Reproducible Notebook Containers using Application Virtualization

Raza Ahmad[a], Naga Nithin Manne[b], Tanu Malik[a]
[a]School of Computing, DePaul University, Chicago, IL, USA
[b] Argonne National Laboratory, Lemont, IL, USA
Email: {*sra0nasir@gmail.com, nithinmanne@gmail.com, tanu.malik@depaul.edu*}

2022 IEEE 18th International Conference on e-Science (e-Science) | 978-1-6654-6124-5/22/$31.00 ©2022 IEEE | DOI: 10.1109/ESCIENCE55777.2022.00015

***Abstract*—Notebooks have gained wide popularity in scientific computing. A notebook is both a web-based interactive front-end to program workflows and a lightweight container for sharing code and its output. Reproducing notebooks in different target environments, however, is a challenge. Notebooks do not share the computational environment in which they are executed. Consequently, despite being shareable they are often not reproducible. The application virtualization (AV) method enables shareability and reproducibility of applications in heterogeneous environments. AV-based tools, however, encapsulate non-interactive, batch applications. In this paper, we present `FLINC`, a user-space method and tool for creating reproducible notebook containers. `FLINC` virtualizes the notebook process that enables interactive computation and creates notebook containers, which include the environment and all data dependencies accessed by the notebook file. It relies on provenance collected during virtualization to ensure the correct behavior of a notebook when run repeatedly in different environments. We demonstrate how `FLINC` exports notebook containers seamlessly to non-notebook environments. Our experiments show that `FLINC` creates lighter weight containers as compared to equivalent non-interactive, batch containers, and preserves the same interactive workflow for the user as in current notebook platforms.**

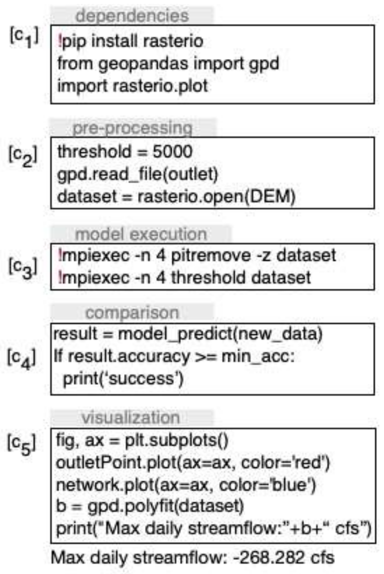
I. Introduction

Computational notebooks (e.g., Jupyter [1] or Apache Zeppelin [2]) have become a popular choice for scientific computing. Notebooks support interactive development of workflows in which users get immediate feedback on executed parts of a workflow. This is useful as users can test and debug workflows as they develop. The programming style induced by notebooks is also contrary to classical workflow systems which require the entire workflow to be specified upfront. Consequently, notebooks are being used for a variety of workflows for exploring data, executing models, and visualizing results.

Sharing of notebook files (e.g. `.ipynb` files) allows other users to interactively repeat the workflow specified in the notebook. Consider Figure 1, which shows an example of a notebook file in which the workflow is specified as a sequence of code 'cells'. Each cell describes a step of the workflow computation. Data results are obtained by manually triggering cell execution, and are often part of the notebook file. Once a user shares the notebook file for reproducibility, another user runs all cells again and gets the same or similar result[1]. A user can also interactively change the workflow, such as the type of plot to be generated to further validate the result. Such sharing and repeating improves collaborative analysis.

[1]subject to non-deterministic constructs in the program.

While a notebook file is always shareable, it does not guarantee reproducibility. A notebook file is a container for code and data results; the container, however, does not include dependencies stated in the cells. Users must often download explicit dependencies, such as data dependencies mentioned in the program or dependencies mentioned in the `import` statements. But users do not automatically download implicit dependencies, such as the dependencies needed by GeoPandas and Rasterio in the notebook file of Figure 1. This creates a 'dependency hell' scenario, in which dependencies are available in the host environment in which the user originally developed the notebook, but not in the target environment.



```
!pip install rasterio
from geopandas import gpd
import rasterio.plot
```

```
threshold = 5000
gpd.read_file(outlet)
dataset = rasterio.open(DEM)
```

```
!mpiexec -n 4 pitremove -z dataset
!mpiexec -n 4 threshold dataset
```

```
result = model_predict(new_data)
If result.accuracy >= min_acc:
  print('success')
```

```
fig, ax = plt.subplots()
outletPoint.plot(ax=ax, color='red')
network.plot(ax=ax, color='blue')
b = gpd.polyfit(dataset)
print("Max daily streamflow:"+b+" cfs")
Max daily streamflow: -268.282 cfs
```

Fig. 1: An illustrative notebook file N_1 which combines script and shell execution. Shell execution is via **!** commands.

Application virtualization is a lightweight method for sharing code, data, and environment of an application that addresses 'dependency hell' scenarios. Several recent systems [3], [4], [5] 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 repeats the application in different environments. These system tools have much to offer, especially to enable computational reproducibility [6] of scientific models and collaborative analytics [7], [8] shared on notebook-based platforms. However, these systems currently assume batch workflows, which are classically developed with scripts and are non-interactive. They do not support web-based, interactive notebooks, which necessitates auditing client-server communication protocols required to enable interaction.

