Collecting and Analyzing Provenance on Interactive Notebooks: when IPython meets noWorkflow

João Felipe Nicolaci Pimentel
Vanessa Braganholo Leonardo Murta
Universidade Federal Fluminense
Brazil
{jpimentel,vanessa,leomurta}@ic.uff.br

Juliana Freire
New York University
USA
juliana.freire@nyu.edu

Abstract

Interactive notebooks help users explore code, run simulations, visualize results, and share them with other people. While these notebooks have been widely adopted in teaching as well as by scientists and data scientists that perform exploratory analyses, their provenance support is limited to the visualization of some intermediate results and code sharing. Once a user arrives at a result, it is hard, and sometimes impossible, to retrace the steps that led to the result, since they do not collect the provenance for intermediate resuls or of the environment. As a result, users must fulfill this gap using external tools such as workflow management systems. To overcome this limitation, we propose a new approach to capture provenance from notebooks. We build upon noWorkflow, a system that systematically collects provenance for Python scripts. By integrating noWorkflow and notebooks, provenance is automatically and transparently captured, allowing users to focus on their exploratory tasks within the notebook. In addition, they are able to analyze provenance information within the notebook, to both reason about and debug their work, using visualizations, SQL queries, Prolog queries, and Python code.

1. Introduction

Interactive Notebooks are computational environments that allow users to write documents containing code, text, plots, and other rich media. Users can perform exploratory research by running computations and visualizing their results interactively. Notebooks can be shared and converted into other formats, such as HTML or PDF. Two well known notebook environments are IPython Notebook[1] and knitr[2]. Notebooks are being widely used: the traffic to the IPython Notebook Web site suggests that more than 500,000 people actively use it [10].

IPython Notebook originated from IPython [9], a Python shell that provides powerful features for interactive scientific computing. While it helps scientists to explore, share, and keep track of their experiments interactively [10], provenance support in IPython Notebook is limited. Provenance can help scientists to interpret experiments' results, check if the experiments were executed as expected, understand the sequence of operations that lead to the results, and enable reproducibility [5]. However, provenance support in IPython Notebook only allows visualization of some intermediate cell results and interpretation of the shared code. It does not help scientists understand how the code executed. In fact, IPython Notebook is not able to collect environment provenance such as the version of the libraries used, and does not collect intermediate results within cells (e.g., different parameter values used in an exploration). Therefore, the exploratory trail is lost and the reproducibility of the results is compromised. If a scientist wants to capture more provenance about an experiment, she must export or rewrite her experiment into an external script and run an external tool to capture and analyze provenance from the script [1–4, 8, 11, 13].

We propose an approach that addresses this limitation and allows transparent provenance capture directly from the notebook. In previous work, we introduced noWorkflow [8], a tool that transparently captures provenance from Python scripts and provides mechanisms that allows users to explore this information. In this paper, we present the integration of noWorkflow and IPython to allow scientists to capture provenance from code executed inside IPython notebooks, visualize the captured provenance as graphs, and query the provenance using Prolog and SQL. We also show that notebooks are powerful tools for interactively exploring provenance.

The remainder of the paper is structured as follows. Section 2 presents both IPython and noWorkflow. Section 3 discusses how to collect provenance inside notebooks. Section 4 discusses how to analyze provenance collected by noWorkflow in notebooks. Section 5 presents related work. Finally, Section 6 concludes this paper and presents future work.

2. Background

IPython is a Python shell that interactively executes cells of code. A cell is a multi-line text input field[3]. When a cell is executed, IPython adds a number to the cell that can be further referenced by a variable. If the last line of a cell is an expression, the cell outputs the expression value. It is possible to implement various representations for the output, such as plain text, PNG, SVG, LaTeX, and HTML. These representations can be used to plot data and media on different tools, such as IPython Notebook, QtConsole, and terminal emulators. IPython Notebook fully supports all these representations, but QtConsole does not support Javascript on

[1] http://ipython.org/notebook.html

[2] http://yihui.name/knitr

[3] http://ipython.org/ipython-doc/3/notebook/notebook.html

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page.

