Future Generation Computer Systems 175 (2026) 108043

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Improving reproducibility of interactive notebooks using application virtualization

Raza Ahmad [a,*], Naga Nithin Manne [b], Tanu Malik [a,c]

[a] School of Computing, DePaul University, 243 S Wabash Ave, 60604, Chicago, IL, USA
[b] Argonne National Laboratory, 9700 S Cass Ave, Lemont, IL, 60439, USA
[c] Department of Electrical Engineering and Computer Science, University of Missouri-Columbia, 416 S 6th St, 65201, Columbia, MO, USA

ARTICLE INFO

Keywords:
Notebook reproducibility
Provenance
Notebook containers
Application virtualization

ABSTRACT

Notebooks have become widely popular in scientific computing, serving as both web-based interactive interfaces for program workflows and lightweight containers for sharing code and its output. However, reproducing notebooks across different target environments remains challenging because they do not include the computational environment in which they were executed. As a result, while notebooks are shareable, they are often not reproducible. Application virtualization (AV) methods enable both shareability and reproducibility of applications across diverse environments. However, AV-based tools typically encapsulate non-interactive, batch applications, making them unsuitable for interactive computational environments like Jupyter notebooks.

We introduce `FLINC2`, an easy-to-use user-space tool designed to create reproducible notebook containers. `FLINC2` virtualizes the notebook process to enable interactive computation while capturing the environment and all data dependencies accessed by the notebook. Unlike current methods, `FLINC2` virtualizes the process without introducing any changes to the content of the notebook. By collecting provenance during virtualization, `FLINC2` ensures consistent notebook behavior across different environments. Additionally, it facilitates seamless export of notebook containers to non-notebook environments. We explore how `FLINC2` integrates with notebook ecosystems and conduct experiments using a dataset of notebooks from the domains of hydrology, data science, and earth science to demonstrate its application and generalizability. Our results show that `FLINC2` generates lighter weight containers compared to equivalent non-interactive batch containers without introducing any changes to notebook platforms, while maintaining the same interactive workflow of notebooks.

1. Introduction

Computational notebooks have become a widely adopted tool in scientific computing. They facilitate interactive workflow development, allowing users to receive immediate feedback on individual sections of a workflow as they are executed. This feature is particularly beneficial for testing and debugging workflows during development. Unlike traditional workflow systems which require the entire workflow to be predefined, notebooks promote a more iterative programming approach. As a result, they are extensively used for a variety of tasks, including data exploration, model execution, and result visualization. As an example, Fig. 1 shows a notebook in which data from the geoscience domain is explored, analyzed, and then given as input to a hydrology model for execution. Features like these have propelled the adoption of interactive notebooks and popular notebook platforms like Jupyter [1] and Apache Zeppelin [2] are now integral to the scientific community.

1.1. Sharing notebooks

Sharing notebook files (e.g., `.ipynb` files) enables other users to interactively repeat the workflow defined in the notebook. For example, Fig. 2 illustrates a notebook where the workflow is organized as a sequence "cells" of code. Each cell represents a step in the computational workflow. Results are generated by manually executing these cells, with outputs often embedded directly in the notebook file. When a user shares the notebook, others can rerun the cells to obtain the same or similar results, depending on the program's deterministic behavior. Users can also modify the workflow interactively, such as altering the type of plot generated, to further validate or extend the results. This

* Corresponding author.
E-mail addresses: sra0nasir@gmail.com (R. Ahmad), nithinmanne@gmail.com (N.N. Manne), tanu@missouri.edu (T. Malik).

https://doi.org/10.1016/j.future.2025.108043
Received 28 January 2025; Received in revised form 15 July 2025; Accepted 23 July 2025
Available online 5 August 2025

```
[1]: import numpy as np
     import matplotlib.pyplot as plt
     import pysumma

[2]: ybeg = int(snow.head(1)['date'].dt.year) + 1
     yend = int(snow.tail(1)['date'].dt.year) - 1
     yrs, snowmax, flowmax = [ ], [ ], [ ]
     for yr in range(ybeg,yend):
        yrs.append(yr)
        start_date = str(yr) + '-01-01'
        end_date = str(yr) + '-12-31'
        snowmax.append(snow.loc[snow['date'] >= start_date]['swe'])
        flowmax.append(flowdata.loc[flowdata['date'] >= start_date['flow'])

[3]: snowmax = np.array(snowmax)
     flowmax = np.array(flowmax)
     plt.scatter(snowmax, flowmax)
     m, b = np.polyfit(snowmax, flowmax, 1)
     plt.plot(snowmax, m*snowmax+b)
     print("Regression fit:")
     print("Max daily streamflow = "+ b + " + "+ m + "*(Max snow water
     equivalent in inches)  (cfs)")
     Regression fit:
     Max daily streamflow = -268.282 + 30.534*(Max snow water equivalent in
     inches)  (cfs)
```

Fig. 1. An illustrative notebook in which user code is divided into cells performing model execution and data exploration.

process of sharing and re-executing notebooks fosters collaborative analysis and enhances reproducibility.

Although a notebook file is inherently shareable, it does not ensure reproducibility. A notebook file is a static document that contains code and data outputs, but it lacks the dependencies required to execute the code specified in its cells. Users typically need to manually obtain explicit dependencies, such as data files referenced in the program or libraries imported via `import` statements. However, implicit dependencies—such as those required by libraries like GeoPandas and Rasterio in the notebook shown in Fig. 2—are not automatically resolved. This often leads to a "dependency hell" [3], where the dependencies used in the original host environment may be missing or incompatible in the target environment due to version mismatches or unavailability, resulting in execution conflicts.

1.2. Application virtualization

To address dependency issues, *application virtualization (AV)* is used which is a lightweight method for sharing code, data, and environment of an application. AV consists of two phases, audit and repeat. The audit phase is used to audit the execution of a running application to capture dependencies and the repeat phase repeats the execution captured in the audit phase using the same dependencies. Several recent systems [4–6] use application virtualization as a method to audit the execution of a program, and create a container-like package comprising all files (code, data, and environment) referenced by the program during its execution. This package can then be used to repeat the application in different environments. These systems enable computational reproducibility [7] of scientific models and collaborative analytics [8,9]. However, current systems assume batch workflows which are classically developed with scripts and are non-interactive. They do not support auditing notebooks which are web-based and interactive, and are thus unaware of the client–server communication protocols required to enable notebook interaction.

A straightforward way to address this challenge is by extending application virtualization to include the underlying client and server components of the notebook system. However, this approach only achieves repeatability, meaning users can rerun the audited notebook as is but cannot modify it. This limitation arises because executing a modified cell initiates a new communication between the client and server, which was not part of the original audited session. This

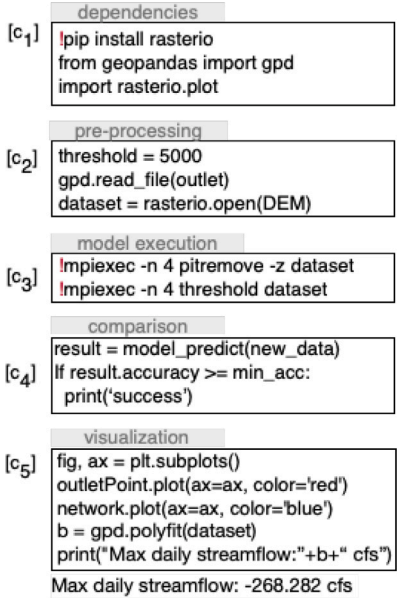


Fig. 2. An illustrative notebook N_1 of five cells of code which combines Python script and shell execution. Shell execution is done via ! commands.

constraint significantly undermines the interactive nature of notebooks, which are designed for users to add, modify, delete, and reorder cells during development. To overcome this, application virtualization must be extended to create notebook containers that can be seamlessly shared and repeated while also supporting *flexible reproducibility*. This includes the ability to modify code and data in various environments. Such containers must retain the core interactive functionality of notebooks, ensuring that users can interact with the new environment as freely and effectively as they could with the original.

1.3. Extending application virtualization

In this work, we detail a systematic approach to extend application virtualization for notebook-based platforms. We introduce `FLINC2`, a user-space tool designed to create compatible kernels for notebooks,

ensuring proper auditing in the host environment and accurate reproducibility in the target environment. During the *audit* phase, FLINC2 generates a lightweight notebook container that includes all necessary dependency files to reproduce the notebook. In the *repeat* phase, FLINC2 uses this container to seamlessly execute the corresponding notebook in the target environment. Users can share the notebook file alongside the container to guarantee reproducible execution. Before repeating the notebook execution, FLINC2 checks the provenance data collected during the audit phase to determine whether the notebook file has been modified. If no changes are detected, the container is used for re-execution; otherwise, regular execution based on interaction with the target environment resumes . Additionally, FLINC2 features an *export* mechanism, which allows the audited container's contents to be exported to other container-like virtual environments. This capability is particularly valuable for ensuring the reproducibility of notebooks on non-notebook platforms.

The design of FLINC2 is general; it only relies on two properties which are generic across all notebook platforms, namely interactive computation and a client–server architecture [10]. In addition to experiments on JupyterHub, we explore the generalization of FLINC2 on two other popular notebook platforms. In our experiments, we use FLINC2 with HydroShare [11] which is a cyber infrastructure for sharing geoscience models and data. To support notebook-based analysis, HydroShare supports multiple computational environments, such as CUAHSI JupyterHub [12] and CyberGIS-Jupyter for Water (CJW) [13]. These environments maintain a unique set of software dependencies to support the various analysis tasks of their user base.

HydroShare allows users to interactively run notebooks within each environment. However, users often develop workflows that must be executed across environments, either to couple with other workflows or simply for more cost-effective hardware or software in other cloud environments. Currently, users can create and reproduce notebooks successfully as long as they remain in the same environment. With FLINC2, users can execute notebooks in a different notebook environment or produce a documentation of needed dependencies to run in a non-notebook environment as well. FLINC2 is a user-friendly tool which is easy to deploy and can be used by domain scientists to execute their notebooks without prior technical knowledge or background.

The rest of the paper is organized as follows: Section 2 describes types of reproducibility issues that users face when sharing notebooks. In Section 3, we describe the model of notebook execution and sharing that prevails in current notebook platforms. Section 4 describes the FLINC2 system and how it extends application virtualization to notebook platforms and ensures reproducible execution. Section 5 describes flexible reproducibility in FLINC2 and the mechanism to export to non-notebook environments. In Section 6, we elaborate the implementation details of FLINC2 and its integration in the notebook environment. In Section 7, we describe our experimental setup and results. In Section 8, we mention some limitations of FLINC2, discuss its generality across different notebook environments with experimental results, and perform a cross-platform analysis across server platforms. Section 9 highlights the related work in this area. We conclude in Section 10 with an overview of the system and future directions.