A straightforward approach would be to extend application virtualization to the underlying client and server modules of the notebook system. However, we observe that such an extension only enables strict reproducibility of notebooks, *i.e.,* a user can only repeat the audited notebook but not modify it, since modification introduces new communication between the client and server that was not previously audited. This is a severe limitation for notebooks which enable interactive development and in which users can add, modify, and delete cells and change their order of execution. Thus, we need to extend application virtualization in such a way such that the resulting notebook containers can be *seamlessly* shared, repeated and *flexibly* reproduced by changing code and data in different environments. Such containers must preserve the fundamental interactive property of notebooks–to add, modify, and delete cells—so that users' interaction with the new environment is at least as powerful as the original environment.

In this paper, we describe how to systematically extend application virtualization to notebook-based platforms by preserving the connection information and redirecting it as needed. To extend, `FLINC` creates compatible kernels for a notebook file, ensuring that the file is correctly audited in the host environment and reproduced in the target environment. During *audit* phase, `FLINC` creates a lightweight notebook container which includes all the dependency files that are necessary to reproduce the notebook file. During *repeat* phase, `FLINC` transparently uses the lightweight notebook container for a notebook file in the target environment. To ensure repeatable execution, `FLINC` uses the audited provenance to determine if the notebook file has changed since being audited. If it has not changed, the container is used; otherwise the redirection ceases and regular execution resumes. `FLINC` also uses the provenance audited during application virtualization to export the contents of the container to other container-like virtual environments. The export mechanism is particularly needed to enable reproducibility of notebooks on non-notebook platforms.

The current implementation of `FLINC` is for JupyterHub which is the most popular notebook platform. However, the design of `FLINC` is general; `FLINC` only relies on two properties which are generic across all notebook platforms, namely interactive computation and a client-server architecture [9]. We have used `FLINC` with HydroShare [10], a cyber infrastructure for sharing geoscience data and models. To support notebook-based analysis, HydroShare supports multiple computational environments, such as CUAHSI JupyterHub [11] and CyberGIS-Jupyter for Water (CJW) [12]. These environments have their own hardware configuration and maintain a unique set of software dependencies to support 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—to couple with other workflows or to simply take advantage of improved hardware or software in the other environment. Currently, users can create and reproduce notebooks successfully as long as they remain in the same environment. With `FLINC`, a user can easily document dependencies for a non-notebook environment or transfer packages to be executed in a notebook environment.

The rest of the paper is organized as follows: section II describes types of reproducibility issues that users face when sharing notebooks. In section III, we describe the current model of notebook execution and sharing. Section IV describes the `FLINC` system, how it extends application virtualization to notebook platforms, and how it ensures reproducible execution in heterogeneous environments. In section IV-D we describe how provenance logs are used to ensure reproducible execution and provide flexible reproducibility. We describe the implementation and performance of `FLINC` in section V, discuss its generality to other platforms and packaging solutions in section VI, highlight the related work in section VII, and conclude in section VIII with an overview and future directions.

II. Motivating Use Case

We describe the reproducibility issues that arise when sharing notebooks. Such issues come up regularly in cyber-infrastructure 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. Figure 1 shows the steps divided into cells with some sample code.

Alice develops her notebook in Python and C; it uses Python packages to perform preprocessing and visualization (steps (i), (iv) and (v)) and C and Fortran[2] binaries to run simulation model and perform the comparison (steps (ii) and (iii)). 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, PyRhessys, etc. 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/similar

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

results. Now, Alice wants to share her notebook with Bob and Charlie, who also aim to repeat her workflow. The following scenarios illustrate the reproducibility-related issues that arise at their ends:

- **Case 1: Same interface, different environments.** Alice has shared her notebook file with Bob, who also uses a notebook platform to develop and test workflows. Although Bob can open and edit Alice's notebook file, he cannot execute it. Alice's notebook uses special Python and C dependencies that are not present in Bob's environment. While Bob is aware of the dependencies (Alice has documented them for Bob) and he could install some Python dependencies via the notebook file 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 file with Charlie, who does not use a notebook platform, and prefers to work on a terminal interface. To aid Charlie, Alice has exported her notebook to a Python file, and also shared a Docker container which has all the necessary and sufficient C and Python packages. Charlie is able to successfully repeat Alice's workflow. However, he wishes to extend her model and compare the execution with his own workflow that is configured outside the container. To effectively compare, he must either import his own workflows and data to Alice's container or export Alice's shared code, data, and environment to his host environment.

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, but it is not flexible to be extended to include new simulation models and data that are available outside the environment. A solution that accounts for these two use cases must also account for the more general, 'different interface, different environment' use case.

III. Notebook Model of Execution and Sharing

We describe the current model of interactive computing within notebook platforms and sharing notebooks.

Notebook platforms operate in a client-server architecture, as shown in Figure 2. Users edit notebook files and execute in a read-eval-print-loop (REPL)-style via a web browser. In REPL-style, an interactive process reads cell-worth of code and executes it. The next read-evaluate (akin to a loop), which is specified in the next cell, proceeds from the computational state of the previous evaluation. Notebook files assign each cell evaluation with a sequence number. The notebook file preserves this sequence, and it shows a safe order for cell re-execution, thus obtaining the same result at a later time.

