SoftwareX 35 (2026) 102821

Contents lists available at ScienceDirect

SoftwareX

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

Original Software Publication

yProv4DV: Filling the visualization gap in reproducible research workflows

G. Padovani*, S. Fiore

University of Trento, Trento, Italy

ARTICLE INFO

Keywords:
Reproducibility
Provenance
yProv4DV
PROV-JSON
RO-crate

ABSTRACT

While results visualization is a critical phase in the communication of new academic results, plots are frequently shared without the complete combination of code, input data, execution context and outputs required to independently reproduce the resulting figures. Existing reproducibility solutions tend to focus on computational pipelines or workflow management systems, not covering script-based visualization practices commonly used by researchers and practitioners. Additionally, the minimalist nature of current Python data visualization libraries tends to speed up the creation of images, disincentivizing users from spending time integrating additional tools into these short scripts. This paper proposes yProv4DV, a lightweight library designed to enable reproducible data visualization scripts through the use of provenance information, minimizing the necessity for code modifications. Through a single call, users can track inputs, outputs and source code files, enabling saving and full reproducibility of their data visualization software. As a result, this library fills a gap in reproducible research workflows by addressing the reproducibility of plots in scientific publications.

Metadata

Code metadata.

Nr.	Code metadata description	Please fill in this column
C1	Current code version	v1.0
C2	Permanent link to code/repository used for this code version	https://github.com/HPCI-Lab/yProv4DV
C3	Permanent link to Reproducible Capsule	
C4	Legal Code License	GPLv3
C5	Code versioning system used	Git
C6	Software code languages, tools, and services used	Python
C7	Compilation requirements, operating environments & dependencies	Prov
C8	If available Link to developer documentation/manual	https://hpci-lab.github.io/yProv4DV.github.io/
C9	Support email for questions	*gabriele.padovani@unitn.it*

1. Motivation and significance

Although the widespread use of Python and open source libraries has improved the reproducibility and openness of data-driven research, reproducibility often still remains a challenge, particularly in disciplines where data visualization of results plays a critical role [1]. While modern visualization libraries in Python (e.g., Matplotlib [2], Seaborn [3], Plotly[1]) make it straightforward to produce compelling graphical outputs, they also incentivize a codebase featuring numerous small scripts [4]. Users may conduct several ad-hoc modifications to these files to produce consecutive image variations, without the ability to go back to previous versions. Although tools such as Git[2,3] aid in code versioning aspects, or focus on reproducible Python environments,[4] these programs are often so small that many modifications

* Corresponding author.
Email address: gabriele.padovani@unitn.it (G. Padovani).

[1] https://plotly.com/.
[2] https://git-scm.com/.
[3] https://wandb.ai/site.
[4] https://github.com/DavHau/mach-nix.

https://doi.org/10.1016/j.softx.2026.102821
Received 17 March 2026; Received in revised form 10 June 2026; Accepted 14 June 2026
Available online 19 June 2026
2352-7110/© 2026 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

Table 1
Comparison of yProv4DV and similar provenance tracking libraries.

Tool	Focus	Collection	Integration
yProv4DV	Viz Reproducibility	Tracks filesystem calls to find dependencies.	In code: `start_run()`
noWorkflow	Script Analysis and Lineage	Profiling and environment inspection.	Prepend to Run script: `now run script.py`
ReproZip	System Portability	Traces system calls to capture all files.	Prepend to run script: `reprozip trace python script.py`.
MLflow [19]	ML Experiment Tracking	Automatic hooking into ML library APIs.	`mlflow.autolog()` in code
CDE [20]	Linux Portability	Monitors Linux system calls during runtime.	Prefix command: `cde python script.py`