2. Motivating use case

We describe the reproducibility issues that arise when sharing notebooks. Such issues come up regularly in cyberinfrastructure that enable notebook-based interactive analytics for its users.

Consider three scientists, Alice, Bob, and Charlie, who are collaboratively engaged in analyzing maximum daily stream flow as a function of maximum snow water equivalent. Alice, the primary scientist, has developed a notebook that: (i) downloads data for a region of interest, (ii) preprocesses it, (iii) uses climate data to run a simulation model; (iv) compares simulated streamflow data with the simulated snow water data, and, finally (v) plots and visualizes the fit function. Fig. 2 shows these steps divided into cells along with the code.

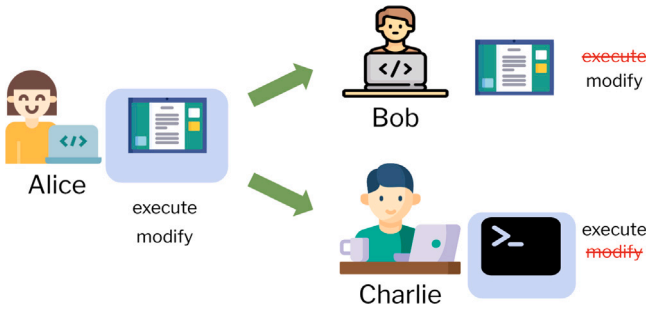


Fig. 3. Illustration of the usecase when Alice wants to share her research with collaborators Bob and Charlie. Bob is able to modify the notebook but not execute it. Charlie is able to execute her program but not modify it.

Alice develops her notebook in Python and C; it uses Python packages to perform preprocessing and visualization (steps (ii) and (v)) and C and Fortran[1] binaries to run simulation model and perform the comparison (steps (iii) and (iv)). Some of these are standard packages such as Numpy and Matplotlib, but it also includes special simulation model packages such as PySumma, ArcGIS, GeoPandas, Rasterio, and PyRhessys, which are specific to the field of hydrology. The C packages are executed by invoking the shell with a '!' from the notebook, as shown in cells 1 and 3.

To collaborate, Alice has successfully repeated her notebook execution several times to ensure it produces the same or similar results. Now, Alice wants to share her notebook with Bob and Charlie, who also aim to reproduce her workflow. The following scenarios illustrate the reproducibility-related issues that arise at their ends and are succinctly illustrated in Fig. 3:

- **Case 1: Same interface, different environments.** Alice has shared her notebook with Bob who also uses a notebook platform to develop and test workflows. Alice has also documented the dependencies for him. Although Bob can open and edit Alice's notebook, he cannot execute it. Alice's notebook uses certain Python and C dependencies that are not present in Bob's environment. While Bob is aware of the dependencies and he could install some Python dependencies via the notebook interface, he does not have sufficient privileges to upgrade his current environment with all dependencies, especially the C dependencies.
- **Case 2: Different interface, same environment.** Alice has shared her notebook with Charlie who does not use a notebook platform and prefers to work on a terminal interface. To ensure Charlie can run her code, Alice has exported her notebook to a Python file, and has also created a Docker container having all the required C and Python dependencies. Using this container and the python file, Charlie is able to successfully repeat Alice's workflow. However, he also wishes to extend her model and compare the execution with his workflow that is configured on his own computer. To do this, he must either import his own workflows and data to Alice's container or bring her code, data, and environment to the environment on his own computer.

Fig. 4 shows that the reproducibility challenge in the first case arises because the environment is not transported with the notebook file. In the second case, the environment is shared in a container, but it is not flexible to be extended to include new simulation models and data that are available outside that environment. A solution that accounts for these two use cases must also account for the more general, *'different interface, different environment'* use case.

[1] For efficiency, scientists often develop simulation models in C and Fortran.

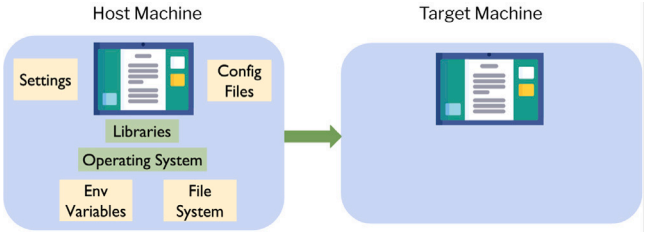


Fig. 4. When a notebook is shared with another system, its accompanying environment necessary to execute that notebook is not shared with it.

3. Notebook model of execution and sharing

We describe the current model of interactive computing within notebook platforms with Jupyter as an example. Fig. 5 illustrates this model and the client–server communication process we describe.

Notebook platforms operate in a client–server architecture. Users edit notebooks and execute in a read-eval-print-loop (REPL)-style via a web browser. In REPL-style, an interactive process reads a cell of computer code and executes it. The next read-evaluate (akin to a loop) specified in the next cell proceeds from the computational state of the previous evaluation. Notebooks assign each cell evaluation with a sequence number. They preserve this sequence which shows a safe order for cell re-execution, thus obtaining the same result at a later time.

On notebook platforms, the server maintains the *kernel* which is a programming language-specific interactive process that runs independently and maintains the state of the notebook computations as it progresses. It is aware of the computational environment and the set of dependencies needed to run those computations. The kernel receives the user code one cell at a time, executes it using its configured environment, and returns the response of the executed code in the form of text and/or visualization which is then displayed to the user through a web browser.

To execute a variety of notebooks, each kernel is typically configured to run in a container or a virtual environment (such as Conda) that is aware of the dependencies and software packages required to support the notebooks. For example, if a notebook uses the Spark library and a specific version of the Python interpreter, then the corresponding kernel would contain Apache Spark and the required version of Python in its environment to execute that notebook successfully. The same kernel cannot be used to execute all other notebooks, for instance, the notebook specified in Fig. 2 which will require a kernel having the dependencies GeoPandas and Rasterio. Multiple kernels may co-exist for different programming needs. For instance, Jupyter notebook platform supports the IPython kernel for Python programs, Xeus Cling for C++ programs [14], and Xeus-sql for SQL-based programs [15].

Since notebook files do not store their respective environments, users adopt certain practices to make notebooks repeatable across heterogeneous environments. One practice is to only use Python dependencies in a notebook and insert the code to install them in the beginning of the notebook. Users can then install those Python dependencies in the user space of their target environment. For example, in the notebook in Fig. 2, the user is allowed to install 'rasterio' in cell 1 within the user space. The kernel recognizes this instruction and will therefore successfully execute it. An alternate practice is to create a Docker container containing all the dependencies needed to run the notebook, and share the notebook file as part of the container image.

Neither of these practices generalize to all kinds of notebooks or fundamentally resolve the reproducibility issue. As the example illustrates in cell 3, scientific computing notebooks often combine Python with C, Fortran, and shell utilities to conceptualize a workflow and its steps. For non-Python code, installing dependencies in user space is often not sufficient as the install requirements vary per computing platform. Custom libraries are generally not available through standard package managers, and often do not properly follow official packaging guidelines. Further, installing all the required dependencies with their correct versions is a manual and labor-intensive task. Finally, sharing Docker images is not very practical since they are well known to over-bloat (i.e., contain more dependencies than needed) and thus often become too large in size for sharing purposes [16,17].

4. Lightweight notebook containers with FLINC2

FLINC2 creates notebook containers that are repeatable on heterogeneous target notebook platforms. In this section, we describe how FLINC2 adapts application virtualization to create and repeat notebook containers.

4.1. Creating and repeating notebook containers

FLINC2 extends application virtualization (AV) to the interactive client–server architecture of notebook platforms. The key idea in application virtualization is to use the Linux system utility *strace* to monitor the processes associated with an executing program and record its interactions with the operating system.

In the case of the notebook, there are two executing programs—the notebook client program and the server-side kernel program connected via network connections. AV can be used to audit the separate client–server programs each communicating via network connections [18,19]. However, we do not need to audit both the client and the server programs in notebook platforms since the server-side kernel performs all computations whereas the client program simply comprises the notebook file (*e.g.,* `.ipynb`), which is already available and shareable. Hence, in FLINC2, we extend AV to the server-side kernel only even though the audited program must still make successful connection with the notebook client program in a new environment.

To extend AV to the server-side notebook kernel, FLINC2 creates two additional kernels — an *audit* kernel and a *repeat* kernel. The audit kernel is for recording the notebook execution and capture its dependencies and the repeat kernel is for reusing the captured dependencies when the notebook is re-executed. Thus, if the notebook in Fig. 2 had chosen *Python-3.4* as the kernel during interactive development, then FLINC2 will create two additional kernels, *Python-3.4-audit* and *Python-3.4-repeat*.

The *audit* kernel comprises of the user kernel, such as *Python-3.4*, executing the notebook code and an additional layer which observes the execution of user kernel using *strace*. *strace* internally uses the *ptrace* system call, which intercepts any system call made to the operating system. In AV, the system call intercept is used for recording the accessed files by copying into a container-like package. Thus the audit kernel monitors the user kernel and captures all dependencies of the notebook during its execution. This auditing is different from typical AV auditing in two ways.

First, the audit kernel misses an important dependency, which is the client notebook file since the notebook file is never transferred to the user kernel—the notebook client only shares the computations to execute on the server-side. This does not pose a problem for executing a notebook since notebook files continue to be shared outside the container as per the standard practice. However, the notebook file can be modified which may violate the contents of the container. We discuss this issue in detail in Section 4.2.

Second, the audit kernel stores the connection information required for connecting the notebook with the kernel process. It contains the IP address of the kernel and the relevant ports to facilitate communication between the notebook client and the kernel.

The *repeat* kernel repeats the shared notebook and its container in a new target environment. Since the shared notebook must use the dependencies from the notebook container created during the audit phase, it first makes a connection with the notebook container. The

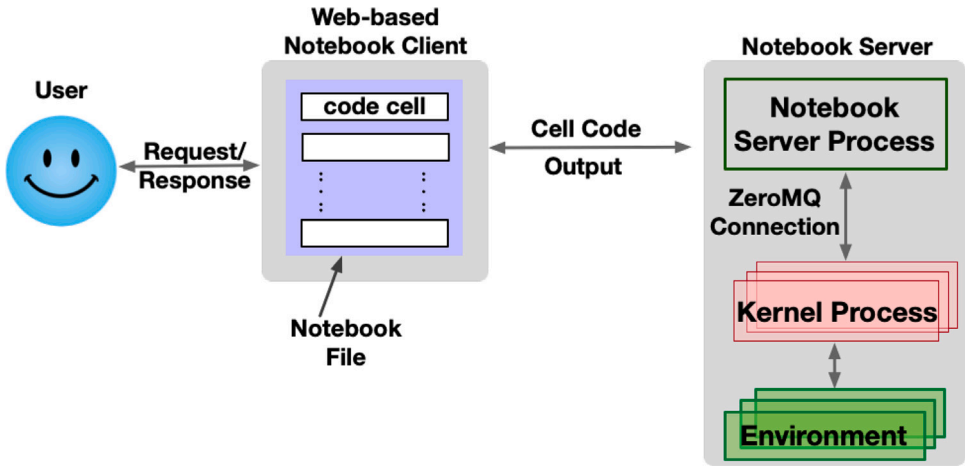


Fig. 5. A notebook platform architecture consisting of the web-based notebook client and the notebook server. A client connects to the server process, which interacts with *one* of the available kernels. The resulting kernel process is configured with an environment. It executes user code and returns the response to display in the web-based client.