On notebook platforms, the server (remote or local) maintains the *kernel*, a programming language-specific interactive process that runs independently and maintains the state of the notebook file computations as it progresses. For example, Jupyter notebook platform supports the IPython kernel for

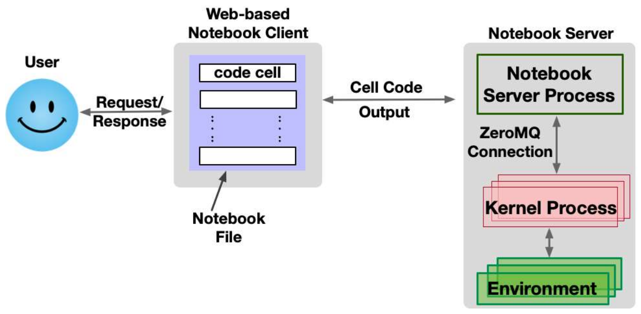


Fig. 2: A notebook platform architecture consisting of the web-based notebook client and the notebook server. A client connects to the server process, which interacts with *one* of the available kernels. The resulting kernel process is configured with an environment.

Python programs, Xeus Cling for C++ programs [13] and Xeus-sql for SQL-based programs [14].

Since the kernel maintains the state of the notebook computation, it is also aware of the computational environment and any dependencies in the environment under which the computation runs. For example, if the IPython kernel of a Jupyter platform contains Apache Spark in its environment, then possible notebooks that will necessarily compute and execute are notebooks that include the Spark library and use the specific version of Python interpreter used in the IPython kernel. If the notebook specified in Figure 1 is executed within this platform, it will report dependencies such as GeoPandas and Rasterio, as missing. Typically, for executing a variety of notebooks, the kernel is configured to run in a container or a virtual environment (such as Anaconda) that is aware of the dependencies and software packages required to support the notebooks.

Since notebook files do not transport environments, users adopt certain practices to make notebooks repeatable across heterogeneous environments. One practice is to only use Python dependencies in a notebook. With this restriction, users can install Python dependencies in their user space of the target environment. For example, in the notebook in Figure 1, 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 Docker containers and share the notebook file as part of a container image.

Neither of the 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 languages, 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.

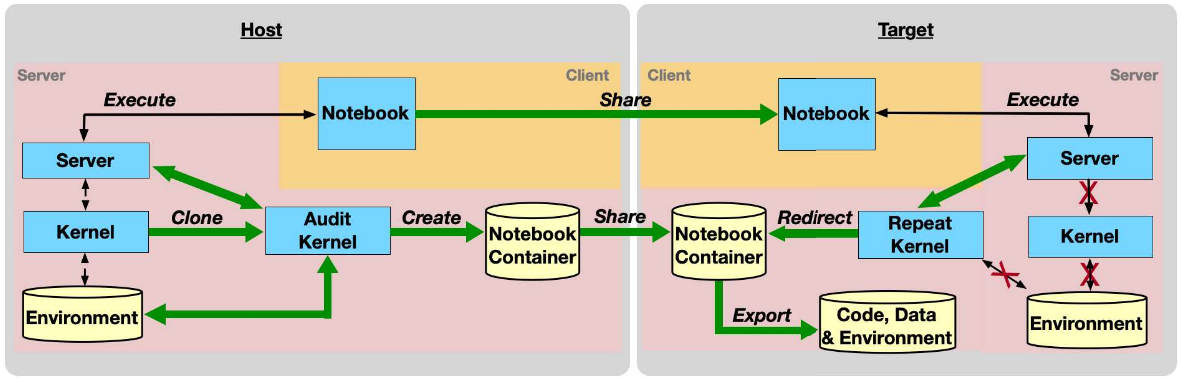


Fig. 3: The workflow of notebook reproducibility using Audit and Repeat Kernels on two different computational environments. The solid green arrows represent FLINC instructions.

IV. FLINC: Lightweight Notebook Containers

Figure 3 shows the FLINC architecture which creates, repeats, and exports notebook containers in different environments. Single dashed arrows show normal execution and solid double arrows show FLINC control and data flow. The term *notebook container* is generic—it only refers to all necessary and sufficient dependencies required by the notebook file. In practice, it can be an encapsulated package or a namespace isolated container; we do not distinguish between the containerization medium.

A. *Creating and Repeating Notebook Containers*

FLINC adapts and 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 for resources. *strace* internally relies on the *ptrace* system call to attach itself to the executing program, which intercepts system calls to the operating system for resources such as access to files and spawning of processes. The accessed resources are copied and virtualized into a container-like package [3].

AV can also be used to audit an interactive program or audit client-server programs that include network connections, since network connections also take place via system calls. Unlike previous work [15], [16], we do not need to audit both the client and the server in notebook platforms, since the kernel performs all the computation. The client program is the notebook file which is already shareable. Therefore, in FLINC, we extend AV to the server-side kernel only.

In extending AV to the kernel, we distinguish between two primary, but independent, features of a notebook platform: supporting REPL-based interactive development, and sharing of notebook files. We observe that sharing is distinct, and often follows interactive development. Virtualizing the kernel during interactive development is redundant as the user does not wish to share a half-baked program. Therefore, to virtualize for sharing, FLINC creates two additional kernels: an *audit* kernel and a *repeat* kernel. Thus, for example, if the notebook file of Figure 1 chose *Python-3.4* as the kernel for interactive development, then FLINC will create two additional kernels, *Python-3.4-audit* and *Python-3.4-repeat*, for virtualization and sharing.