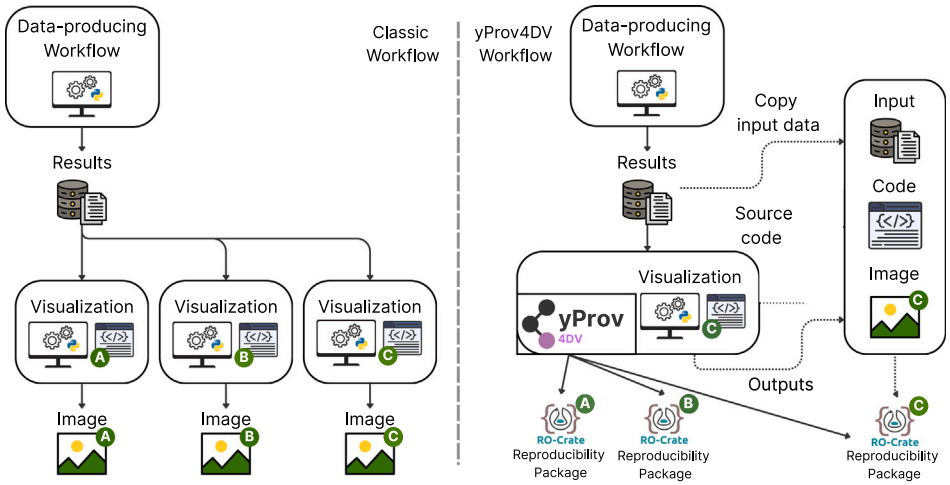


Fig. 1. A standard data visualization pipeline compared to the one proposed when using yProv4DV. Currently, a set of Python scripts is created for the visualization of plots, often with a 1:1 ratio between images and scripts. With yProv4DV, a single Python script can be used, knowing that all runs generating an output will be tracked, and can be edited and re-used according to the user needs.

go untracked [5,6]. Existing efforts toward reproducible research have focused mainly on computational workflows, containerization, or version control of code and data [7,8]. However, these approaches overlook the specific needs of visualization scripts, which are frequently lightweight and detached from formal workflow management systems [9]. Researchers commonly produce visualizations in stand-alone scripts or notebooks, which are often modified to produce variants of the initial plot. As a consequence of this, even when input data, code and the environment specifics are archived, some images might not be reproducible, and the provenance of a figure—how it was generated, from which inputs, under which parameters, and with which exact source code version—remains difficult to reconstruct. Provenance modeling standards, such as the W3C PROV data model [10,11], provide a formal framework for representing the relationships between data, tasks and agents involved in computational processes. Despite their expressive power, these standards are rarely adopted in everyday data visualization practice due to the complexity of integration and the time required to add them into small scripts [12]. Therefore, there is a clear need for tools able to incentivize the use of formal provenance practices, improving their accessibility in small and simple workflows [13,14].

Many general purpose and targeted libraries for provenance collection have been proposed over the years, with the bulk of them being targeted at workflow management systems [15], at recently expanding fields such as machine learning (ML) [16,17], and at dockerized applications [18]. Table 1 contains a comparison of the most popular libraries which are aimed at targeting reproducibility in small Python scripts, and also includes information on what these tools collect and how they can be integrated into existing code-bases.

To address the aforementioned gap, the yProv4DV (yProv for Data Visualization) library is proposed. yProv4DV is a lightweight Python utility that can automatically capture, package, and preserve the full reproducibility context of data visualization scripts. It is designed to minimize the number of directives that need to be invoked, and in most cases, simply importing the library and starting the run is sufficient. The system records the source code, input files, generated outputs, and execution metadata of a visualization run, and organizes these artifacts into a structured directory, producing a self-contained archive in the form of a RO-Crate [21], alongside W3C PROV–compliant provenance descriptions (JSON, DOT,[5] SVG). This enables interpretability through the PROV-JSON standard [22], as well as machine-readable traceability, facilitating reuse and long-term preservation of each script.

Additionally, yProv4DV is one of the components provided in the yProv ecosystem [23], which aims to provide simple tools to collect and manage provenance information. The tools in the framework, such as yProv4ML [17,24] which is designed to collect ML training information, and yProv4WFs [25], which is integrated into several workflow management systems, are fully interoperable, and can facilitate the collection of multi-level provenance data by connecting files through Persistent Identifiers (PIDs) [26,27].

In this paper, the design principles, architecture, and yProv4DV usage are presented, along with a demonstration of how this framework supports reproducibility in visualization-centric scientific workflows.

1.1. Software architecture

Rather than introducing a separate workflow engine or requiring structural changes to code, yProv4DV operates as a lightweight runtime layer that observes, records, and packages the artifacts produced during execution.