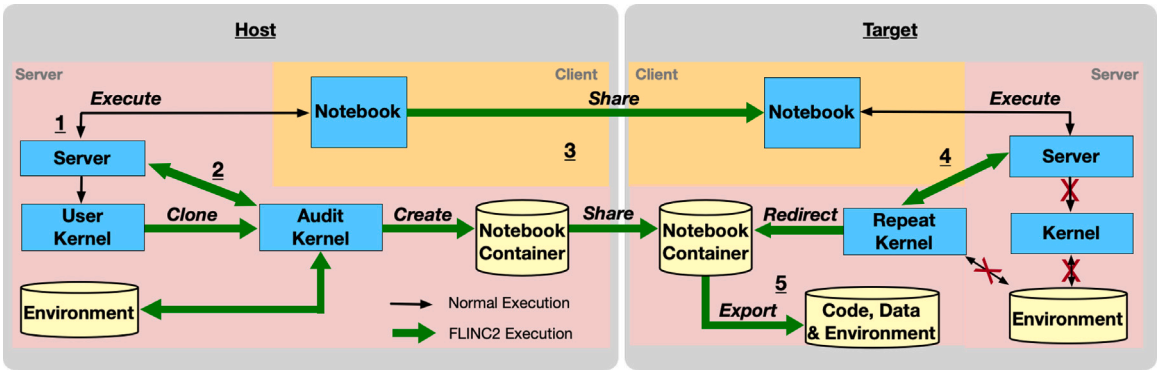


Fig. 6. The workflow of notebook reproducibility using Audit and Repeat Kernels on two different computational environments. The thin black arrows show normal program execution and solid green arrows represent FLINC2 control and data flow.

new connection is established by the repeat kernel which substitutes the old connection information in the notebook container with the new connection information generated dynamically by the notebook client.

Fig. 6 shows the various steps in the workflow of FLINC2 architecture which audits notebook in one environment and repeats it with the created notebook container in a different environment. In both environments, FLINC2 must be installed which creates the respective audit and repeat kernels. In step 1, the normal operation for the notebook client is shown to connect with the server and execute the notebook code by choosing an appropriate user kernel. In step 2, the user chooses the audit kernel to run the notebook and record its dependencies. In step 3, the user shares both the created notebook container and the notebook file with the target environment. In the target environment, in step 4, the user chooses the repeat kernel to run the notebook file. The repeat kernel connects with the container and every time the notebook execution accesses a dependency, it uses the one present in the container.

To contrast, the thin black arrows show normal program execution and solid green arrows show FLINC2 control and data flow execution. The term 'notebook container' used here is generic—it only refers to all necessary and sufficient dependencies required by the notebook. In practice, it can be an encapsulated package or a namespace isolated container as we do not distinguish between the containerization medium.

4.2. Ensuring reproducible execution via lineage tracking

FLINC2 records execution lineage during the audit phase and stores it as a log within the notebook container. This lineage tracking is primarily used to verify whether the execution during the repeat phase matches that of the audit phase—an essential check, given that the notebook file is distributed separately from the container. A further complication of sharing files outside the container is the loss of a reliable mapping between the container's internal state and the specific version of the notebook file, making reproducibility more difficult. We illustrate this issue via an example.

Consider the scenario illustrated in Fig. 2, where Alice shares both a notebook and its associated container with Bob. Upon receiving them, Bob modifies the notebook file and creates a new version—as shown in Fig. 7. If this modified notebook when reconnected to the original container, executes without errors and yields results equivalent to those originally produced by Alice, one might conclude that the notebook has reproduced correctly. However, the container does not record which version of the notebook file it was originally associated with. Ideally, the container should track and store the specific version of the notebook file linked to its internal state, ensuring a clear and verifiable mapping between file versions and container contents.

We solve this issue by auditing *per cell* provenance of the application during the creation of the container. The per cell provenance is generated by differentiating between the contents of the notebook file and its execution, and maintaining a map between the cell code and its provenance generated during execution.

Let N be a notebook consisting of cells $N = [c_1, c_2, \ldots, c_n]$, where c_i represents the ith execution of the cell. Let program state ps_i be the state of the notebook program at the beginning of each cell, as observed by the audit kernel. The program state at any point of execution consists of the values of all variables and objects used by the program at that

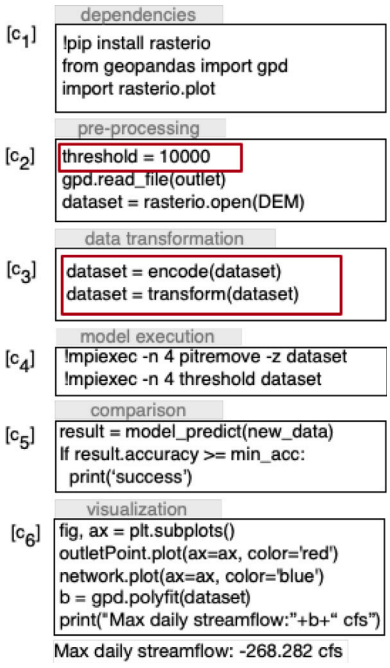


Fig. 7. The notebook N_2 is a modified version of notebook N_1 in Fig. 2. Bob changes the threshold and adds cell c3 containing data transformation.

point — intuitively, it is all the contents of the memory associated with the program. So, for the notebook N_2, the corresponding program states are $[ps_0, ps_1, \ldots, ps_5]$, in which ps_{i-1} denotes the program state just before cell c_i executed. The state ps_0 which is just before the first cell is executed, includes the value of the connection file and any initial input.

The audit kernel receives the code for cell c_i. After its execution, it maintains the following details about each program state ps_i:

- **code hash**, h_i, computed by hashing the code in cell i.
- **state lineage**, g_i, which is determined by combining three features: (i) the predecessor cell's lineage, (ii) the sequence of system events E_i that are triggered by program instructions in the cell i, which includes associated external data dependencies, and (iii) the hash h_i. Thus, $g_i = \{g_{i-1}, E_i, h_i\}$, where E_i is the ordered set of system call events in the cell and h_i is the hash of the content accessed by the event E_i. Initially, $g_0 = \{\}$.

We emphasize that the execution of the program code in cell i and the code in previous cells resulted in ps_i. Therefore, ps_i at the end of a cell's execution depends on its (i) initial environment, (ii) code that is run, and (iii) external input data. The environment is determined by the execution state at the start of the cell. Thus, (i) and (ii) are captured via g_{i-1} and E_i. Further, every external input data file is accessed via a system call event. For each such event, we record a hash of the file's content in h_i. Such hashes are typically obtained while copying content in the container.

To establish reproducible execution, FLINC2 stores the details about a notebook file in the container during audit phase. During repeat, it establishes equality between states after the execution of the cell, i.e., it determines if the cells are (i) equal with respect to their code, and (ii) have identical state lineage g_i. We state this formally as below.

Definition 1 (*State Equality*)**.** Given two program versions L_1 and L_2, state ps_i in L_1 is equal to state ps_j in L_2, denoted $ps_i = ps_j$, *if and only if* $h_i = h_j \wedge g_i = g_j$.

Program states do not remain equal when cell code is edited, which changes the hash value of that cell and any subsequent cell state. Equating state lineage depends on the system events audited during the creation of the container. Since system events are audited at the level of system calls in FLINC2, some pre-processing steps are necessary to establish equality such as accounting for partial orders and abstracting real process identifiers. We describe these issues in Section 5.2.

Suppose the notebook in Fig. 2 is modified as shown in Fig. 7, and is then re-audited using FLINC2. Since FLINC2 is not explicitly aware of user modifications, it monitors the provenance log. If it detects a change in the provenance hash, it deletes the identifier and the files associated with the previous execution and instead persists the current execution along with its associated files. This ensures that when the repeat-kernel is used, there is a single, unambiguous execution to replay. Ideally, a container should support storing multiple versions of a notebook and their corresponding files. Enabling dynamic switching of the repeat-kernel to support this is part of our planned future work. The creation of an identifier mapping to associated files is explained in more detail in [20].

5. Flexible reproducibility with FLINC2

5.1. Exporting to other environments

So far we have considered notebook containers that are created and repeated within the interactive client–server architecture of the notebook platform. In our motivating use case, such notebook containers improve sharing and reproducibility for users such as Bob who also use notebook platforms that are hosted in a different environment. We now consider how notebook containers can help users such as Charlie with whom Alice would also like to share the notebook container, but such users are neither operating on a notebook platform nor are they familiar with the REPL-style notebook programming.

FLINC2 extends application virtualization to *export* the contents of a notebook container to non-notebook isolated environments. By exporting, FLINC2 creates a new environment for the user similar to the notebook container, but into which users can add and remove or update existing files. The workflow of *export* functionality is outlined in step 5 of Fig. 6.

The export functionality is based on a log of all files collected as part of the audit phase. For correctness, FLINC2 does not copy the files in the log, but collects the provenance of all cells and uses the specification of the external input data to determine the software packages that must be installed in the new environment. Typically, each external input data file is accessed via a system call event. Since access is based on file paths, the system event also knows the path of the external input data file in the source environment. It maps each file to its software library package by using the path specification as software package files are generally only accessible from within the software package. Thus, for example, if the provenance of a cell specifies accessing the external input file `libcrypto.so.1.1`, then the software library to install is `libssl` as this software library is mentioned in the file path to `libcrypto.so.1.1`.

Once FLINC2 determines the appropriate software packages and libraries used by the notebook, it determines the version for each of them. Due to the presence of various packaging standards and the lack of strict enforcement of packaging guidelines, identifying the correct version of an installed package is quite challenging and depends on crafting programming language-specific rules. Identifying the specific version of a software package also does not guarantee that a package manager will always install the specific version of the package, which is a function of naming convention and organization adopted by the package managers. We give details of these conventions for the Python programming language as an example.