TaPP 2015, July 8–9, 2015, Edinburgh, Scotland.
Copyright remains with the owner/author(s).

```
In [1]: values = ["noWorkflow", "IPython"]
In [2]: ["<li>{}</li>".format(i) for i in values]
Out[2]: ['<li>noWorkflow</li>', '<li>IPython</li>']
In [3]: '<ul>' + ''.join(_) + '</ul>'
Out[3]: '<ul><li>noWorkflow</li><li>IPython</li></ul>'
In [4]: from IPython.display import HTML
   ...: html_code = _3
   ...: HTML(html_code)
Out[4]:
  • noWorkflow
  • IPython
```

Figure 1. Using IPython to display an HTML list on QtConsole

the HTML and common terminals only support plain text. While QtConsole is an IPython console that behaves like a terminal and executes only cells of code, IPython Notebook is a web-based interactive computational environment where it is possible to write notebook documents. A notebook document is composed of cells, which can contain either code or text.

Figure 1 presents IPython running on QtConsole. In this example, we show how to interactively transform a Python list into an HTML list. Each *In [X]* defines a cell and each *Out[X]* defines the cell result. Note that the third cell references the second cell output through the variable '_' and the fourth cell references the third cell output through the variable '*_3*'. Note also that the fourth cell is composed by three lines of code and its result is an HTML object.

Since IPython runs on top of Python, it keeps all advantages of using Python scripts for computational science, such as a clean and simple syntax, the simplicity to integrate external tools, and to run simulations [6]. In addition, it adds many features to streamline these tasks, such as shell commands, magics, extensions, and interactive visualization.

An external program can be called by prepending an exclamation point to its name. For example, 'a = !ls' assigns a list of lines returned by *ls* command to the variable *a*. *Magics* are special functions on IPython that can modify the way it executes, create new variables, and provide more features. There are two kinds of *magics*: *line magic* and *cell magic*. A *line magic* can appear anywhere in the cell. It is prefixed with the '%' character, and uses the remaining of the line as parameter. For example, with the *line magic* '*%run script.py*', it is possible to run an external *script.py* inside a notebook. On the other hand, a *cell magic* only appears on the top of the cell, is prefixed with a double '%%', and uses the whole cell as parameter. For example, with the *cell magic* '*%%ruby*', it is possible to run ruby scripts written directly on the notebook. IPython also supports extensions that can be loaded through the *line magic* '*%load_ext ext_name*'. An extension defines an initializer that loads customs *magics* and prepares the notebook environment.

noWorkflow (**n**ot **o**nly **Workflow**) [8] is an approach that automatically captures provenance from Python scripts without requiring any modifications to the script. It was conceived to help scientists perform scientific experiments and data exploration, but can be used to run and capture provenance of arbitrary scripts in Python. Due to this reason, every run of a script in noWorkflow is called a *trial*. noWorkflow captures **definition**, **deployment**, and **execution** provenance when a script is executed and stores it. This provenance can be read for further analysis. Figure 2 presents the architecture of noWorkflow.

For every trial (a single execution of the script), noWorkflow generates an identifier and all provenance collected during the execution is stored in a database related to this identifier. It creates the database in the same directory as the script, and puts it inside a ".noworkflow" directory. The database is composed by a relational

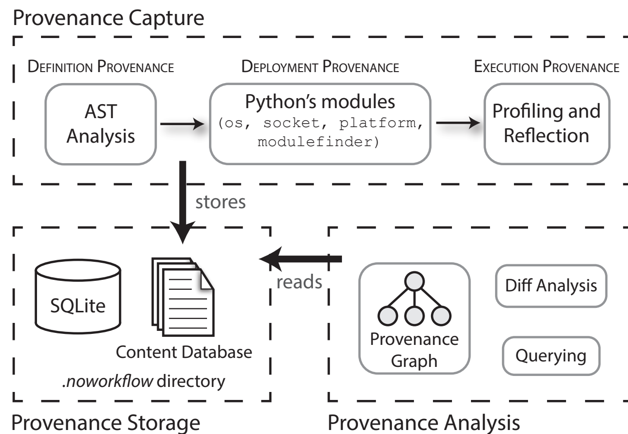

Figure 2. Architecture of noWorkflow [8]

database and a content database. The content database stores all the scripts and files used in the trial and the relational database stores the remaining provenance.