Fig. 1 illustrates the difference between a classic visualization workflow and the one enabled by yProv4DV. In a conventional setting, individual Python scripts are typically developed to generate specific figures, resulting in a one-to-one relationship between programs and output images. These source files often rely on external data and produce

[5] https://graphviz.org/.

visual artifacts without preserving the full execution context in which they were generated. Without proper tracking of all files, this can lead over time to non-reproducible images, when scripts are modified.

The workflow proposed by yProv4DV restructures this process into one that can, depending on the user's preference, be condensed into a single visualization script. Each file's execution is tracked and packaged into a self-contained directory or RO-Crate, supporting reproducibility and facilitating the development process.

The yProv4DV architecture can be broadly subdivided into three main components: execution monitoring, artifact tracking, and provenance and reproducibility packaging. The main idea behind the library is to create a runtime layer that intercepts key file operations during execution, without altering the user's code logic. The module observes file reads performed by Python scripts, and monitors file creation events that occur as the program runs. This tracking is fully transparent to the user, and can be customized to only observe specific types of files.

The artifact tracking component classifies all relevant resources involved in the visualization process into three broad categories:

- Inputs: which include files accessed during execution;
- Outputs: which are files generated by the script;
- Source code: which consists of all the Python scripts responsible for producing the visualization, including all the local modules imported by the program.

These artifacts are copied into a dedicated provenance directory and organized into three respective sub-folders. Once artifacts are identified, and the execution has ended, the library constructs a standardized description of the execution using the W3C PROV format. It models the relationships between the main activity, which is the execution of the visualization script and all the entities (input files, source code, and output images), including an implicit agent, meaning the user or execution environment. This provenance description is then exported to JSON, image and provenance graph formats, providing both machine-readable metadata and human-interpretable visual representations of the execution history. Finally, all data is packaged into a self-contained RO-Crate archive.

Each execution of the script produces its own reproducibility package, meaning that multiple runs of the same script generate distinct provenance bundles. As shown in Fig. 1, the end goal of the library is to allow users to produce, from a single reusable visualization script, multiple fully documented outputs across several runs.

1.2. Software functionalities

As mentioned, yProv4DV is designed to be integrated easily into existing Python data visualization scripts, while automatically capturing the information necessary for reproducibility and provenance tracking. The only requirement from the user, is to import the library and call the *start_run* function:

```
import yprov4dv
yprov4dv.start_run(
    run_name : str = "experiment_run",
    provenance_directory : str = "prov",
    prefix : str = "yProv4DA",
    default_namespace : str = "http://example.org/",
    create_json_file : bool = False,
    create_dot_file : bool = False,
    create_svg_file : bool = False,
    create_rocrate : bool = True,
    save_input_files : bool = True,
    save_input_files_subset : bool = False,
    skip_files_larger_than : int = 50,
    verbose : bool = False,
)
```

This will start the tracking of newly generated files, as well as ingested inputs. Additionally, all libraries and modules imported before and after the calling of *start_run* will be included in the final RO-Crate.

The behavior of the yProv4DV library can be customized by modifying the parameters in the *start_run* function. For example, *save_input_files* determines whether files tracked as input will be saved in the RO-Crate, and if *save_input_files_subset* is *True*, then the library will only save the information used in the actual plot, and not the entire input file.

As mentioned, the main feature of the library is the ability to automatically detect outputs produced during a script's execution. Any file generated by the script is automatically identified and registered as output in the provenance record. In practice, the library creates a set of snapshots of the working directory at both the start and the end of the program execution, enabling the identification of newly created and modified files. All the detected files are then copied into the provenance directory to preserve the outputs of the visualizations, as well as the processes required to reproduce them. In addition, yProv4DV provides automatic input tracking under a set of supported conditions. When a script reads a file—e.g., using Python's built-in *open()* function—the library intercepts the operation and records the accessed file as an input in the provenance record. This way the system is able to infer data dependencies directly from the script's behavior without requiring users to manually declare which datasets or configuration files are being used, and especially without changing the user's execution script. To conclude the tracking, the library identifies all modules loaded by the initial script, which could be, for example, files written by the user to provide utility functions, and stores them in a separate directory in the RO-Crate. All the files are then saved relative to their original location, which allows for the execution of the copied script without requiring additional actions from the user.

