IoLab: Reproducible, In-order Computational Notebooks in JupyterLab

Eleftheria Beres
eberes@cs.washington.edu
University of Washington
Seattle, Washington, USA

Carlyn Schmidgall
crschmid@uw.edu
University of Washington
Seattle, Washington, USA

Ryan Zambrotta
ryanzam@cs.washington.edu
University of Washington
Seattle, Washington, USA

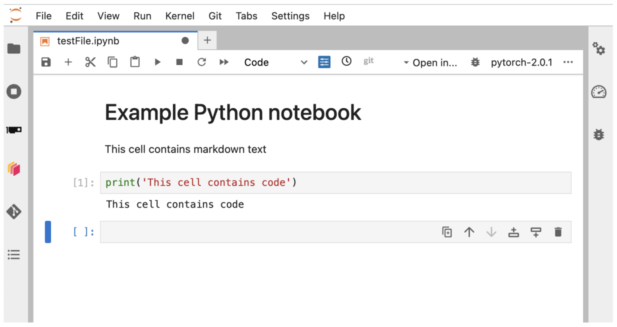
ABSTRACT

Computational notebooks such as Jupyter notebooks have become a de facto standard for computational science. However, despite being widely used for scientific workflows and artifacts, Jupyter notebooks are wildly irreproducible. A primary contributing factor to the irreproducibility of Jupyter notebooks is the ability within them to execute code at any time in any order. This leads to out-of-order execution bugs and program states that are inconsistent with the text of the notebook.

We present IoLab, a JupyterLab extension that prevents out-of-order execution bugs by enforcing a single linear order of execution. With IoLab, the notebook's state is always kept as if cells were executed from the top in order without any skipped cells. IoLab achieves this by saving and loading states as needed. We perform a qualitative user evaluation of IoLab with four scientists at the University of Washington.

1 INTRODUCTION

Computational notebooks have become among the most widely used programming interfaces for data science and computational science [7, 13]. A computational notebook is a collection of editable cells traditionally presented as a single, linear sequence from top to bottom such as in Figure 1. Each cell contains either code or Markdown and may be run on demand by the programmer. Critically, the programmer may execute and edit cells in any order and at any time. While this feature may aid programmers in exploratory programming tasks, it contributes to poor reproducibility of in computational notebooks. We focus on one of the most popular computational notebook implementations: Jupyter Notebooks[6, 9, 12].



Example Python notebook

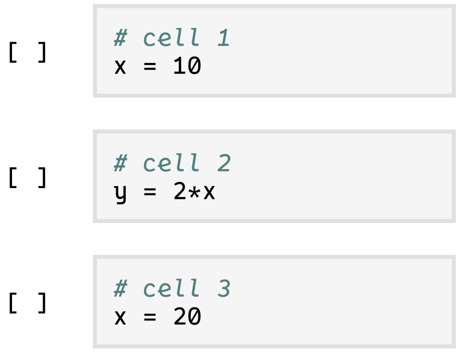
This cell contains markdown text

```
[1]: print('This cell contains code')
     This cell contains code
[ ]:
```

Figure 1: The user interface of a Jupyter Notebook.

Despite the intention of Jupyter Notebooks to be a reproducible platform for computational scientific workflows[9], a large-scale study of published Jupyter Notebooks finds that less than a quarter of Jupyter Notebooks can be re-executed without error and less than 5% produce the same results when re-executed[14]. The same study found that of studied Jupyter notebooks, 36% had evidence of out-of-order execution and 76% had evidence of skips in execution order. Both activities contribute to irreproducibility in Jupyter Notebooks by divorcing the state of a notebook at any given time from the program text presented in the notebook.

Consider a simple Jupyter Notebook with three cells shown in Figure 2. If this notebook was to be run with the cells in their order "on the page"—i.e., cell 1, then cell 2, then cell 3—the final values of the variables would be `x=20; y=20`.



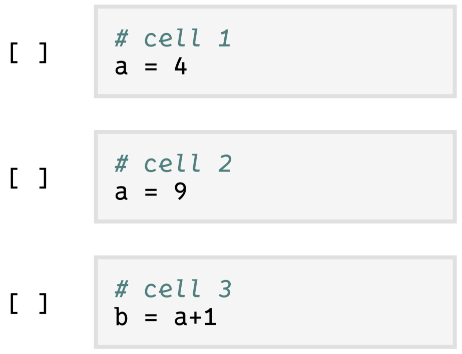
```
[ ]   # cell 1
      x = 10

[ ]   # cell 2
      y = 2*x

[ ]   # cell 3
      x = 20
```

Figure 2: A simple notebook with 3 cells.