In Python, two popular packaging conventions exist: *wheels* and *eggs*, which respectively use the *dist-info*and *egg-info* formats for storing the

```
# @agent: user
# @machine: Linux 37b8d9256754 4.15.0-175-generic #184-Ubuntu SMP
Thu Mar 24 21:08:16 UTC 2022 x86_64 x86_64 x86_64 GNU/Linux
# @namespace: cde-root
1653867146 28558 EXECVE 28564 /data/bin/python /data/contents ["/data/bin/python",
"-m", "ipykernel_launcher", "-f",
"/home/.local/share/jupyter/runtime/kernel-fae76d45-037c-479a-90fc-d7090dd5d3d7.json"]
1653867146 28564 EXECVE2 -1
1653867146 28564 MEM 4198400
1653867147 28564 READ /etc/ld.so.cache
1653867147 28564 CLOSE /lib/x86_64-linux-gnu/libpthread.so.0
1653867163 28564 EXECVE 28567 /data/software/GLib/2.62.0-GCCcore-8.3.0/bin/ip
1653867163 28564 EXECVE 28567 /data/software/GDAL/3.0.2-foss-2019b/bin/ip
1653867163 28564 EXECVE 28567 /data/software/libiconv/1.16-GCCcore-8.3.0/bin/ip
1653867163 28564 EXECVE 28567 /data/software/netCDF-C++4/4.3.1-gompi-2019b/bin/ip
1653867163 28564 EXECVE 28567 /data/software/libdap/3.20.6-GCCcore-8.3.0/bin/ip
1653867163 28564 EXECVE 28567 /data/software/netCDF-Fortran/4.5.2-gompi-2019b/bin/ip
1653867163 28564 EXECVE 28567 /data/software/GRASS/7.8.3-foss-2019b/bin/ip
1653867163 28564 EXECVE 28567 /data/software/Java/11.0.2/ip /data/contents
1653867161 28564 READ /data/lib/python3.7/site-packages/matplotlib-3.3.4.dist-info
1653867161 28564 CLOSE /data/lib/python3.7/site-packages/rasterio-1.2.10.dist-info
1653867161 28564 READ /data/lib/python3.7/site-packages/xarray-0.16.2.dist-info
1653867161 28564 READ /data/lib/python3.7/site-packages/numpy-1.20.1.dist-info
1653867161 28564 READ /data/lib/python3.7/site-packages/geopandas-0.10.1.dist-info
1653867161 28564 READ /data/lib/python3.7/site-packages/pandas-1.2.2.dist-info
```

Fig. 8. A sample log of the operating system provenance collected during a notebook execution using *strace*. System calls made to the C and Python libraries can be seen.

package metadata. If either of these files/directories are present for a package, we explore the package directory to find the version of that package. The **-info* files or directories contain a `METADATA` folder which contains the version information. However, it is not required that this folder be present — in our experiments, we found several packages which did not contain it. If the folder is found missing, we then search for a file having the name similar to *version.py* in the installed package directory and scan that file to find the package version. Nevertheless, the conventions do not guarantee the correct identification of versions.

The final list of packages is curated in the form of the standard *requirements.txt* file, each line of which has the format ⟨*package-name*==*version-number*⟩. This requirements file can also be used to instantiate new computational environments since it is widely supported by several package and environment managers like *pip* and *Docker*. After determining the packages, `FLINC2` instantiates a new virtual environment. In particular, it maintains a separate directory where software packages are copied and installed. For example, in Python, this will be a *virtualenv* and in C/C++, these packages could be installed in a user-configured location such as *tmp* or *usr*.

5.2. Processing provenance logs for export and reproducible execution

Application virtualization captures the execution of a program using the *ptrace* system call which captures the details of each instruction executed by the program using operating system primitives. Fig. 8 shows the structure and a cross-section of the log file captured from a few initial cells of a notebook. This log is a representation of the lineage of the interactive kernel process which is modeled as an activity. Each file used or generated by this process is represented as an *entity*. Further, any process spawned by the interactive process is represented as an *activity* initiated by the parent process.

Each uncommented line in the log file corresponds to a system event that took place as a result of an instruction executed within that cell. It contains the instruction timestamp, process identifier, instruction type, and the operands of the instruction. This enables us to identify the executable file, its version, and any arguments passed to the program. We utilize this information to gather the list of all dependencies. The commented lines in the beginning of the log contain essential system information like user agent, namespace, machine specifications, etc.

During the export process, we process the inferred provenance to determine (i) files that are inputs to activities, (ii) files that are read by activities, and (iii) files that are outputs. Within this classification, we determine files that are in user directories versus files that are in system directories. This is necessary since system files that are read by activities determine which packages to install. We note that the log contains noise in the form of information about temporary files, outputs, and process memory execution. We filter all such files which contain execution-specific information and are not relevant for determining which packages to install.

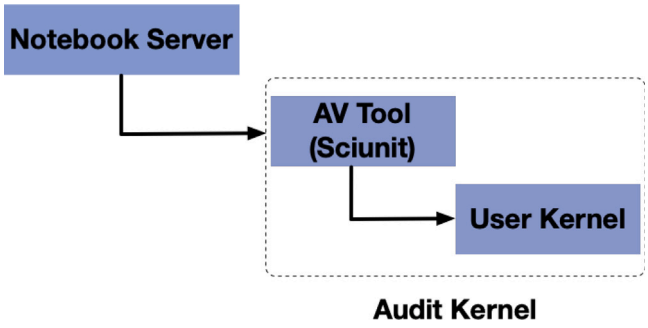


Fig. 9. The notebook server starts the audit kernel which comprises the AV process (Sciunit) and the user kernel process, initiated in this order.

For reproducible execution, we must establish state equality as per Definition 1. Lineage equality implies that at the end of cell i of version N_1, g_i is the same as that at the end of cell i of version N_2. This is true if and only if the *sequence* of system call events and their parameters till i in N_1 and i in N_2 exactly match. But, for example, if a cell forks a child process which itself issues system calls, then each version's sequence will contain the parent calls and the child process calls interleaved in possibly different orders. To address this problem, we initially separate the events of all cells into PID-specific sequences and then compare corresponding sequences to generate the provenance. Note that all time and process specific information is abstracted. Finally, memory accesses cannot be abstracted and we just count the number of accesses in a cell.

6. Implementing FLINC in notebook environments

To implement `FLINC2`, the audit and repeat kernels utilize application virtualization functionalities available from tools such as Sciunit [5] to manage notebook execution and ensure reproducibility. These kernels are installed using a specification file that defines the environment, variables, and arguments for the kernel process. To start the audit process, the user selects the audit kernel from the list of available kernels, as shown in Fig. 11. The audit kernel (started by the server) is designed to launch the user kernel needed to run the notebook code, which enables its execution to be monitored and audited by the AV tool. To start the repeat process, the repeat kernel is launched by the notebook server which uses the audited provenance data in the form of notebook container to repeat the notebook execution. The order of execution of processes for both audit and repeat processes is shown in Figs. 9 and 10, respectively.

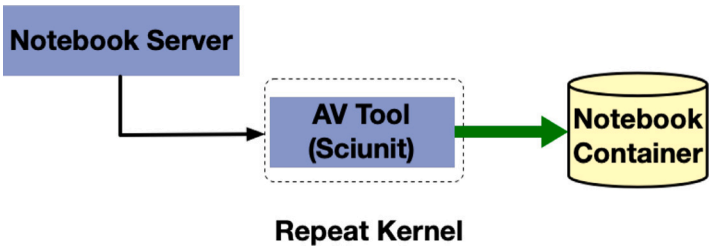


Fig. 10. The repeat kernel is launched by the notebook server process and uses the notebook container created during audit phase to satisfy all dependency requirements.

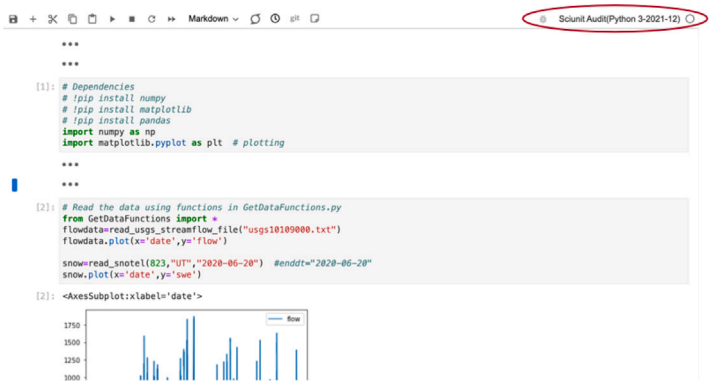


Fig. 11. To start notebook execution using FLINC2, the audit kernel must be selected from the list of available kernels.

Unlike traditional application virtualization methods that require the conversion of a notebook to a batch script for auditing, FLINC2 allows users to execute and audit the code executing within the notebook directly. Since notebooks are long-running interactive processes, they do not have a fixed point of termination as in the batch scripts. Hence, the audit kernel requires an explicit user signal such as an `exit()` function call to terminate the auditing process. However, shutting down the kernel process prematurely can terminate the entire process tree before the audit data is saved, leading to issues such as incomplete provenance storage. Adding a termination signal to the end of the notebook, as shown in the last cell of Fig. 12, could address this, but would require the introduction of code not related to the user within the notebook. This code addition compromises the reproducibility guarantees and may potentially cause system behavior issues such as kernel restarts, data duplication, and computational overhead.

To resolve this, it is essential to systematically manage the initialization and termination of kernel processes. Notebook platforms operate within a client–server architecture, where the client side (web browser) communicates with the server to execute code in a REPL-style interface. The client and the server interact with each other using the HTTP standard and web sockets on both ends. The contents of each cell are communicated to the server that executes the code, computes the results, and returns the result. The server handles computational environments using language-specific kernels, exchanging information through a set of message protocols over system sockets and ports. During execution, the server sends shutdown signals to terminate kernel processes when requested by the user.

FLINC2 ensures proper process management by intercepting these shutdown signals and managing the termination order of processes. It employs a signal handler process that launches the virtualization process and the user kernel in this order. When the server sends termination signals, the signal handler captures them and gracefully terminates all processes in the correct sequence, as shown in Fig. 13. This approach prevents system behavior issues and ensures successful completion of the audit process. Fig. 14 provides a snapshot of the process tree generated by FLINC2 during the notebook auditing which shows that the signal handler launches the AV tool which in turn launches the kernel process.