noWorkflow uses *abstract syntax tree* (AST) analysis to collect definition provenance. It identifies user defined functions, function calls, arguments, and global variables referenced in the script. noWorkflow collects deployment provenance by identifying the version of imported modules and environment variables. Finally, noWorkflow collects execution provenance through the use of profiling and reflection to identify function activations (calls), argument values, return values, global values, start and finish times for each activation, as well as their context. It also captures the content of all files manipulated by the experiment script before and after the manipulation and stores them in the content database.

Scientists can perform SQL queries on the database or export a trial as Prolog facts and perform Prolog queries to analyze captured provenance. They can also use the visualization tool provided by noWorkflow to visualize activation graphs or use the command line to compare trials during the analysis.

Figure 3 presents a small piece of Python code that we will use as example. Note that it is composed by two scripts. The main script is *script1.py* and it imports functions 'y' and 'z' from *script2.py*. These functions manipulate text files and have different delays to simulate slow functions. Running *script1.py* with noWorkflow is as simple as running it with Python: `now run script1.py`.

In this example, it is possible to see that noWorkflow will store *script1.py* and the function *x* with parameter *i* as definition provenance; environment variables, *script2* module at version 1.0.2, *random*, *time* and other Python modules imported by those modules as deployment provenance; calls to *range*, *x*, *y* and *z*, and files 'y.txt' and 'z.txt' as execution provenance. Currently, noWorkflow only captures calls on the main script. So it will not store calls to *sleep*, *open*, *write* and *format*. If this is the first trial, noWorkflow will associate all the collected provenance to the trial id 1. It is possible to get basic trial information (i.e. main script name, main script hash, start time, finish time) by running `now show 1` as we show in Figure 4.

Note that noWorkflow will not capture the definition provenance from *script2.py*, even though it is in the same directory of the main script and may be part of the workflow definition. This occurs because noWorkflow statically captures most of the definition provenance only to support the execution provenance collection, but it only captures the execution provenance from the main script, reducing the size and time overhead. Thus, it only captures the def-

```python
# script1.py
from script2 import y, z

def x(i):
    if i % 2:
        z(i)
        return y(i)
    return z(i)

if __name__ == '__main__':
    for i in range(3):
        x(i)
    z(i)
```

```python
# script2.py
from random import random
from time import sleep

def y(i):
    sleep(.01)
    with open('y.txt', 'a') as f:
        f.write('- {}\n'.format(i))

def z(i):
    sleep(.1)
    with open('z.txt', 'w') as f:
        f.write('- {}\n'.format(i))

__version__ = "1.0.2"
```

Figure 3. Script example

```
[now] trial information:
  Id: 1
  Inherited Id: None
  Script: script1.py
  Code hash:
   6d4bb7baa267060e4356860801c30060083403f1
  Start: 2015-03-23 23:28:32.253135
  Finish: 2015-03-23 23:28:33.629065
```

Figure 4. Showing trial 1 basic information

inition provenance from $script1.py$. However, it does capture the $script2.py$ content during deployment provenance collection.

3. Provenance Collection in IPython Notebook

We integrated noWorkflow's provenance collection and IPython Notebook by using IPython's concepts of *line magic* and *cell magic*. While our *line magic* collects provenance from external scripts, our *cell magic* collects provenance inside notebooks.

Line magic. The easiest way to capture provenance from external scripts is to simply execute noWorkflow as is. We propose a *line magic*, '$\%now_run$', to perform that. One could argue that this could be performed by a simple shell command. However, to analyze the externally collected provenance in the notebook, a scientist would have to know the generated trial id and load a trial object that provides an interface for analysis, as we show in Section 4. Our *line magic* not only executes noWorkflow externally, but also returns the trial object, which can be used for immediate analysis. This *line magic* supports all arguments that the default `now run` command supports (see [8] for details on those commands).