yProv4DV supports a set of popular libraries that are often used for data visualization, such as pandas [28], NumPy [29], PyTorch [30], Xarray[6] and more. In case the user plots through an unsupported library, yProv4DV might not be able to fetch the input automatically. For these situations, the *log_input* and *log_output* directives have been exposed, which allow the user to inform the module about the files to track.

```
yprov4dv.log_input(path_to_untracked_input_file : str)
yprov4dv.log_output(path_to_untracked_output_file : str)
```

In some cases, files used for visualization might be too large and copying them into an additional directory can be impractical and expensive. To address this, a file size threshold can be defined, allowing the library to ignore files exceeding a specified limit. This threshold can be configured through the *start_run* function, by setting the *skip_files_larger_than* parameter (in MB). In such cases, the reproducibility of the original script depends on whether the large input data can be shared through alternative means. To facilitate this the *save_input_files_subset* option can be set to *True*, allowing only the subset of data actually used for plotting to be stored. The library is able to track all the major plotting functions from widely used libraries, such as pandas, matplotlib and seaborn, and to identify the specific dataframes involved in the visualization. This flag does not guarantee reproducibility, but rather ensures that the subset of data required to reproduce the plotting logic is included in the final RO-Crate.

```
yprov4dv.untrack_file(path_to_tracked_output_file : str)
```

If a user wants to prevent a specific file from being tracked, then the *untrack_file* directive can be used. Both input and output files that would normally be copied into the RO-Crate, when flagged, will be ignored.

```
yprov4dv.end_run()
```

[6] https://docs.xarray.dev/en/stable/index.html.

Table 2
Comparison of execution times (in seconds) and scaling factors between the yProv4DV version (With) and the baseline (Without) for files ranging from 8Kb to 8Mb.

Metric	8Kb (s)	4Mb (s)	8Mb (s)	Scaling
Init + Closing Time	1.9014	1.9725	1.9627	1.03x
Total Time (With)	2.6361	4.5090	5.9459	2.26x
Total Time (Without)	1.1784	2.8126	4.1623	3.53x

Finally, the *end_run* function call has been defined to close the current execution and produce the complete reproducibility package. By default, the yProv4DV library calls this directive on exit, which makes it transparent to the user, however in some cases, such as when creating multiple charts in the same script, it allows each visualization to be separated into several bundles.

Table 2 presents a performance benchmark based on 30 execution cycles, quantifying the library's overhead across increasing data scales. While the total execution time scales with the file size, the data reveal that the primary source of overhead is concentrated within the initialization and closing phases, and as the plot time increases, this wait time becomes more and more negligible.

1.3. Sample code snippets analysis

The following code section shows how to integrate the yProv4DV library into an existing codebase:

```
# this can be imported even before the start_run function
from lib import elaborate
import yprov4dv
yprov4dv.start_run()

import pandas as pd
import matplotlib.pyplot as plt

# This is just to provide a reproducible script
import urllib.request
urllib.request.urlretrieve(
    "https://raw.githubusercontent.com/HPCI-Lab/" \
    "yProv4DV/main/assets/results.csv",
    "results.csv"
)

data = pd.read_csv("results.csv")
# In this case this is not necessary,
# the file will be copied automatically
yprov4dv.log_input("results.csv")

data["second_series"] = elaborate(data["points"])

data.plot()
plt.savefig("example.png")

# In this case this is not necessary,
# the file will be copied automatically
yprov4dv.log_output("example.png")
```

The only two necessary lines are the importing of the library and the *start_run* call, after which all opened and created files will be stored either as inputs or outputs. In the current example, the input file will be the one at *"./results.csv"*, and the final image is *"example.png"*. For explanatory purposes, the *log_input* and *log_output* directives are also shown, to give an example of how a user could decide to track additional files, but in this instance the calls would be redundant, as yProv4DV would track the files automatically.