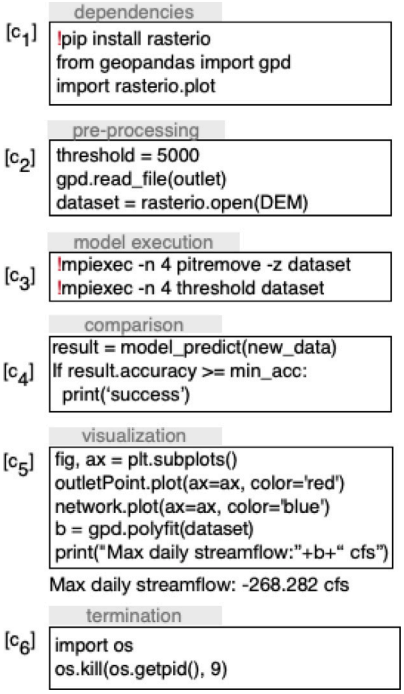


Fig. 12. An illustrative notebook showing the termination of audit process explicitly using the kill signal.

7. Experiments

FLINC2 relies on application virtualization to create notebook containers. We performed initial experiments with two popular AV tools for batch programs, Sciunit [5,21] and Reprozip [6]. Table 2 presents and compares their audit runtimes on a smaller dataset of four notebooks. We chose Sciunit over Reprozip for further experimentation

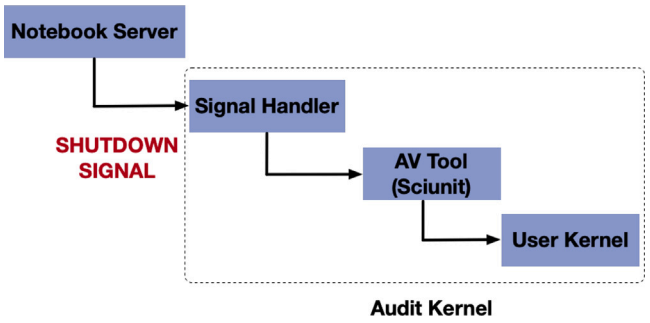


Fig. 13. The final order of processes launched by the notebook server. The signal to terminate is sent by the server to the signal handler which manages shutting down the process tree in the reverse order of process creation.

```
$ ps -ef --forest

PID  PPID   CMD
7       1   \_ python handler.py sciunit exec python -m ipykernel_launcher -f kernel-xyz.json
3169    7     \_ python sciunit exec python -m ipykernel_launcher -f kernel-xyz.json
3179    3169    \_ python sciunit exec python -m ipykernel_launcher -f kernel-xyz.json
3181    3179      \_ sciunit2/libexec/ptu python -m ipykernel_launcher -f kernel-xyz.json
3187    3181        \_ python -m ipykernel_launcher -f kernel-xyz.json
```

Fig. 14. A simplified illustration of the process tree generated for the audit kernel rooted at the notebook server process with PID 7. PTU stands for provenance-to-use which is a tool employed by Sciunit to audit the execution using *strace*.

and extension to interactive AV based on its faster audit time of 63% on average as well as higher fidelity of provenance collected during program execution. For FLINC2, we reused application virtualization functionality in Sciunit and modified it to support transparent connection of repeat kernel with the container. The *audit* and *repeat* kernels are essentially wrappers over Sciunit, and FLINC2 adds provenance information for each cell execution to the resulting logs.

7.1. Experimental setup

To conduct our experiments, we used HydroShare [11] which is an open source platform used by geoscientists for collaborative and reproducible research. HydroShare provides tools for managing and publishing models and data by leveraging interactive development and access to cloud services. It supports multiple computational platforms where users can develop and share their scientific models. One of these platforms is CyberGIS Jupyter for Water (CJW) [13]. It provides users with basic hardware and software setup to support dependency management and isolation through the use of one or more kernels. Users can use the pre-existing kernels or create their own kernel in their user space. We run our experiments on CJW using notebooks which use Python, C, and Fortran dependencies. The experiments were run on a 64 bit machine running Ubuntu 20.04.

Our dataset of experiments consists of ten notebooks. They are written in Python and C language and also invoke utilities via shell commands. Each notebook corresponds to a different use case; four from the field of hydrology, three from data science, and three from Earth science. The hydrology notebooks perform water flow analysis using data from different sources with the help of standard hydrological models. In the data science notebooks, two notebooks use machine learning algorithms to analyze and predict market behavior and the third notebook performs image classification using neural networks. The earth sciences notebooks read data from satellites and geographical data, generates labels, performs calculations, and comparison with the simulated data. We describe the important characteristics of our notebooks in Table 1.

7.2. AV experiments

We perform two different experiments with FLINC2 using our dataset. In the first experiment, we study the overhead incurred during notebook execution due to application virtualization. We analyze the execution time and storage size in both the audit and repeat modes of notebook execution using the *audit* and *repeat* kernel respectively. We term this the **Interactive-AV** method. We compare its performance against three other methods. The first method is executing a Python file that corresponds to the notebook, but one that uses application virtualization in batch or classic, non-interactive mode. We term this method **Batch-AV**. The comparison with this method will help us to estimate any overheads incurred due to the addition of the notebook architecture. The second method is the base performance of the notebook using the original kernels (i.e, without application virtualization). We term this method **No-AV notebook**. Comparison with this method will help us estimate the overheads, if any, due to interactive computation. The third method is the baseline for this experiment, which is executing the source code of the notebook in a Python file. We term this method **No-AV python**. We use nine out of the ten notebooks in our dataset for the first experiment.

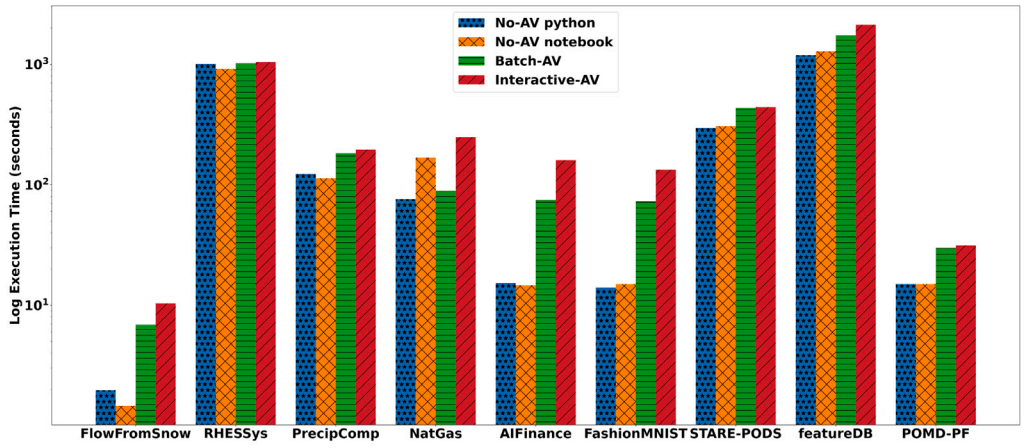
Fig. 15 shows the execution time and the corresponding storage footprint generated by the four methods for these nine notebooks. We also compute the average time it takes across all notebooks and across three runs of each notebook to execute and repeat using the four approaches, as shown in Fig. 16. Fig. 16(a) shows that the overhead in execution time using **Interactive-AV** as compared to **batch-AV** is about 16 percent. This increase is attributed to the extra time spent auditing the entire notebook execution machinery in addition to the notebook itself, as described in Section 6. For repeating the same notebook, the **Interactive-AV** method adds an overhead of only around 3.5% compared to the actual notebook execution, as shown in Fig. 16(b). This is almost 11% less than the time it takes to repeat the same notebook using **batch-AV** method. Comparing with itself, **Interactive-AV** spends about 33% less time to repeat the notebook than to execute the same notebook during audit mode, as shown by Fig. 17. This shows that once a set of experiments have been executed, it is relatively inexpensive to reproduce them multiple times. For **batch-AV** method, the percentage gain for repeating the same notebook is on average less than 13%. Thus, reproducing the notebook code using FLINC2 is interactive and convenient as well as computationally efficient.

Fig. 15(b) shows the storage footprint for each notebook when using **Batch-AV** and **Interactive-AV** methods. The baseline here is the **No-AV** which refers to both **No-AV python** and **No-AV notebook**. This is the size of the notebook file which is roughly the same as its Python code file. We see that the containers created using **Interactive-AV** are about 34% larger in size as compared to the containers created using **Batch-AV**. This increase in size is attributed to the additional provenance metadata and the files and directories gathered and stored as the audit kernel audited the entire notebook execution machinery.

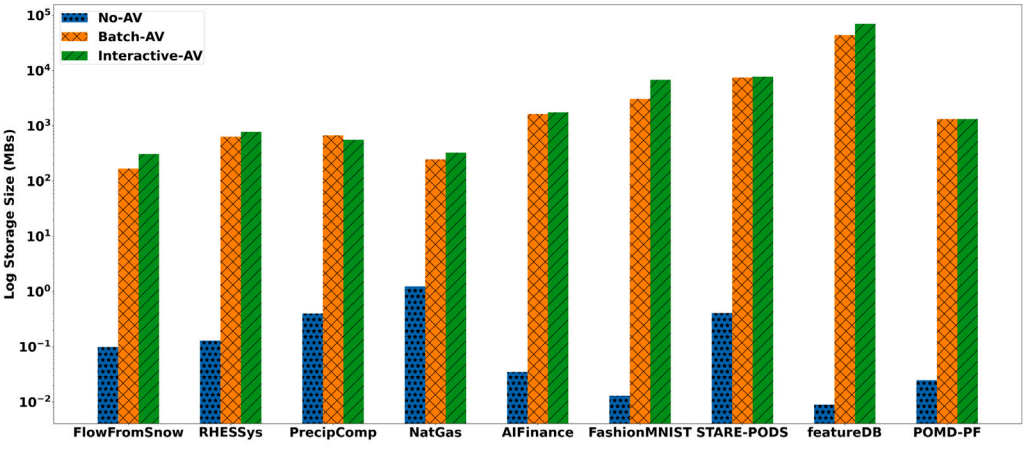
7.3. Export experiments

In our second experiment, we compute the time taken to create the new virtual environment using the *export* functionality in FLINC2. We term this method **FLINC2 Export** and compare it with the time required to create a corresponding Docker container using the same set of dependencies. We term that **Docker Container**. The baseline for this experiment is the time taken to create Docker base image from Ubuntu 20.04, termed **Docker Base**. We use five out of the ten notebooks in our dataset for the second experiment.