Cell magic. While the aforementioned *line magic* improves the usability for analyzing the notebook, it is tailored to execute external scripts outside the notebook. This would require the script to be previously created and saved into a file before running it in the notebook. To avoid this step, we propose a *cell magic*, '$\%\%now_run$', which runs the script defined in its body. When this *cell magic* is executed, it creates a temporary file with the cell content as file content. Then, it runs noWorkflow with this file as input. Considering that the file runs externally, it is not possible to use notebook variables directly in the cell. It is only possible to pass these variables as parameters to the script. The same way, it is not possible to use the result of the trial directly, but it is possible to load the output into a variable.

By default, noWorkflow uses the script name to identify trials and this name can be used in queries to look at a trial family (i.e., trials that are probably similar since they were generated from scripts with the same name, that belong to the same experiment or exploratory analyses). Since cells have no name, we added an optional argument, *name*, on the *magics* to indicate the trial family. With this argument, it is possible to indicate that a given trial from a specific cell belongs to a specific experiment.

IPython extension. We implemented an IPython extension, called *noworkflow* to register these and other *magics* related to noWorkflow in its initializer. The *line magic* '$\%now_ls_magic$' lists existing noWorkflow *magics*.

Figure 5 presents provenance collection using noWorkflow. The first cell loads the extension, sets the default graph width to 392px and the default graph height to 150px. The second cell uses a *line magic* to execute an external script with a custom script name (*tapp*) and returns the trial id (4). The third cell assigns a value to a variable. The fourth cell uses a *cell magic* to execute an internal script, with the same name, defines that the cell output will be stored on the variable *out_var*, passes the variable *size* as argument, and returns a trial object. Note that the result of the fourth cell is a trial object and it is represented as a graph as we explain on Section 4. Finally, the last cell just returns the value of *out_var*. Note that Python's print appends a '\n' by default.

4. Provenance Analysis in IPython Notebook

The first step in supporting provenance analysis on notebooks is to connect to the provenance database. With access to the database, it is possible to query the provenance and use it for analysis. We propose visualizations, querying methods and trial objects to perform analysis using notebooks.

A trial object represents a single trial. It can be instantiated by specifying only the trial id. A trial has information about the scripts that generated it, its start time, finish time, environment variables, imported modules, accessed files, and activations (function calls). When a user wants to perform common queries to get these trial information and process the results using Python, she can access properties and call methods from the trial object and it will connect to the database to retrieve data ready for immediate analysis. The trial object caches some results to avoid querying the database every time. In addition, the trial object has properties and methods to retrieve other derived information, such as the trial duration. The default visualization of a trial object is an activation graph that shows the sequence of calls and sub-calls. An example of activation graph is shown in *Out[4]* of Figure 5.

IPython supports many display methods for objects. Since we display graphs for trials, the most suitable methods are PNG, SVG, and HTML. While PNG and SVG have the advantage of supporting visualization not only on the Notebook, but also on QtConsole

```
In [1]: %load_ext noworkflow
        %now_set_default graph.width=392 graph.height=150

In [2]: trial = %now_run script1.py --name tapp
        trial.id
Out[2]: 4

In [3]: size = 5

In [4]: %%now_run --name tapp --out=out_var $size
        import sys
        l = range(int(sys.argv[1]))
        c = sum(l)
        print(c)
Out[4]:
```

Trial 5. Ctrl-click to toggle nodes /tmp/now_run_O53AQP/now_run sum range

```
In [5]: out_var
Out[5]: '10\n'
```

Figure 5. Provenance collection in notebook using noWorkflow

```
In [1]: import noworkflow.now.ipython as nip
        nip.init(path='/home/joao/projects/tapp15')
        nip.set_default('graph.width', 392)

In [2]: nip.Trial(1, graph_height=350, graph_mode=2)
Out[2]:
```

Trial 1. Ctrl-click to toggle no /home/joao/projects/tapp15b/script1 range z x y x z z z

Figure 6. Trial visualization in notebook

during the execution of IPython shell, HTML has the advantage of supporting JavaScript, allowing better interactions. For this reason, we chose HTML with JavaScript as the output display format for trials.