The *audit* kernel is a clone of the default kernel except the interactive programming language-specific process spawned by the default kernel is now observed using *strace* and any access for a file resource is encapsulated in a package. By observing this language-specific process which executes code from the notebook file, and is therefore aware of the environment variables and any datasets or binaries accessed by the notebook file, the *audit* kernel is able to encapsulate all the necessary and sufficient dependencies.

The *audit* kernel creates a notebook container which contains all the dependencies that a notebook file accesses. A crucial part of this container is the connection information used by the notebook to connect to the kernel process. However, an important dependency missing from it is the notebook file itself. This dependency is missing because the audit kernel — owing to client-server architecture — is unaware of the notebook file and only executes the code that is communicated via socket messages. While in our current implementation, the user can use FLINC to add the notebook file to the container post execution, notebook files can continue to be shared outside the container as they are routinely shared. The absence of the notebook file in the container, however, creates issues when repeating since files may change externally. We address such issues in Section IV-B.

To repeat the notebook file in a new target environment, it must execute using the dependencies from the notebook container. In general, the notebook file by default will attempt to make a connection to the kernel process resident in the target environment. To reroute this connection to the notebook container, the repeat kernel substitutes the old connection information in the container with the new connection information. The connection information is an environment variable that is typically read by application virtualization methods before creating a container. This substitution is achieved dynamically to support interactive development. The dynamic substitution method is also used to support any changes to external dependencies such as data files or newer versions of a library and accessed by notebook files. Note that such changes are not the same as changing the notebook file, which we describe in Section IV-B.

A notebook platform that hosts different kinds of kernels for interactive development can use `FLINC` to create corresponding audit kernels for each hosted kernel. Thus, if the notebook platform hosts two kernels, then `FLINC` can create two *audit* kernels. The platform, however, only needs to create only one *repeat* kernel, which is specific to the location where the notebook containers are stored.

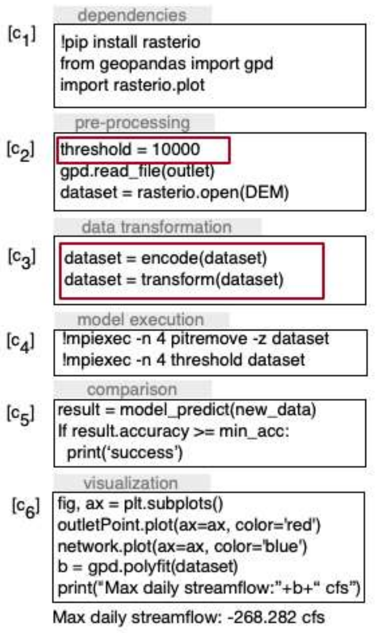
B. Ensuring Reproducible Execution with `FLINC`

In `FLINC`, the audit kernel creates the notebook container which includes all dependencies used by the notebook file, except the notebook file itself. This absence of notebook file in the container creates two issues when using the container to repeat the notebook file: First, it is not possible to map the container contents to the specific version of a notebook file which was initially used to generate the container. Consequently, given a notebook file, the container may execute, but it is not possible to guarantee that execution is the same as during audit or repeat. Second, the outputs of the container are included in the notebook file and not the container. These outputs are often necessary to compare reproducible execution. We illustrate these issues through a use case.

Consider the notebook file N_1 of Figure 1, which Alice intends to share with Bob. For this, Alice has containerized N_1 using the audit kernel, which now contains all the referenced files such as *python3.8*, *pytorch.py*, *resent18.model*, *weather2016-2018.dat*, and other configuration files. She shares with Bob the container and the notebook file N_1. Bob is able to successfully repeat N_1 and then creates another version of the file N_2 by modifying N_1's threshold parameter and by adding a data transformation step to the training dataset, as shown in Figure 4. N_2 executes without any failures but a different threshold value modifies the generated output file, and the addition of data transformation steps change the internal system call sequence. Since the result of N_2 is similar to that of N_1, the user might think that the notebook reproduced correctly. However, there is no valid means to compare the result produced from the previous execution as outputs are sent via messages. Consequently, the container does not map its contents to the specific version of a notebook file, which was initially used to generate that container.

We solve this issue by auditing *per cell* provenance of the application during 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 file consisting of cells $N = [c_1, c_2, ..., c_n]$, where c_i represents the i^{th} execution of the cell.



```
!pip install rasterio
from geopandas import gpd
import rasterio.plot
```

```
threshold = 10000
gpd.read_file(outlet)
dataset = rasterio.open(DEM)
```

```
dataset = encode(dataset)
dataset = transform(dataset)
```

```
!mpiexec -n 4 pitremove -z dataset
!mpiexec -n 4 threshold dataset
```

```
result = model_predict(new_data)
If result.accuracy >= min_acc:
 print('success')
```

```
fig, ax = plt.subplots()
outletPoint.plot(ax=ax, color='red')
network.plot(ax=ax, color='blue')
b = gpd.polyfit(dataset)
print("Max daily streamflow:"+b+" cfs")
```

```
Max daily streamflow: -268.282 cfs
```

Fig. 4: The notebook N_2 is the modified version of notebook N_1 in Figure 1. Bob changes the threshold and adds cell c3 containing data transformation.