Figs. 18 and 19 show the comparison of computational time and space between FLINC2 Export and the two Docker containers. They demonstrate that FLINC2 requires around 46% less time than Docker to create a new virtual environment on the target system and install the required dependencies. The virtual environment created by FLINC2 is

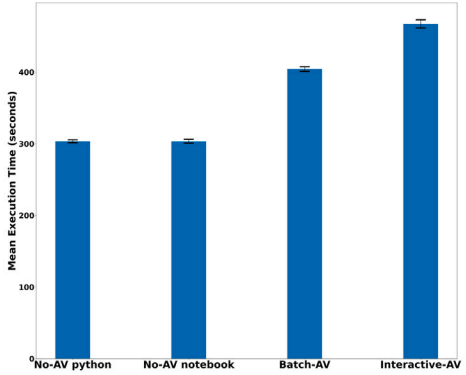


(a) Notebook Execution Time

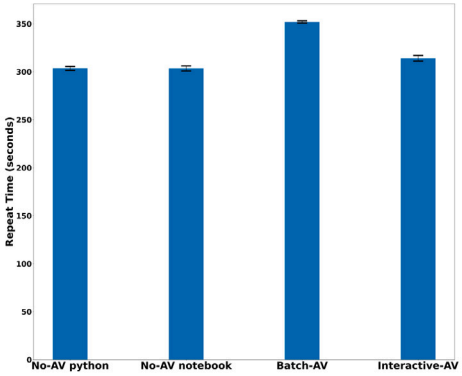


(b) Notebook Storage Size

Fig. 15. Comparison of execution times and storage size for the nine notebooks using four methods: No-AV python: running the Python file, No-AV notebook: running the notebook with its own kernel, Batch-AV: running the Python file with Sciunit, and Interactive-AV: running the notebook with Sciunit kernel. No-AV refers to both No-AV python and No-AV notebook.



(a) Notebook Execution Time



(b) Notebook Repeat Time

Fig. 16. Comparison of average execution and repeat times using each of the four methods: (i)No-AV python: running the Python file of the code, (ii)No-AV notebook: running the notebook with its own kernel, (iii)Batch-AV: auditing the Python file with Sciunit, and (iv)Interactive-AV: auditing the notebook with the Sciunit kernel.

Table 1
Characteristics of the ten notebooks in the dataset used in our experiments.

Notebook	Discipline	Size w/ Output	Size w/o Output	Primary Dependencies	Description
FlowFromSnow	Hydrology	105K	6.5K	NumPy, Matplotlib	Simple linear regression analysis on USGS streamflow data and climate data.
HAND	Hydrology	515K	6.9K	NumPy, Matplotlib, TauDEM, RasterIO, GeoPandas	Calculates height above the nearest drainage (HAND) using digital elevation model. Needs external dataset.
RHESSys	Hydrology	133.1K	15.4K	RHESSys, NetCDF, Pandas	Executes the RHESSys model to run a simulation of the Sagehen creek watershed and conduct sensitivity analysis. Needs external dataset.
PrecipComp	Hydrology	408.6K	26.6K	Cartopy, GeoPandas, Rioxarray, Shapely	Compares precipitation estimates for the Logan River watershed using two datasets AORC v1.1 and PRISM. Needs external dataset.
NatGas	Data Science	1263.6K	39.9K	NumPy, Matplotlib, openpyxl	Market analysis of natural gas by running market simulation of 35 years based on economic indicators.
AIFinance	Data Science	35.8K	9.2K	NumPy, Matplotlib, Scikit-learn, Keras, TensorFlow	Deep neural network to predict stock prices in the short term using financial news data.
FashionMNIST	Data Science	13.3K	7.2K	NumPy, Matplotlib, Torch Vision	Image classification using convolutional neural networks on the Fashion-MNIST dataset.
STARE-PODS	Earth Science	419.8K	30.7K	PySARE, STAREPandas, PySTAREPlotlib, GeoPandas, Cartopy	Reads US states geographical data and converts them into STARE trixels, chooses one event, and performs various calculations. Needs external dataset.
featureDB	Earth Science	9.2K	6.1K	CC3D, NETCDF4, NumPy, Pandas	Loads precipitation data from IMERG satellite and runs a connected components algorithm to find and label connected areas. Needs external dataset.
POMD-PF	Earth Science	25.6K	25.6K	NumPy, Pandas, PyHDF, SciPy, Cartopy	Compares data from IMERG satellite with simulated GEOS5 data by subsetting most signnificant rain values. Needs external dataset.

Table 2
Comparison of execution time in seconds of four notebooks for batch-AV method executed using Sciunit and Reprozip.

Notebook	Reprozip	Sciunit	Decrease
FlowFromSnow	39	7	82%
AIFinance	260	75	71%
NatGas	131	89	32%
FashionMNIST	538	73	86%
Average	**310**	**79**	**63%**

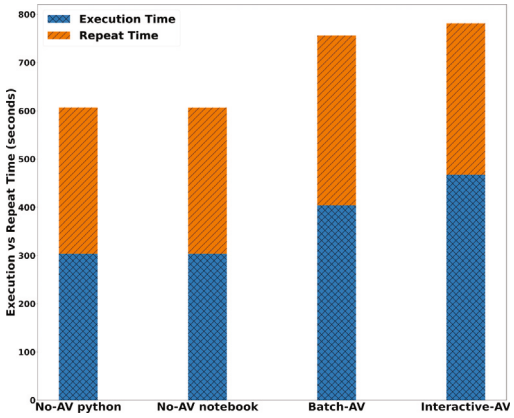


Fig. 17. The average Notebook Execution vs Repeat Time.

also lean compared to the size of containers created by Docker. Docker containers are, on average, 91% larger in size for all our notebooks compared to the containerized environment created by FLINC2.

In summary, executing a notebook using FLINC2 adds little overhead in execution time compared to running the notebook using traditional **Batch-AV** method. Repeating the notebook using FLINC2 is faster than the **Batch-AV** method and has very small overhead over the actual notebook computation. Compared to Docker containers, FLINC2 takes almost half the time to create a new virtual environment and about half the storage footprint to store the virtual environment.

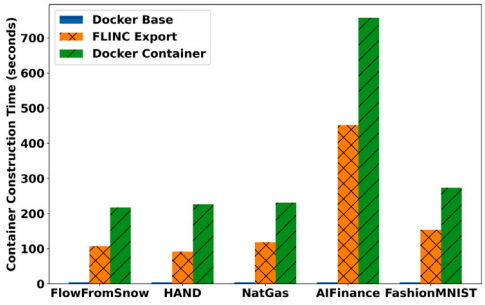
8. Discussion

The source code for FLINC2 and its documentation are readily available via GitHub [22]. Researchers can download and install FLINC2 as a user-space tool on Linux through a shell script in a few steps by following simple instructions given in the documentation wiki. The audit and repeat kernels are installed as part of its installation. The audit kernel can be used right away to start auditing a notebook to create its notebook container, which could then be shared externally and reproduced using the repeat kernel.

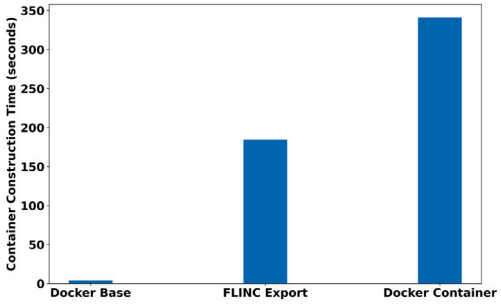
We now discuss some limitations of FLINC2 with respect to operating systems and programming languages. We then elaborate on how FLINC2 can be generalized to other notebook environments and server platforms and share some experiments for comparison.

8.1. Limitations

The objective in FLINC2 is to *automatically* determine the dependencies and provide a sandbox instead of manually creating a package

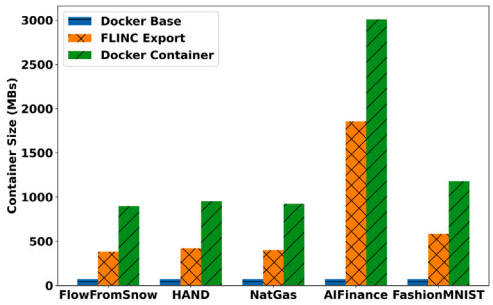


(a) Container Export Time

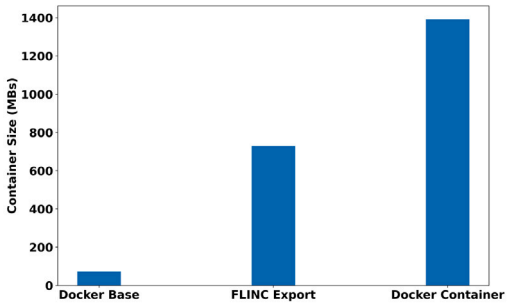


(b) Average Export Time

Fig. 18. Comparison of average export times using each of the three methods: Docker Base: creating Docker base image, Docker Container: creating the Docker container with all the required dependencies, and FLINC2 Export: creating virtual environment using FLINC2 with all the required dependencies.



(a) Container Export Size



(b) Average Export Size

Fig. 19. Comparison of average sizes using each of the three methods: Docker Base: creating Docker base image, Docker Container: creating the Docker container with all the required dependencies, and FLINC2 Export: creating virtual environment using FLINC2 with all the required dependencies.

and then sandboxing it. Currently, we do not focus on the medium of containerization or the format of the resulting container. If the container is a package, then one of the emerging formats such as Flatpak [23] or NEXTSTEP/MacOS bundles [24] can be used. Organization of container contents into meaningful bundles is further explored here [25].

FLINC2 currently depends on *ptrace* — a process tracing utility that is only available on Linux platforms. While this restricts FLINC2 to Linux, the fundamental concepts in FLINC2 can also be extended to other operating systems through the use of event tracing systems specific to them. For example, as shown previously, the Process Monitor (procmon) application on Microsoft Windows and Mac OS X kernel's auditing of system calls with OpenBSM reporter collects the same fidelity of provenance and content details as *ptrace* on Linux [26]. Thus the fundamental concepts of FLINC2 can be applied to other operating systems as well.

The mechanism and process for the *export* functionality in FLINC2 is general and could be applied to several programming languages. However, its implementation is currently limited to Python programs. This is due to the lack of standardization of packaging conventions and package management in other languages including C, C++, and FORTRAN [27]. This precludes FLINC2 from finding the entire set of libraries and packages efficiently and automating their installation in the target environment.

FLINC2 provides user-space sandboxing, ensuring that all code is re-executed with restricted capabilities under a non-root user context. This design mitigates the risk of privilege escalation from malicious code. Currently, FLINC2 enforces isolation only at the file-system level and does not extend to process or network isolation. If process or network isolation is desirable, then Docker [28] or Singularity [29] can be used in addition to FLINC2. One of the primary security risks is the potential inclusion of sensitive credentials—such as API keys, passwords, or SSH keys within FLINC2 packages. Often the use of these credentials is for accessing computing infrastructure. While machine learning approached can be used for intelligent secret scanning [30,31] across packages, it is generally a good practice to access computing infrastructure via secure tokens [32] and use of interoperable security credentials [33]. Finally, there is the possibility of unauthorized modifications to the notebook itself. To guard against this, FLINC2 employs provenance tracking to ensure the integrity and authenticity of the notebook content. Apart from the above limitations, we are currently not aware of any compatibility issues with notebook configurations and dependencies.