We adopted the D3 JavaScript library[4] to display the graphs. We chose D3 because it is fast enough to display thousands of nodes in a graph and customizable for applying different techniques. We implemented our visualizations in external JavaScript files with custom CSS. Thus, it is necessary to load D3 and load our JavaScript and CSS files to run our visualizations on the notebook. This is accomplished by an *init* function that has the purpose of loading both external JavaScript and CSS dependencies, and setting the project path on the persistence module. By default, the path is the current directory, but it is possible to specify other directories with a named argument.

This *init* function is the part of noWorkflow that eases its integration with IPython. Besides this function, noWorkflow also provides access to the trial objects and to special functions that sets default variables. The *noworkflow* extension initializer calls this function when it is loaded. Thus, it sets the project path to the current directory. It is possible to overwrite this configuration by manually calling the *init* function afterwards.

Figure 6 presents the use of a notebook to open the noWorkflow visualizations. In the first cell, we imported the module *ipython* and named it *nip*, then we called the function *init* to load all JavaScript and CSS dependencies and to set the project path to "*/home-/joao/projects/tapp15*". Finally, we set the default graph width of 392px for all graphs. In the second cell, we just instantiated a trial object referring to trial 1 with custom height (350px) and custom graph mode (2). This visualization presents activation as nodes and fill the nodes color according to their duration (using the traffic light scale). Edges on the graph represents calls (black arrows), sequences (blue arrows), and returns (dashed black arrows). In this case, it is possible to see that the main script called *range*, then *x* three times, then *z*. First and third *x* called *z*. Second *x* called *z*, then *y*. The graph mode specifies how to combine activations and summarize the graph. It is possible to visualize all activations (mode 1), combine activations with the same name and sub-structure (mode 2) or combine activations by namespace (mode 3).

As we mentioned before, visualization is not the only way to analyze provenance in noWorkflow. The trial objects have fields that can be explored. For example, the field $script_content$ returns the content of the main script, while the file *id* returns the trial id.

It is also possible to run Prolog and SQL queries. We propose two *cell magics* to allow queries: '$\%\%now_prolog$' and '$\%\%now_sql$'. Both *cell magics* execute queries and may receive a variable result as parameter. If they receive a variable result, the *magic* assigns the result to the variable as an iterator. If they do not receive it, the result is presented as output. The *cell magic* '$\%\%now_sql$' outputs a table where the first row is the header. The *cell magic* '$\%\%now_prolog$' outputs a list. Each entry in the list is a match. It is possible to interpolate the content of both *cell magics* with python code.

The cell magic '$\%\%now_prolog$' may also receive trial ids as parameters. The ids indicate that it should export provenance from specified trials as Prolog facts. This way, we avoid eagerly loading the whole database and export facts on demand. This *magic* also loads Prolog rules that are automatically generated by noWorkflow (see [8]).

Figure 7 presents these possibilities of analysis. The first cell (*In [3]*) assigns the result of the last cell (i.e. the trial object from Figure 6) to the variable *trial* and presents the code that generated trial 1. The second cell (*In [4]*) queries the duration of *z* activations using Prolog. Note that this cell loads trial 1 facts, interpolates the '*trial.id*' into the query content, and stores an iterator into the variable *result*. The third cell (*In [5]*) iterates through the result and prints the matches. Finally, the last cell (*In[6]*) performs a SQL query to get all the activations that accessed the file '*y.txt*' and outputs its result.