CSE 503 2025, 19 March, 2025, Seattle, Washington
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

However, if cell 2 is re-executed at this point—once cell 3 has already been executed—then the values would be `x=20; y=40`. Additionally, if cell 1 is now re-executed, then the variable values become `x=10; y=40`. Finally, if cell 2 were to now be re-executed without cell 3 being executed, the variable values would be `x=10;y=20`. This one notebook of three cells can result in four different program states depending on the order in which cells have been executed, which is inherently problematic for reproducibility. Of these four reachable states in which the program may be found after execution, only one—the first—is consistent with the order of the text of the program. Thus, the notebook may fail to be reproducible unless the user executes the cells in the exact same order each time, which is not guaranteed.

Additionally, Jupyter Notebooks enable programmers to "skip" code when executing notebooks. In Figure 3, if the three cells are all executed in order, then the variable values would be `a=9; b=10`. However, if the programmer skips cell 2 during program execution—i.e., if cell 3 was executed directly after cell 1 and cell 2 was not executed at all—then the variables would take values `a=4; b=5`. Here again, only the first state is consistent with the text of the program.



```
[ ]   # cell 1
      a = 4

[ ]   # cell 2
      a = 9

[ ]   # cell 3
      b = a+1
```

Figure 3: A different simple notebook with 3 cells.

Out-of-order execution[1] and the challenges it poses to reproducibility and consistency are found across a range of notebook implementations including Jupyter, R Markdown notebooks[3] and Quarto Notebooks[5]. However, we limit our scope to solely focusing on Jupyter.

In this paper, we introduce IoLab[2], a tool built on top of JupyterLab that prevents out-of-order execution in Jupyter notebooks by enforcing textual top-to-bottom execution of cells. Our primary contributions are:

- The design and implementation of the IoLab system to prevent out-of-order execution in computational notebooks in JupyterLab.
- A user study with scientific researchers at UW providing insights into the usability of a linearly executing Jupyter notebook and into the changes to notebook workflows necessitated by a linearly executing notebook (forthcoming).

2 BACKGROUND

2.1 Jupyter notebook architecture

The Jupyter notebook architecture is composed of three component parts: a front-end notebook interface, a kernel back-end, and a messaging protocol between the two.

The notebook front-end is responsible for allowing programmers to edit the text of notebook cells and for sending execution requests to the kernel. The notebook keeps the text of the program cells and is reliant on the kernel for code execution, code completion, and other code interaction tasks.

The back-end to a Jupyter Notebook is a Jupyter kernel. The kernel is the program that executes code on command from the notebook front-end. Kernels can be written in any language and can execute code in any language; the only requirement is that the kernel implements the necessary infrastructure to receive and reply to messages from the front-end using the Jupyter messaging protocol.

2.2 REPL vs. In-order

Traditionally, the default Jupyter kernel is the IPython Kernel. The IPython Kernel is effectively a Python REPL (read-eval-print loop) that interactively executes Python commands in the order received. Most Jupyter kernels for other programming languages also follow this model. Thus, from the perspective of the IPython kernel—and most other Jupyter kernels—there is no program or "correct" order in which cells should be executed. The kernels are simply REPLs that execute code on command in the order received.

We define **in-order execution** to mean the order in which program text appears "on the page." Thus, in-order execution for the notebook presented in Figure 2 would be cell 1, then cell 2, then cell 3. This is equivalent to the order in which lines of code would be executed if they were to be copied directly from a notebook to a script in order from the top of the notebook to the end.

3 DESIGN

We seek to facilitate more reproducible computational scientific work in computational notebooks. As such, there are three primary design principles for our system:

(1) **The system must prevent out-of-order execution.** As we seek to eliminate out-of-order execution bugs, our system must ensure that it is not possible for a programmer to inadvertently or intentionally execute cells in a notebook out of their linear order.
(2) **The system must keep the program state always consistent with the text of the notebook.** Since irreproducibility can emerge due to both out-of-order code execution and to

[1]We include both executing notebook cells in a different order than they're presented linearly in a notebook and skipping cells in notebook execution under the umbrella term "out-of-order execution" since both contribute to the same challenges to irreproducibility.
[2]Named both for "In-Order JupyterLab" and after Io, one of the Galilean moons of Jupyter.

moving or editing cells after execution, we also must eliminate the possibility for the user to change the program text within a notebook by editing or reordering cells without updating the notebook state to reflect the textual change.