8.2. Other notebook environments

We explore the generalization of FLINC2 to two popular notebook platforms: Google CoLab [34] and Apache Zeppelin [2].

Apache Zeppelin: Apache Zeppelin is quite similar to Jupyter notebooks. It is capable of handling very large amounts of data which do not fit inside the main memory. Zeppelin uses interpreters and interpreter groups as the backend for processing different languages and environments which can be installed from a list of supported interpreters. Apache Zeppelin allows executing any Jupyter kernel if it contains the necessary dependencies to run the code. It has a Jupyter interpreter which acts as a bridge between the Zeppelin interpreter and the Jupyter kernel. After installing the ipykernel, we can run python code in the Jupyter interpreter by declaring '%jupyter(kernel=python)' in the beginning, followed by Python code.

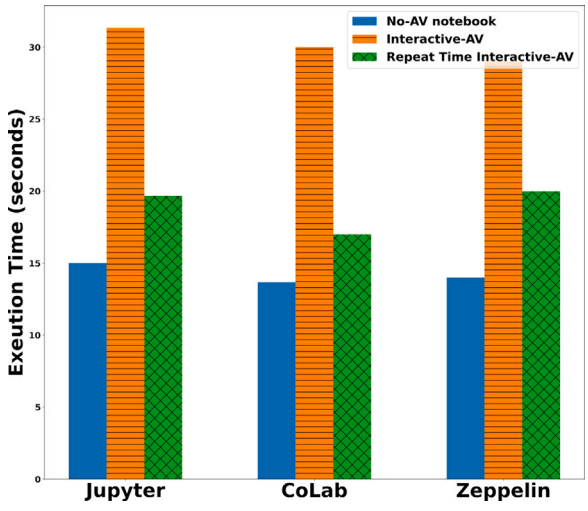


Fig. 20. Comparison of execution time of POMD-PF notebook on Jupyter, Google CoLab, and Apache Zeppelin environments against the three execution methods.

Google CoLab: Google CoLab has recently emerged as a popular notebook platform, especially for data science experiments. CoLab notebooks work with the entire data in the cloud and only support Python notebooks at this time. CoLab typically performs the entire processing on a remote machine, and allows for notebook-level versioning and (a)synchronous collaboration. The notebook code is executed in a virtual machine private to each account. CoLab also allows users to connect to runtimes hosted locally. The simplest way to do that is to instantiate the Colab Docker image and connect it to a local Jupyter runtime using web sockets from the Colab front end.

We perform experiments on both these platforms and execute one notebook from our dataset to perform comparison of execution time with the Jupyter environment. Fig. 20 shows the time it takes to execute the notebook on all three notebook environments Jupyter, CoLab, and Zeppelin, with and without FLINC2. It is clear that FLINC2 generalizes well to other notebook platforms and gives similar performance.

8.3. Cross-platform analysis

The above experiments were conducted such that audit and repeat both occurred on the same environment. In general, this is not true. To perform cross-platform analysis, we audited notebooks using the audit kernel on CyberGIS-Jupyter for Water (CJW) [13] and created notebook containers. We then shared those notebook containers and their corresponding notebooks with two different cyberinfrastructures: CUAHSI [12] and Chameleon [35], which do not have the environment and dependencies to run those notebooks. We performed this exercise on three notebooks from our notebook dataset (FlowFromSnow, AI-Finance, FashionMNIST) and determined that each of them repeated successfully without requiring explicit porting of dependencies. Our exercise was merely a test to determine successful execution therefor we chose three notebooks from our dataset that are from different domains and utilize different set of dependencies.

BinderHub [36] is another popular cloud-based service that allows users to create and share reproducible notebook environments from code repositories. It uses a JupyterHub running on Kubernetes for most of its functionality. Users are able to create self-contained environments using Docker images which can be launched and shared on a public platform like `mybinder.org`. This process has to be done manually by the user and results in the creation of Docker containers configured with the user dependencies. We give detailed comparison of the performance of FLINC2 and Docker containers in Section 7.3.

9. Related work

Lau et al. contains a detailed survey of 60 different notebook platforms [10]. REPL-style interactive computation and client–server architecture are stated as two distinctive properties of notebook platforms. We have developed FLINC2 keeping these two properties as invariant.

Notebook Containerization. The issue of reproducing notebooks in different environments was already recognized by [10]. Cunha et al. address this problem by context-aware migration of a notebook cell for execution on another platform [37]. This is achieved through a JupyterLab extension which analyzes the execution of each cell and uses a hand-crafted knowledge base to decide the environment in which to perform computation. FLINC2 focuses on the reproducibility of the entire notebook at most changing the kernel specification.

Application virtualization (AV) is a generic approach to build container-like packages without modifying applications, and predominantly uses the *ptrace* system call to automatically create a container-like package [4,38]. Some prominent tools that use AV are Sciunit [5, 39], Reprozip [40], Care [41], PRUNE [42], Nutanix AHV [43], and Parallels RAS [44]. As demonstrated by our previous work [45], AV must be extended to apply to notebook platforms. In this paper, we have shown that the extension as previously proposed still requires notebook code to be modified to terminate recording, and a signal redirection model must be adopted to avoid any modifications to notebook code.

Container specifications such as TOSCA [46] explicitly define all the code, data, and dependencies. Specifications are useful but we further show how the dependency specification can be exported and copied into the native environment and make it more flexible by bypassing calls to files that are not within the container.

Provenance Tracking in Notebooks. There are several provenance-based notebook extensions that improve reproducibility of notebooks during interactive development. Nodebook [47] is a plugin for Jupyter that checkpoints notebook state in between cells to force in-order cell evaluation. It imposes a strict ordering in notebook execution to capture its provenance. Similarly, Dataflow notebooks [48] extend Jupyter with immutable identifiers for cells and the capability to reference the results of a cell by its identifier. This allows the user to dynamically execute a cell such that all cells with related data dependencies are also executed, and all cells with downstream data dependencies can be updated as well. However, users may have to rewrite the code to utilize cell identifiers in computations. FLINC2 is used primarily for sharing notebooks in the post-interactive development phase of collaborative analytics. Apart from installing FLINC2 currently no modification to the user code is needed for sharing. The audited provenance is used to verify that shared notebooks are not inadvertently modified and guarantee repeatability.

Cyberinfrastructure for Notebooks. Notebooks are increasingly becoming part of cyberinfrastructure for scientific computing [11,49] and data analysis [50]. Policies and standards with respect to reproducible notebooks [51] are also emerging. We have shown how FLINC2 can be used in a variety of notebook platforms and scientific cyberinfrastructure.

9.1. Extension on prior work

A preliminary version of this research appeared in eScience '22 [45]. This paper extends the prior work with (i) experiments with three times larger dataset of notebooks, their analysis, and the corresponding visualizations in Section 7, (ii) Section 6 which has details of integration of FLINC2 with notebook environments and the internal mechanism of notebook execution in client–server architectures, (iv) updated Section 4 with expanded comparison and examples, (v) Section 8 which

has discussion on the limitations and generality of FLINC2 with empirical cross-platform analysis, (vi) revised and updated related work in Section 9, (vii) Table 2 for additional comparison with discussion, and (viii) new figures 1, 3, 4, 9, 11, 13, 14, and 12.

10. Conclusion

Notebooks are increasingly becoming popular in scientific computing and are being adopted for various analysis, modeling, and visualization tasks. Reproducing notebooks is critical as the target environment continues to evolve, especially in collaborative analytics. We have highlighted that notebooks do not transport their underlying environments despite being easy to share. To address this limitation, we have presented FLINC2 which a system that supports interactive and flexible reproducibility of notebooks. FLINC2 adapts and extends application virtualization (AV) to notebooks which is a popular method for addressing computational reproducibility. FLINC2 monitors notebook execution and captures it to create isolated notebook containers.

The notebook containers in FLINC2 enable reproducibility in target notebook environments while preserving their interactive programming paradigm. They are also used to export notebook containers to non-notebook environments for flexible reproducibility. FLINC2 is user-friendly and its usage requires minimal user intervention and background technical knowledge. Experiments show that FLINC2 provides efficient reproducibility of notebooks and takes significantly less time and space to execute and repeat notebook as compared to Docker containers for the same notebooks.

In the future, we plan to generalize FLINC2 to multiple operating systems and make it available for high-performance and distributed computing workflows. FLINC2's auditing and replay technology is language independent; however, we currently focus on Python notebooks as most notebooks are developed in Python. In the future, we plan to extend FLINC2 to support auditing notebooks written in other programming languages as well. We will perform experimentation and comparison with various containerization mediums like Flatpak and Singularity to provide a more holistic evaluation. We also plan to enable dynamic switching of the repeat kernel to support storing multiple versions of a notebook and their corresponding files.

CRediT authorship contribution statement

Raza Ahmad: Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Resources, Project administration, Methodology, Investigation, Data curation, Conceptualization. **Naga Nithin Manne:** Validation, Software, Methodology. **Tanu Malik:** Writing – review & editing, Validation, Supervision, Funding acquisition.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Tanu Malik reports financial support was provided by National Science Foundation. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work is supported by National Science Foundation, United States under grants CNS-1846418, NSF ICER-1639759, ICER-1661918. We acknowledge the collaboration of David Tarboton, Jonathan L. Goodall, Shilvi Satpati, YoungDon Choi, Ayman Nassar, Binata Roy, Iman Maghami, Zhiyu Li, Anthony Castronova, and Nicole Brewer.

Data availability

Data will be made available on request.

