jan, B., Thomas, W. & Jan, G. File format and library for neuroscience data and metadata. *Front. Neuroinformatics Conf. Abstr. Neuroinformatics 2014* 10.3389/conf.fninf.2014.18.00027 (2014).

23. Pezoa, F., Reutter, J. L., Suarez, F., Ugarte, M. & Vrgoč, D. Foundations of json schema. In *Proceedings of the 25th International Conference on World Wide Web*, WWW '16, 10.1145/2872427.2883029 (International World Wide Web Conferences Steering Committee, 2016).

24. Herculano-Houzel, S. The human brain in numbers: a linearly scaled-up primate brain. *Front. Hum. Neurosci.* **3**, 31, 10.3389/neuro.09.031.2009 (2009).

25. Kunkel, S. *et al.* Meeting the memory challenges of brain-scale simulation. *Front. Neuroinform.* **5**, 35, 10.3389/fninf.2011.00035 (2012).

26. Morrison, A., Mehring, C., Geisel, T., Aertsen, A. & Diesmann, M. Advancing the boundaries of high connectivity network simulation with distributed computing. *Neural Comput.* **17**, 1776–1801, 10.1162/0899766054026648 (2005).

27. Morrison, A. & Diesmann, M. Maintaining causality in discrete time neuronal network simulations. In Graben, P. b., Zhou, C., Thiel, M. & Kurths, J. (eds.) *Lectures in Supercomputational Neurosciences: Dynamics in Complex Brain Networks*, 267–278, 10.1007/978-3-540-73159-7_10 (Springer, Berlin, Heidelberg, 2008).

28. Kunkel, S. *et al.* Spiking network simulation code for petascale computers. *Front. Neuroinform.* **8**, 78, 10.3389/fninf.2014.00078 (2014).

29. Jordan, J. *et al.* Extremely scalable spiking neuronal network simulation code: From laptops to exascale computers. *Front. Neuroinform.* **12**, 2, 10.3389/fninf.2018.00002 (2018).

30. Pronold, J. *et al.* Routing brain traffic through the von Neumann bottleneck: Parallel sorting and refactoring. *Front. Neuroinform.* **15**, 785068, 10.3389/fninf.2021.785068 (2022).

31. Pronold, J. *et al.* Routing brain traffic through the von Neumann bottleneck: Efficient cache usage in spiking neural network simulation code on general purpose computers. *Parallel Comput.* **113**, 102952, 10.1016/j.parco.2022.102952 (2022).

32. Hines, M. & Carnevale, N. T. The NEURON simulation environment. *Neural Comput.* **9**, 1179–1209 (1997).

33. Stimberg, M., Brette, R. & Goodman, D. F. Brian 2, an intuitive and efficient neural simulator. *eLife* **8**, 10.7554/elife.47314 (2019).

34. Yavuz, E., Turner, J. & Nowotny, T. GeNN: a code generation framework for accelerated brain simulations. *Sci. Rep.* **6**, 10.1038/srep18854 (2016).

35. Saam, N. J. Validation benchmarks and related metrics. In *Simulation Foundations, Methods and Applications*, 433–461, 10.1007/978-3-319-70766-2_18 (Springer International Publishing, 2019).

36. Golosio, B. *et al.* Fast simulations of highly-connected spiking cortical models using GPUs. *Front. Comput. Neurosci.* **15**, 627620, 10.3389/fncom.2021.627620 (2021).

37. Golosio, B. *et al.* Runtime construction of large-scale spiking neuronal network models on GPU devices. *Appl. Sci.* **13**, 9598, 10.3390/app13179598 (2023).

38. Potjans, T. C. & Diesmann, M. The cell-type specific cortical microcircuit: Relating structure and activity in a full-scale spiking network model. *Cereb. Cortex* **24**, 785–806, 10.1093/cercor/bhs358 (2014).

39. Senk, J. *et al.* A collaborative simulation-analysis workflow for computational neuroscience using HPC. In Di Napoli, E., Hermanns, M.-A., Iliev, H., Lintermann, A. & Peyser, A. (eds.) *High-Performance Scientific Computing. JHPCS 2016.*, vol. 10164 of *Lecture Notes in Computer Science*, 243–256, 10.1007/978-3-319-53862-4_21 (Springer, Cham, 2017).

40. van Albada, S. J. *et al.* Performance comparison of the digital neuromorphic hardware SpiNNaker and the neural network simulation software NEST for a full-scale cortical microcircuit model. *Front. Neurosci.* **12**, 291, 10.3389/fnins.2018.00291 (2018).

41. Heittmann, A. *et al.* Simulating the cortical microcircuit significantly faster than real time on the ibm inc-3000 neural supercomputer. *Front. Neurosci.* **15**, 10.3389/fnins.2021.728460 (2022).