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 point — intuitively, it is all the contents of the memory associated with the program. So, for example, for the notebook N_2, the corresponding program states are $[ps_0, ps_1, ..., ps_5]$, 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 and 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, and
- **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 and finally (iii) the hashes h_i of the associated external data dependencies. 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[3].

To establish reproducible execution, FLINC stores the details about a notebook file in the container during audit time. 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 (note that state lineage of i^{th} cell depends on state lineage of previous cell). We state this formally as:

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 (i) $h_i = h_j$, (ii) $g_i = g_j$.*

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 in FLINC, system events are audited at the level of system calls, there are some pre-processing steps that are necessary to establish equality, such as accounting for partial orders, abstracting real process identifiers. We describe these issues in Section IV-D.

C. Flexible Reproducibility with FLINC

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.

FLINC extends application virtualization to *export* the contents of a notebook container to non-notebook isolated environments. By exporting, FLINC creates a new environment for the user, similar to the notebook container, but into which users can add or remove files or change existing notebook files. Figure 3 shows the *export* action in FLINC.

The export is based on a log of all files, collected as part of the auditing phase. For correctness, FLINC does not copy the files in the log, but collects the lineage 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. Export 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 lineage 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 export determines the appropriate software packages and libraries used by the notebook application, 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 challenging and depends on some programming language-specific rules. Identifying the specific version of a software package is also not a 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 manager. For example in Python, two popular packaging conventions are *wheels* and *eggs* which respectively use the dist-info and egg-info formats for storing the 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 to have that folder and we found several packages which did not contain the folder. We then search for a file named version.py or a similar name in the installed package directory and scan that file to find the version of the package. Nevertheless, for prominent programming languages such as C, C++, R, and Fortran, FLINC follows standard conventions as a best effort to find version numbers, though the conventions do not guarantee identification.

After determining the packages, FLINC instantiates new virtual environments. In particular, FLINC 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 are installed in a user-configured location such as *tmp* or *usr*.

D. 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. In FLINC these events are captured on a *per cell* basis in a provenance log. Each line in the log file corresponds to a system event that took place as a result of an instruction executed within that cell, and contains the instruction timestamp, process identifier, instruction type, and the operands of that instruction. This log represents the lineage of the interactive kernel process modeled as an activity.

In export, 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 as system files that are read by activities determine which packages to install. We note that the log is noisy in that it also contains information about temporary files, outputs, and process memory execution. We filter such files as this is execution-specific information and not relevant for determining which packages to install.

For reproducible execution, we must establish state equality per Definition 1. Lineage equality implies that at end of cell i of version N_1, g_i is the same as that at end of cell i

[3]Such hashes are typically obtained while copying content in the container.

Notebook	Area	Size w/ Output	Size w/o Output	Primary Dependencies	Description
FlowFromSnow	Hydrology	105K	6.5K	NumPy, Matplotlib	Simple linear regression analysis on USGS stream-flow 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.
NatGas	Data Science	250K	40K	NumPy, Matplotlib, openpyxl	Market analysis of natural gas by running market simulation of 35 years based on economic indicators.
AIFinance	Data Science	25K	9.1K	NumPy, Matplotlib, Scikit-learn, Keras, TensorFlow	Deep neural network to predict stock prices in the short term using financial news data.
FashionMNIST	Data Science	7.5K	7.1K	NumPy, Matplotlib, Torch Vision	Image classification using convolutional neural networks on the Fashion-MNIST dataset.

TABLE I: Characteristics of the five notebooks in the dataset used in our experiments.

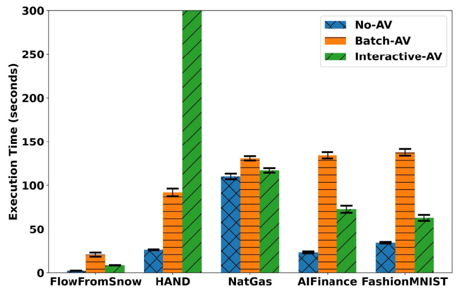
of version N_2. This is true if and only if the *sequence* of system call events (and their parameters)—till i in N_1 and i in N_2—exactly match. But if a cell, e.g., forks a child process, which itself issues system calls, then each version's sequence will contain the parent calls and the child process 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 sequence to generate the provenance. Note all time and process specific information is abstracted. Memory accesses cannot be abstracted and we just count the number of accesses in a cell.

V. Experiments

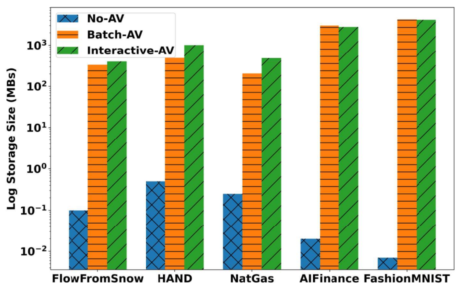
FLINC relies on application virtualization to create notebook containers. We reused application virtualization as available with Sciunit [17], [4], and modified it to support transparent connection of repeat kernel with the container. The *audit* and *repeat* kernels are wrappers over the Sciunit Python API, and FLINC adds per cell information to the resulting logs obtained via execution.

To conduct our experiments, we used HydroShare [10] 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. Two of these platforms are CyberGIS Jupyter for Water (CJW) [12] and CUAHSI JupyterHub [11]. They provide 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 these two platforms using notebooks which use Python, C, and Fortran dependencies. The experiments were run on a Docker instance with memory size of 30GB and disk size of 64GB, and running Ubuntu 20.04 on a 64 bit machine with 8 CPU cores.