(3) **The system must be practical for use in scientific workflows.** The system must not require inordinate memory or computation time to reach the other two design principles.

Furthermore, in this work, we limit our scope to a "closed-world" model of computational notebooks. In other words, we do not consider interactions between notebook code and external sources such as files on the file system. Practically, this means that we do not track notebook file dependencies the notebook may have. Potential future work includes integrating our system with the wider file system to track changes to data source files scientists may rely on for their work.

4 IMPLEMENTATION

Our tool, IoLab[3], prevents out-of-order execution by enforcing a in-order, top-to-bottom execution order. IoLab consists of two pieces: a JupyterLab extension and a Jupyter kernel. IoLab is designed to work with JupyterLab version 4.3.5, the current stable JupyterLab version as of this writing.

IoLab seeks to be a lightweight change to the JupyterLab user interface. The only interface change is a new "run" button on cells to be used in place of the regular JupyterLab run buttons. Additionally, from the user's point of view, IoLab notebook functions similarly to a regular Jupyter Notebook. When a programmer writes and executes code cells in order from top to bottom (without skipping cells or re-running cells), IoLab behaves identically to a traditional Jupyter notebook.

However, when a previous cell—one "higher up" on the page—is re-executed, a traditional notebook would simply execute this cell, building off of the current program state. Instead, IoLab will "roll back" to the proper state, reverting the program state directly before the re-executed cell before executing it. Thus, the program state would be identical to if the notebook were restarted and run from the top up to and including the re-executed cell. The output of all following cells after the re-executed cell become "dead" and the cells need to be re-executed. The program state of IoLab will always be the same as if the notebook was always restarted and the cells were executed in order from the start.

The most naive implementation of a system that preserves this desired invariant—that the state always be equivalent to what it would be if the notebook were always restart and run from the top—would be a system that always deletes any existing program state and re-executes all cells up to and including the desired cell. By construction, this design preserves in-order execution order because it is equivalent to concatenating all notebook code cells and running them as a script. Unfortunately, for many scientific workflows which often rely on expensive computations such as loading in data or training models, this constant recomputation would prove to be costly and impractical. As such, a system relying on this mechanism would violate our design principles.

[3]https://github.com/ellifteria/iolab-kernel and https://github.com/ellifteria/iolab-extension

Fortunately, a property of in-order execution order is that code changes lexically later in the program cannot[4] impact the execution of earlier code. As a consequence, with this invariant, a notebook cell's state is unaffected by code changes to later cells; this is different from a traditional notebook. This means re-execution of a cell does not necessarily demand re-execution of the entire program. Re-execution may begin at the program state immediately preceding the earliest modified cell. Therefore, it is not necessary that all cells always be recomputed, it is possible to preserve the desired invariant while allowing only a subset of cells in a notebook to be re-executed at a given time.

4.1 IoLab JupyterLab extension

The IoLab JupyterLab extension functions as a listener attached to the JupyterLab front-end user interface. The extension keeps track of changes to the notebook and updates the notebook's output and kernel state accordingly. We achieve this by providing an alternate semantics for "running" notebook cells and by implementing a listener on the front-end that detects when changes to the notebook occur that alter the current state of the kernel.

As opposed to the traditional "run" semantics of a Jupyter notebook which simply send a cell's code to the kernel for execution, when a user runs a cell with the IoLab extension enabled, IoLab first determines what the state of the kernel should be prior to the execution of that cell. This means that if the cell at index n is run by the user, the kernel should be in the state that it would be in after the prior cell, cell $n - 1$, was executed. In the case where $n = 0$, the kernel should have an "empty" state; in practice, this means the state that it would be on starting up the IPython REPL before any code is executed. By induction, therefore when any cell n is executed, the kernel should be in the state it would be in if and only if all prior cells $0, \ldots, n - 1$ were executed in ascending order of index—i.e., the state the notebook would be in if all preceding cells were executed in order from the top after restarting the kernel. Thus, once cell n is executed, the state of the program is the state the program would be in if the kernel were restarted and all cells up to and including n were run in order from the top.