42. Kurth, A. C., Senk, J., Terhorst, D., Finnerty, J. & Diesmann, M. Sub-realtime simulation of a neuronal network of natural density. *Neuromorphic Comput. Eng.* **2**, 021001, 10.1088/2634-4386/ac55fc (2022).

43. Kauth, K., Stadtmann, T., Sobhani, V. & Gemmeke, T. neuroaix-framework: design of future neuroscience simulation systems exhibiting execution of the cortical microcircuit model 20× faster than biological real-time. *Front. Comput. Neurosci.* **17**, 10.3389/fncom.2023.1144143 (2023).

44. Samaniego, L. *et al.* Hydrological forecasts and projections for improved decision-making in the water sector in europe. *Bull. Am. Meteorol. Soc.* **100**, 2451–2472, 10.1175/bams-d-17-0274.1 (2019).

45. Samaniego, L., Kumar, R. & Attinger, S. Multiscale parameter regionalization of a grid-based hydrologic model at the mesoscale. *Water Resour. Res.* **46** (2010).

46. Kumar, R., Samaniego, L. & Attinger, S. Implications of distributed hydrologic model parameterization on water fluxes at multiple scales and locations. *Water Resour. Res.* **49**, 360–379 (2013).

47. Gupta H. V., Y. K. K., Kling H. & F., M. G. Decomposition of the mean squared error and nse performance criteria: Implications for improving hydrological modelling. *J. Hydrol.* **377**, 80–91, doi:10.1016/j.jhydrol.2009.08.003 (2009).

48. Rakovec, O. *et al.* Diagnostic Evaluation of Large-Domain Hydrologic Models Calibrated Across the Contiguous United States. *J. Geophys. Res. Atmospheres* **124**, 13991–14007, 10.1029/2019JD030767 (2019).

49. Clark, M. P. *et al.* A unified approach for process-based hydrologic modeling: 1. Modeling concept. *Water Resour. Res.* **51**, 2498–2514, 10.1002/2015WR017198 (2015).

50. Diercks, P. *et al.* Evaluation of tools for describing, reproducing and reusing scientific workflows. *ing.grid* **1**, 10.48694/inggrid.3726 (2023).

51. Vieth, B. V. S. JUSUF: Modular tier-2 supercomputing and cloud infrastructure at jülich supercomputing centre. *J. large-scale research facilities JLSRF* **7**, 10.17815/jlsrf-7-179 (2021).

52. Thörnig, P. JURECA: Data centric and booster modules implementing the modular supercomputing architecture at jülich supercomputing centre. *J. large-scale research facilities JLSRF* **7**, 10.17815/jlsrf-7-182 (2021).

53. Knight, J. C., Komissarov, A. & Nowotny, T. PyGeNN: A python library for GPU-enhanced neural networks. *Front. Neuroinform.* **15**, 659005, 10.3389/fninf.2021.659005 (2021).

54. Dasbach, S., Tetzlaff, T., Diesmann, M. & Senk, J. Dynamical characteristics of recurrent neuronal networks are robust against low synaptic weight resolution. *Front. Neurosci.* **15**, 757790, 10.3389/fnins.2021.757790 (2021).

55. Thober, S. *et al.* Seasonal soil moisture drought prediction over europe using the north american multi-model ensemble (nmme). *J. Hydrometeorol.* **16**, 2329 – 2344, 10.1175/JHM-D-15-0053.1 (2015).

56. Rakovec, O., Kumar, R., Attinger, S. & Samaniego, L. Improving the realism of hydrologic model functioning through multivariate parameter estimation. *Water Resour. Res.* **52**, 7779–7792, https://doi.org/10.1002/2016WR019430 (2016). https://agupubs.onlinelibrary.wiley.com/doi/pdf/10.1002/2016WR019430.

57. Dembélé, M., Hrachowitz, M., Savenije, H. H. G., Mariéthoz, G. & Schaefli, B. Improving the predictive skill of a distributed hydrological model by calibration on spatial patterns with multiple satellite data sets. *Water Resour. Res.* **56**, 10.1029/2019wr026085 (2020).

58. Saha, T. R., Shrestha, P. K., Rakovec, O., Thober, S. & Samaniego, L. A drought monitoring tool for south asia. *Environ. Res. Lett.* **16**, 054014, 10.1088/1748-9326/abf525 (2021).

59. Livneh, B. *et al.* A spatially comprehensive, hydrometeorological data set for mexico, the u.s., and southern canada 1950–2013. *Sci. Data* **2**, 10.1038/sdata.2015.42 (2015).

60. Bahra, A. Managing work flows with ecflow, 10.21957/nr843dob (2011).