Starting from the collected information, the user who produced this RO-Crate can simply execute the following commands to reproduce the package.

```
# Run the original script,
# creating the reproducibility package in "prov_0"
> python examples/main.py
# Run the saved script, without moving from the original directory
> python prov_0/src/examples/main.py
```

On the other hand, if the package is shared, the other user will have to set up the Python environment using the *requirements.txt* file, and adjust the paths in the source file according to their file system.

```
# Run the original script,
# creating the reproducibility package in "prov_0"
> python examples/main.py

# Adjust data paths (if the cwd is the same this step is not needed)
> vim prov_0/src/examples/main.py
# Install the requirements taken from the original env
> pip install -r prov_0/inputs/requirements.txt

# Run the saved script, without moving from the original directory
> python prov_0/src/examples/main.py
```

2. Illustrative examples

Fig. 2 presents a simplified graph generated from the execution of a visualization script to demonstrate how yProv4DV captures and represents provenance information. This example illustrates in practice how the library models the relationships between inputs, source code, execution metadata, and outputs according to the W3C PROV standard.

On the left side of the graph, the input entities used in the visualization process are shown. These include the datasets used to generate the figure (e.g., *results.csv*), the *requirements.txt* file describing the Python environment, and the source files (in this case *lib.py*, *simple.py*) that contribute to the execution. All these inputs are automatically detected when the program accesses them through standard file operations. Each of these entities is marked as "used" by the main activity and, in case the user explicitly logs the file through *log_input*, a parameter is specified to differentiate this possibility.

At the center of the graph is the activity node symbolizing the current run, which is the visualization script (i.e., experiment_run). This activity corresponds to a single run of the Python script tracked through yProv4DV. The activity is connected to an attribute node, containing execution metadata captured at runtime including, in this case:

- the exact command used to run the script,
- the start and end timestamps of the execution,
- and the Git commit hash associated with the source code.

On the right side of the graph, the output entities are shown (e.g., *example.png*), and are marked as being "generated" by the execution activity. The same parameter distinguishing between user-logged and library-logged is also applied here, analogous to the inputs.

An additional example of a use case of this library is provided through an European project consisting of the execution and gathering of information from small-scale HPC jobs into a dataset. Part of the data can be accessed at the AIMP-Bench repository,[7] which also includes a set of visualizations processed using yProv4DV. Starting from the elaboration of more than 300 ML trainings, an analysis phase was conducted to understand the behavior of different hardware and GPU configurations. A report[8] containing several charts has been produced, while all

[7] https://github.com/HPCI-Lab/AIMP-Bench.

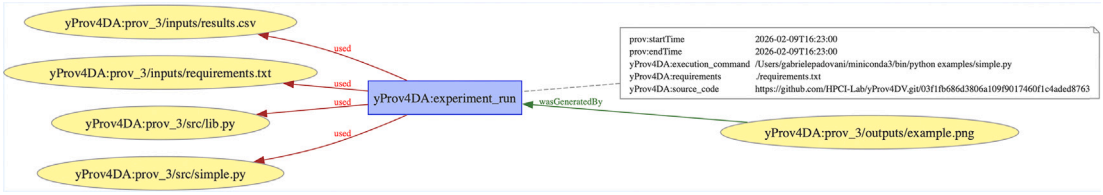


Fig. 2. A brief example of a provenance graph containing the visualization pipeline is shown. On the left, inputs are shown, including the data used (*results.csv*), the requirements for the Python environment and the modules and scripts used. On the right, the image output is shown. Attributes of the main activity include the Git hash of the source code, the execution command used and the start and end times.

the reproducibility bundles created with yProv4DV have been uploaded to the `reprod` branch of the repository.

The context of the repository shows a subset of the analysis, particularly for benchmarking the scaling capabilities of different GPUs starting from small ML use cases.

3. Impact

yProv4DV has the potential to improve reproducibility practices in domains where data visualization is frequently used to explore results [6]. The ease of integration has the chance to lower the technical barrier to provenance capture, and hopefully incentivize researchers to move from ad hoc figure generation to systematically documented visualization workflows. Many users will not switch to large workflow management systems just to integrate provenance tracking, but obtaining reproducible packages with a single additional line might drive some to try.