Our dataset of experiments consists of five notebooks. They are written in Python and C language and also invoke utilities via shell commands. Each notebook corresponds to a different use case; two from the field of hydrology and three from data 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,



(a) Notebook Execution Time

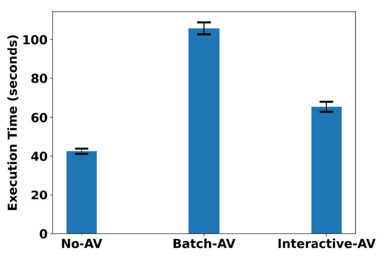


(b) Notebook Storage Size

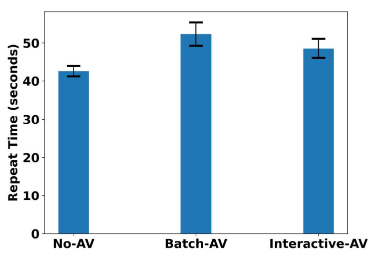
Fig. 5: Comparison of execution times and storage size for the five notebooks using each of the three methods: running the Python file with Sciunit, running the notebook with its own kernel, and running the notebook with Sciunit kernel.

two notebooks use machine learning algorithms to analyze and predict market behavior and the third notebook performs image classification using neural networks. We describe the important characteristics of our notebooks in Table I.

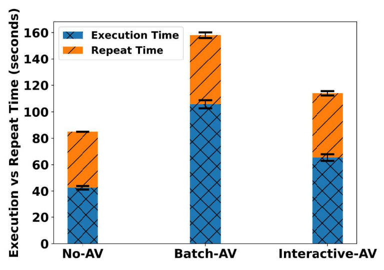
We perform two different experiments with FLINC using our dataset. First, we study the overhead incurred during notebook execution due to application virtualization. We analyze the execution time and storage size in both the audit and repeat modes of notebook execution using the *audit* and *repeat* kernel respectively. We term this the **Interactive-AV** method. We compare the performance of the **FLINC** method against two other methods. The first method is executing a Python file that corresponds to the notebook, but one that uses the



(a) Notebook Execution Time

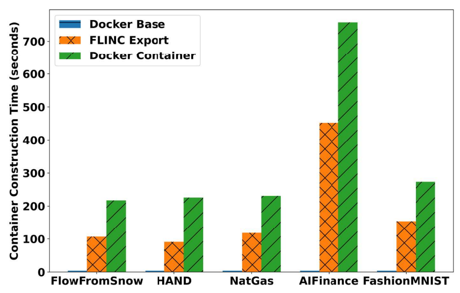


(b) Notebook Repeat Time

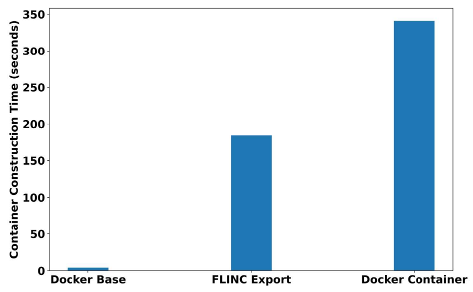


(c) Notebook Execution vs Repeat Time

Fig. 6: Comparison of average execution and repeat times using each of the three methods: running the Python file with Sciunit, running the notebook with its own kernel, and running the notebook with Sciunit kernel.



(a) Container Export Time



(b) Average Export Time

Fig. 7: Comparison of average export times using each of the three methods: creating virtual environment with Sciunit export, creating Docker base container, and creating the Docker container with the required dependencies.

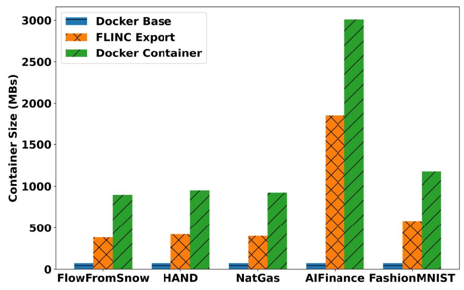
application virtualization in batch mode. We term this method **Batch-AV**. Comparison with this method will help us learn the overheads, if any, due to interactive computation. The second method is the base performance of the notebook using the original kernels (i.e, without application virtualization). We term this method **No-AV** method.

Figure 5 shows the execution time and the corresponding storage footprint generated by the three methods for each notebook. Figure 5a shows that the execution time using **Interactive-AV** for each notebook, except HAND, is less than running the Python file of the same notebook using **Batch-AV**. We speculate that this behavior is due to notebook server optimizations for the MPI modules used by HAND and the context switch for the MPI shell commands called from the Python code. For each container created using **Interactive-AV**, there is very little to no increase in the container size as compared to the containers created using **Batch-AV**, as shown in Figure 5b. We also compute the average time it takes across all notebooks and across multiple (3 runs) runs of each notebook to execute and repeat using the three approaches. We observe that running a notebook with **Interactive-AV** is $\sim$38% faster than running the Python file of the same notebook using **Batch-AV**, and has low overhead over the base execution time of the notebook using its own kernel, as shown by Figure 6a. Comparing the time to repeat the notebooks, **Interactive-AV** spends $\sim$7% less time than **Batch-AV**, as shown in Figure 6b. Comparing with itself, **Interactive-AV** spends about 26% less time to repeat the notebook than to execute the same notebook during audit mode, shown by Figure 6c. With **No-AV**, the repeat and execution times are exactly the same.