References

[1] T. Kluyver, B. Ragan-Kelley, F. Pérez, B.E. Granger, M. Bussonnier, J. Frederic, K. Kelley, J.B. Hamrick, J. Grout, S. Corlay, et al., Jupyter notebooks-a publishing format for reproducible computational workflows, Position. Power Acad. Publ.: Play. Agents Agendas 2016 (2016).
[2] Apache Foundation, Apache zeppelin, 2022, https://zeppelin.apache.org/, Online; (Accessed 31 May 2022).
[3] D. Merkel, et al., Docker: lightweight linux containers for consistent development and deployment, Linux J. 239 (2) (2014) 2.
[4] P.J. Guo, D. Engler, CDE: Using system call interposition to automatically create portable software packages, in: USENIX'11, USENIX Association, Berkeley, CA, USA, 2011.
[5] D.H. Ton That, G. Fils, Z. Yuan, T. Malik, Sciunits: Reusable research objects, in: IEEE EScience, Auckland, New Zealand, 2017.
[6] F. Chirigati, R. Rampin, D. Shasha, J. Freire, ReproZip: Computational reproducibility with ease, in: SIGMOD'16, 2016, pp. 2085–2088.
[7] V. Stodden, F. Leisch, R. Peng, Implementing reproducible research, in: Chapman & Hall/CRC The R Series, Taylor & Francis, 2014.
[8] B.T. Essawy, J.L. Goodall, M.M. Morsy, W. Zell, J. Sadler, T. Malik, Z. Yuan, D. Voce, Achieving reproducible computational hydrologic models by integrating scientific cyberinfrastructures, in: 9th International Congress on Environmental Modelling and Software, 2018.
[9] B.T. Essawy, J.L. Goodall, D. Voce, M.M. Morsy, J.M. Sadler, Y.D. Choi, D.G.T. Tarboton, T. Malik, A taxonomy for reproducible and replicable research in environmental modelling, Environ. Model. Softw. (2020) 104753.
[10] S. Lau, I. Drosos, J.M. Markel, P.J. Guo, The design space of computational notebooks: An analysis of 60 systems in academia and industry, in: 2020 IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC, IEEE, 2020, pp. 1–11.
[11] HydroShare, HydroShare, 2022, https://www.hydroshare.org/, Online; (Accessed 31 May 2022).
[12] CUAHSI, CUAHSI JupyterHub, 2022, https://www.cuahsi.org/, Online; (Accessed 31 May 2022).
[13] CyberG.I.S. Center for Advanced Digital and Spatial Studies, Cybergis-jupyter for water, 2022, https://go.illinois.edu/cybergis-jupyter-water, Online; (Accessed 31 May 2022).
[14] Xeus Cling, Xeus cling, 2022, URL https://github.com/jupyter-xeus/xeus-cling, Online; (Accessed 27 January 2025).
[15] Xeus SQL, Xeus SQL, 2022, URL https://github.com/jupyter-xeus/xeus-sql, Online; (Accessed 27 January 2025).
[16] T. Harter, B. Salmon, R. Liu, A.C. Arpaci-Dusseau, R.H. Arpaci-Dusseau, Slacker: Fast distribution with lazy docker containers, in: 14th {USENIX} Conference on File and Storage Technologies ({FAST} 16), 2016, pp. 181–195.
[17] T. Shaffer, N. Hazekamp, J. Blomer, D. Thain, Solving the container explosion problem for distributed high throughput computing, in: 2020 IEEE International Parallel and Distributed Processing Symposium, IPDPS, IEEE, 2020, pp. 388–398.
[18] Q. Pham, T. Malik, B. Glavic, I. Foster, LDV: Light-weight database virtualization, in: ICDE'15, 2015, pp. 1179–1190, http://dx.doi.org/10.1109/ICDE.2015.7113366.
[19] Q. Pham, T. Malik, D.H.T. That, A. Youngdahl, Improving reproducibility of distributed computational experiments, in: Proceedings of the First International Workshop on Practical Reproducible Evaluation of Computer Systems, 2018, pp. 1–6.
[20] Z. Yuan, D.H. Ton That, S. Kothari, G. Fils, T. Malik, Utilizing provenance in reusable research objects, Informatics 5 (1) (2018) http://dx.doi.org/10.3390/informatics5010014.
[21] Sciunit, 2022, https://github.com/depaul-dice/sciunit, Online; (Accessed 1 September 2022).
[22] FLINC, FLINC, 2022, URL https://github.com/depaul-dice/Flinc, Online; (Accessed 27 January 2025).
[23] Flatpak, Flatpak, 2022, URL https://flatpak.org, Online; (Accessed 27 January 2025).
[24] Apple Inc., MacOS Bundles, 2022, URL https://developer.apple.com/documentation/bundleresources, Online; (Accessed 27 January 2025).
[25] M.S. Islam, T. Azaz, R. Ahmad, A.S.M.S. Hossain, F. Baig, S. Wang, K. Lannon, T. Malik, D. Thain, Backpacks for notebooks: Enabling containerized notebook workflows in distributed environments, in: 2025 IEEE 21stth International Conference on E-Science, E-Science, IEEE, 2025.
[26] S. Wiki, Collecting provenance in SPADE, 2021, URL https://github.com/ashish-gehani/SPADE/wiki/Collecting-provenance, Online; (Accessed 27 January 2025).
[27] J. Chuah, M. Deeds, T. Malik, Y. Choi, J.L. Goodall, Documenting computing environments for reproducible experiments, in: Parallel Computing: Technology Trends, IOS Press, 2020, pp. 756–765.

[28] Docker Inc., Docker, 2019, https://www.docker.com/, Online; (Accessed 27 January 2025).
[29] G.M. Kurtzer, V. Sochat, M.W. Bauer, Singularity: Scientific containers for mobility of compute, PloS One 12 (5) (2017) e0177459.
[30] A. Saha, T. Denning, V. Srikumar, S.K. Kasera, Secrets in source code: Reducing false positives using machine learning, in: 2020 International Conference on COMmunication Systems & NETworkS, COMSNETS, 2020, pp. 168–175, http://dx.doi.org/10.1109/COMSNETS48256.2020.9027350.
[31] R. Feng, Z. Yan, S. Peng, Y. Zhang, Automated detection of password leakage from public GitHub repositories, in: Proceedings of the 44th International Conference on Software Engineering, ICSE '22, Association for Computing Machinery, New York, NY, USA, 2022, pp. 175–186, http://dx.doi.org/10.1145/3510003.3510150.
[32] Y.A. Gao, J. Basney, A. Withers, SciTokens SSH: Token-based authentication for remote login to scientific computing environments, PEARC '20, Association for Computing Machinery, New York, NY, USA, 2020, pp. 465–468, http://dx.doi.org/10.1145/3311790.3399613.
[33] B. Aydemir, J. Basney, B. Bockelman, J. Gaynor, D. Weitzel, SciAuth: A lightweight end-to-end capability-based authorization environment for scientific computing, in: Practice and Experience in Advanced Research Computing 2022: Revolutionary: Computing, Connections, You, PEARC '22, Association for Computing Machinery, New York, NY, USA, 2022, http://dx.doi.org/10.1145/3491418.3535160.
[34] E. Bisong, E. Bisong, Google colaboratory, in: Building Machine Learning and Deep Learning Models on Google Cloud Platform: a Comprehensive Guide for Beginners, Springer, 2019, pp. 59–64.
[35] K. Keahey, J. Mambretti, P. Ruth, D. Stanzione, Chameleon: a large-scale, deeply reconfigurable testbed for computer science research, in: 2019 IEEE 27th International Conference on Network Protocols, ICNP, IEEE, 2019, pp. 1–2.
[36] B. Ragan-Kelley, C. Willing, F. Akici, D. Lippa, D. Niederhut, M. Pacer, Binder 2.0-reproducible, interactive, sharable environments for science at scale, in: Proceedings of the 17th Python in Science Conference, 2018, pp. 113–120.
[37] R.L.F. Cunha, L.C. Villa Real, R. Souza, B. Silva, M.A.S. Netto, Context-aware execution migration tool for data science jupyter notebooks on hybrid clouds, in: 2021 IEEE 17th International Conference on EScience, EScience, 2021, pp. 30–39, http://dx.doi.org/10.1109/eScience51609.2021.00013.
[38] P.J. Guo, CDE: Run any Linux application on-demand without installation, in: LISA'11, USENIX Association, Berkeley, CA, USA, 2011.
[39] Q. Pham, T. Malik, I. Foster, Using provenance for repeatability, in: TaPP'13, USENIX Association, Berkeley, CA, USA, 2013, pp. 1–4.
[40] F. Chirigati, D. Shasha, J. Freire, ReproZip: Using provenance to support computational reproducibility, in: Proceedings of the 5th USENIX Workshop on the Theory and Practice of Provenance, TaPP '13, USENIX Association, Berkeley, CA, USA, 2013, pp. 1:1–1:4.
[41] Y. Janin, C. Vincent, R. Duraffort, CARE, the comprehensive archiver for reproducible execution, in: Proceedings of the 1st ACM SIGPLAN Workshop on Reproducible Research Methodologies and New Publication Models in Computer Engineering, TRUST '14, ACM, New York, NY, USA, 2014, pp. 1:1–1:7.
[42] P. Ivie, D. Thain, PRUNE: A preserving run environment for reproducible scientific computing, in: E-Science (E-Science), 2016 IEEE 12th International Conference on, IEEE, 2016, pp. 61–70.
[43] B. Ball, Highlighting key new capabilities for major AHV update in AOS 6.7, 2023, https://www.nutanix.com/blog/highlighting-key-new-capabilities-for-major-ahv-update-in-aos-6-7, Online; (Accessed 12 September 2024).
[44] P.I. GmbH, Parallels remote application server administrator's guide, 2024, https://download.parallels.com/ras/v19/docs/en_US/Parallels-RAS-19-Administrators-Guide.pdf, Online; (Accessed 12 September 2024).
[45] R. Ahmad, N.N. Manne, T. Malik, Reproducible notebook containers using application virtualization, in: 2022 IEEE 18th International Conference on E-Science (E-Science), IEEE, 2022, pp. 1–10.
[46] O. Standard, Topology and orchestration specification for cloud applications version 1.0, 2013.
[47] K. Zielnicki, Nodebook, 2017, URL https://multithreaded.stitchfix.com/blog/2017/07/26/nodebook/, Online; (Accessed 10 July 2021).
[48] D. Koop, J. Patel, Dataflow notebooks: encoding and tracking dependencies of cells, in: 9th USENIX Workshop on the Theory and Practice of Provenance, TaPP 2017, 2017, 17–17.
[49] K. Keahey, J. Anderson, Z. Zhen, P. Riteau, P. Ruth, D. Stanzione, M. Cevik, J. Colleran, H.S. Gunawi, C. Hammock, et al., Lessons learned from the chameleon testbed, in: 2020 USENIX Annual Technical Conference, USENIX ATC 20, 2020, pp. 219–233.
[50] M. Brachmann, W. Spoth, O. Kennedy, B. Glavic, H. Mueller, S. Castelo, C. Bautista, J. Friere, Your notebook is not crumby enough, replace it, in: Conference on Innovative Data Systems Research, CIDR, 2020.
[51] A. Rule, A. Birmingham, C. Zuniga, I. Altintas, S.-C. Huang, R. Knight, N. Moshiri, M.H. Nguyen, S.B. Rosenthal, F. Pérez, et al., Ten simple rules for reproducible research in jupyter notebooks, 2018, arXiv preprint arXiv:1810.08055.