When a scientist collects provenance by running noWorkflow outside a notebook, she may want to perform the analysis on a

[4] http://d3js.org

```
In [3]: trial = _
        print(trial.script_content)

        from script2 import y, z

        def x(i):
            if i % 2:
                z(i)
                return y(i)
            return z(i)

        if __name__ == '__main__':
            for i in range(3):
                x(i)
            z(i)

In [4]: %%now_prolog --result result 1
        duration({trial.id}, z, X)

In [5]: for match in result:
            print(match['X'])

        0.101475000381
        0.10070514679
        0.100728988647
        0.100710868835

In [6]: %%now_sql
        SELECT A.id, A.name, A.trial_id
        FROM file_Access F
        JOIN function_activation A
        ON A.id == function_activation_id
        WHERE F.name == "y.txt"

Out[6]:
```

id	name	trial_id
7	y	1
17	y	2
30	y	4

Figure 7. Provenance analysis in notebook

notebook. To easy this task, we implemented an export command line option on noWorkflow to export notebook files related to trial objects. The export command receives the trial id and generates a notebook file with the code used for loading the trial.

A scientist can perform analysis on any data collected by noWorkflow through SQL queries, Prolog queries and calls to methods. Thus, it is possible to perform analysis on the definition provenance, by getting the defined functions, arguments, and globals in the script used for a trial. It is possible to perform analysis on deployment provenance by identifying the versions of imported modules; reading their source code; and checking the environment variables. Finally, it is possible to perform analysis on the execution provenance by getting the files content before and after opening them; extracting and visualizing activation graphs; and getting duration, parameters, globals, and return values from activations.

Scientists can also integrate different tools and queries, since the queries results can be obtained as Python objects and connected through Python code. For example, in an experiment in which an environment variable defines the name of a file, a scientist may run a SQL query to get the environment variable first, then run a Prolog query to get the file hash, use a method to get the file content and open it in an external tool for analysis. If it is a recurring process, the scientist can easily define a function on IPython to perform it. Figure 8 presents a function that performs these operations for a specific trial and returns attributes from the result extracted by an external tool.

```
In [3]: def extract_from_result(trial_id, attributes):
            trial = nip.Trial(trial_id)

            # SQL query
            sql = first(nip.persistence.query("""
                SELECT value
                FROM environment_attr
                WHERE name='EXP_NUMBER'
                AND trial_id={}""".format(trial.id)))
            name = 'exp{}.dat'.format(sql['value'])

            # Prolog query
            prolog = first(trial.trial_prolog.query("""
                access({0}, _, '{1}', _, _, X, _, _)
            """.format(trial.id, name)))
            hash = prolog['X']

            # Create new file with old content
            content = nip.persistence.get(hash)
            with open('.temp.dat', 'w') as f:
                f.write(content)

            # Call external program to extract result
            result = !./extractor .temp.dat $attributes

            # Transform result into a dict
            values = result[0].split()
            attrs = attributes.split()
            return {key:values[i]
                    for i, key in enumerate(attrs)}

In [4]: extract_from_result(1, "x result")
Out[4]: {'result': '3', 'x': '0'}

In [5]: extract_from_result(2, "result y")
Out[5]: {'result': '40', 'y': '4'}
```

Figure 8. Provenance analysis with a custom function

5. Related Work

Previous approaches have been proposed to integrate provenance into interactive notebook systems. Ducktape [14] generates IPython notebooks with visualizations for provenance captured from external experiments. While they focus on using the notebook only for interactive visualization, our approach allows scientists to run any kind of analysis on the notebook, including running Prolog and SQL queries, and using common object queries. We also allow scientists to fully use notebooks as their main experimentation environment, by providing *cell magics* for collecting provenance.

Lancet [12], on the other hand, allows scientists to perform exploratory research on IPython Notebook by capturing provenance and visualizing results. However, it does not capture provenance from pure Python code. It requires the definition of special launchers and the usage of the existing ones to capture provenance. This may result in a steeper learning curve than using our approach, as we capture provenance from any Python code.

6. Conclusion

In this paper, we present a new mechanism to collect and analyze provenance for IPython notebooks. The mechanism supports provenance analytics through different tools, including SQL queries, Prolog queries, object properties, and graph visualization. With our approach, it is possible to get provenance data from different queries, and combine them with custom Python code. We also propose *magics* to improve the usability during the collection and analysis of provenance inside the notebook. This allows scientists to perform all their exploratory work interactively in-

side notebooks: they need not to switch environments to capture or analyze the provenance information. noWorkflow is available as open source software at `https://github.com/gems-uff/noworkflow`.