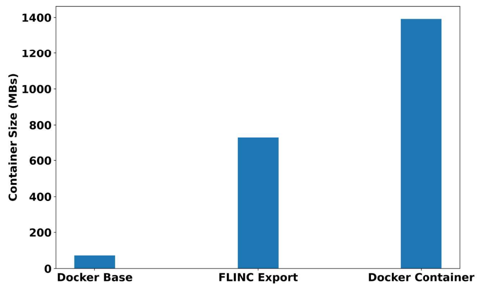
In our second experiment, we compute the time taken to create the new virtual environment using the *export* functionality in FLINC. We term this method **FLINC Export** and we compare it with the time required to create a corresponding Docker container using the same set of dependencies. We term that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

Figures 7 and 8 show the results that FLINC requires $\tilde{4}6$% less time than Docker to create a new virtual environment on the target system and install the required dependencies. The virtual environment created by FLINC is 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 FLINC.

In summary, executing and repeating a notebook using FLINC adds little overhead in terms of execution time compared to running the notebook itself, and uses less time compared to running the notebook with **Batch-AV** method. Compared to Docker containers, FLINC takes almost half the time to create a new virtual environment and about half the storage footprint to store the virtual environment.



(a) Container Export Size



(b) Average Export Size

Fig. 8: Comparison of average sizes using each of the three methods: creating virtual environment with Sciunit export, creating Docker base container, and creating the Docker container with the required dependencies.

VI. Discussion

We discuss the generality of FLINC while encapsulating and sharing dependencies. FLINC is an open-source system [18]. Currently, we do not focus on the medium of containerization or the format of the resulting container, which is beyond the scope of this paper. The objective in FLINC is to *automatically* determine the dependencies and provide a sandbox instead of manually creating a package and then sandboxing it. If the container is a package then one of the emerging format such as Flatpak [19] or NEXTSTEP/MacOS bundles [20] can be used. If the container has namespace isolation, then Docker or Singularity images can be used.

FLINC currently depends on *ptrace*, a process tracing utility that is only available on Linux platforms. While this restricts FLINC to Linux, the fundamental concepts in FLINC can also be extended to other operating systems through the use of event tracing systems specific to them. For example, as shown previously [21] 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. Thus the fundamental concepts of FLINC can be applied to other operating systems as well.

VII. Related Work

Notebooks are increasingly becoming part of cyberinfrastructure for scientific computing [22], [10]. Policies and standards with respect to reproducible notebooks [23] are also emerging. Several current efforts aim to determine the most appropriate notebook platform for scientific computing and data exploration. [9] provides an excellent survey of about 60 different notebook platforms. They state REPL-style interactive computation and client-server architecture as two distinctive properties of notebook platforms. We have developed FLINC keeping these two properties as invariant.

There are several notebook extensions that improve the reproducibility of notebooks during interactive development. Nodebook [24] is a plugin for Jupyter that checkpoints notebook state in between cells to force in-order cell evaluation; Dataflow notebooks [25] extend Jupyter with immutable identifiers for cells and the capability to reference the results of a cell by its identifier. Both of these are specialized notebook clients that aim to impose a strict order of notebook execution and capture their provenance. This order is important during repeated interactive development. Use of FLINC is post interactive development for collaborative analytics.

The issue of reproducing notebooks in different environments was already recognized by [9]. [26] addresses this problem by context-aware migration of a notebook cell for execution on another platform. 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. FLINC focuses on the reproducibility of the entire notebook with minimal user intervention.

Several methods describe the process and specifications to capture programs executions and convert them into self-contained environments. They include methods such as PRUNE [27] and TOSCA [28] which require users to explicitly define all the code, data, and dependencies with their system and users often need to learn new standards and languages to use them. In our work, we use application virtualization which 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 [3], [29]. Some prominent tools that use AV are Sciunit [4], [30], Reprozip [31], and Care [32].As this paper shows, AV must be extended to apply to notebook platforms. Further as we show in this paper, for notebooks, the redirection model must also be extended when repeating within notebook platforms due to the client-server architecture. The export method is similar to the copying of dependencies used in [33], [34], but we make it further flexible by bypassing calls to files that are not within the container.

VIII. Conclusion

Notebooks are increasingly becoming popular in scientific computing, and are adopted for programming analysis, modeling, and visualization tasks. Reproducing notebooks is critical as target environment continues to evolve, especially in collaborative analytics. In this paper, we have highlighted that notebook files do not transport their underlying environments despite being easy to share. To address this limitation, we have presented `FLINC` a system that supports interactive and flexible reproducibility of notebooks. `FLINC` adapts and extends application virtualization (AV), an already popular method for addressing computational reproducibility, to notebooks. `FLINC` monitors notebook execution and captures them to create isolated notebook containers. It extends AV to accept new network connections from a notebook file and redirect them to the notebook container. We show how `FLINC` guarantees reproducibility despite new connections in target environments, and also exports notebook containers. Experiments show that `FLINC` 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 `FLINC` to multiple OSes and make `FLINC` available for high-performance computing workflows.