One of the key impacts of yProv4DV lies in making figure-level reproducibility feasible. While many reproducibility initiatives focus on entire computational pipelines, individual figures in publications often remain insufficiently documented. yProv4DV directly addresses this issue by packaging with a standardized format, each execution, its inputs, source code, and execution context. This capability enhances the integrity of scientific reporting and simplifies peer review, replication of studies, and educational reuse of published materials. The integration of W3C PROV–compliant metadata and RO-Crate packaging further improves the usability of yProv4DV, especially when users have to share their own work. The generated artifacts are interoperable with existing provenance analysis tools, lowering the barrier to the creation of standardized archives. As research infrastructures increasingly emphasize FAIR (Findable, Accessible, Interoperable, Reusable) principles [31], yProv4DV, along with the yProv ecosystem, will help contribute to making visualization scripts more FAIR-compliant, especially if each artifact can be easily uploaded with its own associated PID.

In conclusion, yProv4DV promotes Open Science principles through its provenance aware capabilities, which ensure the reproducibility of visualization workflows in scientific research and publications, without imposing additional burdens on users. Over time, we hope its integration will be able to shift the common practices toward such a provenance-aware model.

4. Conclusions and future work

In this paper, we introduce yProv4DV, a lightweight Python utility designed to address a critical gap in reproducible research: the lack of systematic provenance capture for data visualization scripts, such as those needed for scientific papers. The library is designed to track source code, inputs, outputs, and execution metadata, packaging them into W3C PROV–compliant JSON files and RO-Crates. yProv4DV provides a standard solution for Python scripts' reproducibility. Its minimal integration requirements and customization make it accessible to a wide range of users, and aim to expand the yProv ecosystem of tools [23] that improve the production and handling of provenance files, integrating it into visualization scripts without the need for complex workflow orchestration tools.

Future work related to this library will include: integration with more visualization libraries, allowing for more automatic reproducibility with a path tracing system, and a thorough validation of this library's functionality on larger use cases, in addition to the one already present for the AIMP Benchmark. Additionally, with the improvement of ML-based visualization tools [32], tools such as yProv4DV will have to evolve to keep pace with new and increasingly advanced scenarios: further work will explore interoperability with data repositories, compatibility with other existing provenance standards and data management platforms, such as MLflow servers, enabling direct publication of provenance artifacts. Finally, further evaluations to assess the effectiveness of yProv4DV in improving reproducibility practices across different research domains will be addressed.

CRediT authorship contribution statement

G. Padovani: Writing – original draft, Software, Methodology, Investigation. **S. Fiore:** Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

This work was partially funded under the National Recovery and Resilience Plan (NRRP), Mission 4 Component 2 Investment 1.4 - Call for tender No. 1031 by 17/06/2022 of the Italian Ministry for University and Research funded by the European Union – NextGenerationEU (proj. nr. CN_00000013), and the RI-SCALE project (Grant Agreement 101188168).

References

[1] Becker G, et al. trackr: a framework for enhancing discoverability and reproducibility of data visualizations and other artifacts in R. J Comput Graph Stat 2019;28(3):644–58.
[2] Hunter JD. Matplotlib: a 2D graphics environment. Comput Sci Eng 2007;9(3):90–5.
[3] Waskom ML. seaborn: statistical data visualization. J Open Source Softw 2021. [Online]. Available: 6(60):3021. https://doi.org/10.21105/joss.03021
[4] Perkel JM, et al. Data visualization tools drive interactivity and reproducibility in online publishing. Nature 2018;554(7690):133–4.
[5] Liu J, et al. Promoting and enabling reproducible data science through a reproducibility challenge. Harv Data Sci Rev 2022;4(3).
[6] Isenberg T. The state of reproducibility stamps for visualization research papers. In: 2024 IEEE evaluation and beyond-methodological approaches for visualization (BELIV). IEEE; 2024. p. 97–105.
[7] Apostal SFJ, et al. Containers and reproducibility in scientific research. In: 2018 IEEE international conference on Electro/Information technology (EIT). IEEE; 2018. p. 0525–30.
[8] Moreau D, et al. Containers for computational reproducibility. Nat Rev Methods Primers 2023;3(1):50.