IPython Notebook, QtConsole, and other language-agnostic parts of IPython will move to a new project called Jupyter in the near future. Jupyter will use IPython as its kernel for running Python code interactively. Since Jupyter will keep its concepts, we believe our approach will stay valid for capturing Python provenance under Jupyter notebooks.

As future work, we plan to implement new visualizations for provenance, such as variable dependency graphs and diff visualization for files, modules, and environment. We foresee the integration of noWorkflow with Pandas [7] to improve provenance analysis. We also plan to support provenance collection internally to IPython to ease variable interchanging. Finally, it would be interesting to extend noWorkflow to capture provenance from other languages supported by Jupyter.

Acknowledgments

Authors would like to thank CNPq, CAPES and FAPERJ (Brazil) for partially supporting this work. This work was supported in part by a Google Faculty Award, an IBM Faculty Award, the Sloan Foundation, the Moore-Sloan Data Science Environment at NYU, NSF awards CNS-1229185 and 1405927.

References

[1] E. Angelino, D. Yamins, and M. Seltzer. StarFlow: A script-centric data analysis environment. In *International Provenance and Annotation Workshop (IPAW)*, pages 236–250, Troy, NY, USA, 2010.

[2] C. Bochner, R. Gude, and A. Schreiber. A Python Library for Provenance Recording and Querying. In *International Provenance and Annotation Workshop (IPAW)*, pages 229–240, Salt-Lake City, UT, USA, 2008.

[3] F. S. Chirigati, D. Shasha, and J. Freire. Reprozip: Using provenance to support computational reproducibility. In *USENIX Workshop on the Theory and Practice of Provenance (TaPP)*, pages 977–980, 2013.

[4] A. Davison. Automated Capture of Experiment Context for Easier Reproducibility in Computational Research. *Computing in Science & Engineering*, 14(4):48–56, 2012.

[5] J. Freire, D. Koop, E. Santos, and C. Silva. Provenance for Computational Tasks: A Survey. *Computing in Science & Engineering*, 10(3): 11–21, May 2008.

[6] H. P. Langtangen. *Python scripting for computational science*, volume 3 of *Texts in Computational Science and Engineering*. Springer, 2006.

[7] W. McKinney. *Python for data analysis: Data wrangling with Pandas, NumPy, and IPython*. "O'Reilly Media, Inc.", 2012.

[8] L. Murta, V. Braganholo, F. Chirigati, D. Koop, and J. Freire. noWorkflow: Capturing and Analyzing Provenance of Scripts. In *International Provenance and Annotation Workshop (IPAW)*, pages 71–83, Cologne, Germany, 2014.

[9] F. Perez and B. E. Granger. IPython: a system for interactive scientific computing. *Computing in Science & Engineering*, 9(3):21–29, 2007.

[10] H. Shen. Interactive notebooks: Sharing the code. *Nature*, 515(7525): 151–152, 2014.

[11] M. Stamatogiannakis, P. T. Groth, H. J. Bos, and others. Looking Inside the Black-Box: Capturing Data Provenance using Dynamic Instrumentation. In *International Provenance and Annotation Workshop (IPAW)*, pages 155–167, Cologne, Germany, 2014.

[12] J.-L. Stevens, M. Elver, and J. A. Bednar. An automated and reproducible workflow for running and analysing neural simulations using Lancet and IPython Notebook. *Frontiers in Neuroinformatics*, 7:44, 2013.

[13] D. Tariq, M. Ali, and A. Gehani. Towards Automated Collection of Application-level Data Provenance. In *USENIX Workshop on the Theory and Practice of Provenance (TaPP)*, pages 1–5, Boston, MA, USA, 2012.

[14] A. Wibisono, P. Bloem, G. K. de Vries, P. T. Groth, A. Belloum, M. Bubak, and others. Generating scientific documentation for computational experiments using provenance. In *International Provenance and Annotation Workshop (IPAW)*, pages 168–179, Cologne, Germany, 2014.