Acknowledgements

This work is supported by National Science Foundation 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, and Anthony Castronova.

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." *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, vol. 2016, 2016.

[2] Apache Foundation, "Apache zeppelin," https://zeppelin.apache.org/, 2022, [Online; accessed 31-May-2022].

[3] P. J. Guo and D. Engler, "CDE: Using system call interposition to automatically create portable software packages," in *USENIX'11*. Berkeley, CA, USA: USENIX Association, 2011.

[4] D. H. Ton That, G. Fils, Z. Yuan, and T. Malik, "Sciunits: Reusable research objects," in *IEEE eScience*, Auckland, New Zealand, 2017.

[5] F. Chirigati, R. Rampin, D. Shasha, and J. Freire, "ReproZip: Computational reproducibility with ease," in *SIGMOD'16*, 2016, pp. 2085–2088.

[6] V. Stodden, F. Leisch, and R. Peng, *Implementing Reproducible Research*, ser. Chapman & Hall/CRC The R Series. Taylor & Francis, 2014.

[7] B. T. Essawy, J. L. Goodall, M. M. Morsy, W. Zell, J. Sadler, T. Malik, Z. Yuan, and D. Voce, "Achieving reproducible computational hydrologic models by integrating scientific cyberinfrastructures," in *9th International Congress on Environmental Modelling and Software*, 2018.

[8] B. T. Essawy, J. L. Goodall, D. Voce, M. M. Morsy, J. M. Sadler, Y. D. Choi, D. G. T. Tarboton, and T. Malik, "A taxonomy for reproducible and replicable research in environmental modelling," *Environmental Modelling and Software*, p. 104753, 2020.

[9] S. Lau, I. Drosos, J. M. Markel, and 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.

[10] HydroShare, "HydroShare," https://www.hydroshare.org/, 2022, [Online; accessed 31-May-2022].

[11] CUAHSI, "CUAHSI JupyterHub," https://www.cuahsi.org/, 2022, [Online; accessed 31-May-2022].

[12] CyberGIS Center for Advanced Digital and Spatial Studies, "Cybergis-jupyter for water," https://go.illinois.edu/cybergis-jupyter-water, 2022, [Online; accessed 31-May-2022].

[13] "Xeus Cling," 2022, [Online; accessed 31-Aug-2022]. [Online]. Available: https://github.com/jupyter-xeus/xeus-cling

[14] "Xeus SQL," 2022, [Online; accessed 31-Aug-2022]. [Online]. Available: https://github.com/jupyter-xeus/xeus-sql

[15] Q. Pham, T. Malik, B. Glavic, and I. Foster, "LDV: Light-weight database virtualization," in *ICDE'15*, April 2015, pp. 1179–1190.

[16] Q. Pham, T. Malik, D. H. T. That, and 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.

[17] Sciunit, https://github.com/depaul-dice/sciunit, 2022, [Online; accessed 1-Sep-2022].

[18] "FLINC," 2022, [Online; accessed 1-Sep-2022]. [Online]. Available: https://github.com/depaul-dice/Flinc

[19] "Flatpak," 2022, [Online; accessed 1-Sep-2022]. [Online]. Available: https://flatpak.org

[20] "MacOS Bundles," 2022, [Online; accessed 1-Sep-2022]. [Online]. Available: https://developer.apple.com/documentation/bundleresources

[21] "Collecting provenance in SPADE," 2021, [Online; accessed 31-Aug-2022]. [Online]. Available: https://github.com/ashish-gehani/SPADE/wiki/Collecting-provenance

[22] 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.

[23] 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," *arXiv preprint arXiv:1810.08055*, 2018.

[24] K. Zielnicki, "Nodebook," 2017, [Online; accessed 10-July-2021]. [Online]. Available: https://multithreaded.stitchfix.com/blog/2017/07/26/nodebook/

[25] D. Koop and J. Patel, "Dataflow notebooks: encoding and tracking dependencies of cells," in *9th USENIX Workshop on the Theory and Practice of Provenance (TaPP 2017)*, 2017, pp. 17–17.

[26] R. L. F. Cunha, L. C. Villa Real, R. Souza, B. Silva, and 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.

[27] P. Ivie and 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.

[28] O. Standard, "Topology and orchestration specification for cloud applications version 1.0," 2013.

[29] P. J. Guo, "CDE: Run any Linux application on-demand without installation," in *LISA'11*. Berkeley, CA, USA: USENIX Association, 2011.

[30] Q. Pham, T. Malik, and I. Foster, "Using provenance for repeatability," in *TaPP'13*. Berkeley, CA, USA: USENIX Association, 2013, pp. 1–4.

[31] F. Chirigati, D. Shasha, and J. Freire, "Reprozip: Using provenance to support computational reproducibility," in *Proceedings of the 5th USENIX Workshop on the Theory and Practice of Provenance*, ser. TaPP '13. Berkeley, CA, USA: USENIX Association, 2013, pp. 1:1–1:4.

[32] Y. Janin, C. Vincent, and 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*, ser. TRUST '14. New York, NY, USA: ACM, 2014, pp. 1:1–1:7.

[33] "Docker," https://www.docker.com/, 2019, [Online; accessed 8-Jan-2019].

[34] "Vagrant," https://www.vagrantup.com/, 2017, [Online; accessed 10-Sep-2017].