[8] https://github.com/HPCI-Lab/AIMP-Bench/blob/main/AIMP.pdf.

[9] Da Silva RF, et al. A characterization of workflow management systems for extreme-scale applications. Future Gener Comput Syst 2017;75:228–38.
[10] Missier P, et al. The W3C PROV family of specifications for modelling provenance metadata. In: Proceedings of the 16th international conference on extending database technology; 2013. p. 773–6.
[11] Belhajjame K, et al. Prov-dm: the prov data model. W3C Recommendation 2013;14:15–16.
[12] Herschel M, et al. A survey on provenance: what for? What form? What from? VLDB J 2017;26(6):881–906.
[13] Silva CT, et al. Provenance for visualizations: reproducibility and beyond. Comput Sci Eng 2007;9(5):82–9.
[14] Ragan ED, et al. Characterizing provenance in visualization and data analysis: an organizational framework of provenance types and purposes. IEEE Trans Vis Comput Graph 2015;22(1):31–40.
[15] Davidson SB, et al. Provenance and scientific workflows: challenges and opportunities. In: Proceedings of the 2008 ACM SIGMOD international conference on management of data; 2008. p. 1345–50.
[16] Souza R, et al. Workflow provenance in the lifecycle of scientific machine learning. Concurr Comput Pract Exp 2022;34(14):e6544.
[17] Padovani G, et al. yProv4ML: effortless provenance tracking for machine learning systems. Softwarex 2025. [Online]. Available: 31:102298. https://doi.org/10.1016/j.softx.2025.102298
[18] Kunstmann L, et al. Scientific workflow deployment: container provenance in high-performance computing. In: Simpósio brasileiro de banco de dados (SBBD). SBC; 2024. p. 457–70.
[19] Zaharia M, et al. Accelerating the machine learning lifecycle with MLflow. IEEE Data Eng Bull 2018;41(4):39–45.
[20] Guo PJ. CDE: automatically package and reproduce computational experiments. In: Implementing reproducible research. Chapman and Hall/CRC; 2018. p. 79–111.
[21] Soiland-Reyes S, et al. Packaging research artefacts with RO-crate. Data Sci 2022;5(2):97–138.
[22] Niu X, et al. Interoperability for provenance-aware databases using {PROV} and {JSON}. In: 7th USENIX workshop on the theory and practice of provenance (TaPP 15); 2015.
[23] Padovani G, et al. A software ecosystem for multi-level provenance management in large-scale scientific workflows for AI applications. In: SC24-W: workshops of the international conference for high performance computing, networking, storage and analysis. IEEE; 2024.
[24] Padovani G, et al. Provenance tracking in large-scale machine learning systems. In: Workshop proceedings of the 54th international conference on parallel processing. New York, NY, USA: Association for Computing Machinery; 2025. p. 167–74. [Online]. Available: https://doi.org/10.1145/3750720.3757295, ICPP Workshops '25.
[25] Sacco L, et al. Enabling provenance tracking in workflow management systems. In: 2024 IEEE international conference on big data (BigData); 2024. p. 4402–9.
[26] Fiore S, et al. A graph data model-based micro-provenance approach for multi-level provenance exploration in end-to-end climate workflows. In: 2023 IEEE international conference on big data (BigData); 2023. p. 3332–9.
[27] Manzi A, et al. interTwin: advancing scientific digital twins through AI, federated computing and data. Future Gener Comput Syst 2026. [Online]. Available: 179:108312. https://www.sciencedirect.com/science/article/pii/S0167739X25006065.
[28] McKinney W, et al. pandas: a foundational Python library for data analysis and statistics. Python high perform sci comput 2011;14(9):1–9.
[29] Harris CR, et al. Array programming with NumPy. Nature 2020;585(7825):357–62.
[30] Ketkar N, et al., Introduction to PyTorch. In: Deep learning with Python: learn best practices of deep learning models with PyTorch. Springer; 2021. p. 27–91.
[31] Wilkinson MD, et al. The FAIR guiding principles for scientific data management and stewardship. Sci Data 2016;3(1):1–9.
[32] Zhu D, et al. PaperBanana: automating academic illustration for AI scientists, arXiv preprint arXiv:2601.23265. 2026.