To achieve this, whenever a cell is executed, the kernel first determines how the kernel's current state compares to the desired state. It does so by keeping track of the last successfully executed cell in the notebook using the cell's ID since each cell in a Jupyter notebook has a unique UUID ID. If there is no most last executed cell or if the cell being executed is the first cell in the notebook, then the IoLab extension simply tells the kernel to reset its state and orders execution of all cells from the first cell in the notebook to the cell being executed in order. Otherwise, the IoLab extension obtains the current ordering of cells in the notebook and compares the indices of the cell being executed and the last executed cell. If the last executed cell directly precedes the cell being executed, then the extension can assume the kernel is in the correct state to execute the cell. Therefore, it tells the kernel to simply execute that cell without changing its current state beforehand. If instead the last executed cell is after the cell being executed or is the cell being executed, then the IoLab extension knows two things. First,

[4]In the case of loops, the code contents of a cell are restricted to being valid grammatical statements meaning the entire body of the loop must be contained within a single cell.

that the kernel state needs to be changed before executing the cell and second, that the kernel has in the past been in the correct state to execute the cell being executed. Therefore, the IoLab extension orders the kernel to revert its state to the desired state and then execute the cell. Lastly, if the last executed cell is before the cell being executed but does not directly precede it, the IoLab extension knows that the kernel state needs to be changed before executing the cell however the kernel has not necessarily previously been in the correct state. Thus, to achieve the desired state, the cells between the last executed cell and the cell being executed must be executed in order before the cell being executed can be executed. Therefore, it orders the execution of all cells between the last executed cell and the cell being executed without changing its state beforehand. Note that since this is checked on the execution of every cell, the invariant that for a cell to be executed, the kernel state must be as if all preceding cells have been executed in order from an empty kernel state is always preserved on cell execution.

However, modifications to the content or order of cells can also decouple the program state from the text of the notebook. Thus, the IoLab extension also listens for changes to the content and ordering of cells. It does this by listening to all changes to the notebook's active cell: the cell currently selected by the user. This fires whenever the user edits a cell and then leaves it, deletes a cell, inserts a cell, moves cells around, or leaves a cell without editing it. Therefore, the IoLab extension must determine which of these operations occurred whenever the hook fires. It first compares the current ordered list of cells to the previous list of cells. There are five possibilities:

(1) If the current list is longer than the previous one, a cell has been added.
(2) If the current list is shorter than the previous one, a cell has been deleted.
(3) If the current list is the same length as the previous one but the ordering of cell IDs within it is different, then a cell has been moved around.
(4) If the current list is the same length as the previous one and the ordering of cell IDs within it is the same, but the text within a cell has changed, then a cell's text has been edited.
(5) Finally, if the current list is the same length as the previous one and the ordering of cell IDS within it is the same and the text within all cells is the same, then there has been no change to the ordering or content of cells—the user simply left a cell without changing it.

In the case of the last possibility, no change to the kernel state is needed. Likewise, if a cell has been added, then no change to the kernel state is needed. This is because cells on addition are empty; no code has been added to the notebook at this point. In all other cases, however, the state of the kernel may need to be updated to reflect the new text of the program.

In items 2–4, the IoLab extension determines where the change to the cells occurred. If the first change to the cell list occurred after the last executed cell, then the change to the cells cannot have any effect on the state of the program run up to this point. Otherwise, the extension instructs the kernel to revert the program state to that directly before the first changed cell. If the change to the cells was an edit to the text of the cell, then that cell is re-executed to keep the notebook state consistent with the text of the notebook. If not, then no cell is to be executed; only the state of the kernel needs to change.

In order for the IoLab extension to instruct the kernel that it needs to change its state before executing a cell, the IoLab extension modifies the metadata of the cell being executed to include an instruction to change the kernel's state and the state to which the kernel should be set. These instructions are parsed by the kernel. To change the state of the kernel without executing code, the IoLab extension sends an execute request to the kernel with the modified metadata and `pass` as the code.

Whenever the IoLab extension sends any execution command to the kernel, it receives a promise that resolves whenever the kernel finishes executing the code. The promise resolves to `true` if the code executed without error and `false` if an error occurred in execution. If the code executed without error, then the IoLab extension updates the last executed cell to be the cell that was executed successfully.

Since the IoLab extension works to keep the program state as if the executed cells in the notebook were executed in order after every individual change to the notebook, the IoLab extension always keeps the program state correct.

4.2 IoLab Jupyter kernel

The second part of IoLab is the IoLab kernel. The IoLab kernel is build on top of the IPykernel[12], a Jupyter kernel implementing the IPython interactive Python REPL. It has one key difference to the IPykernel: it must change its state to according to instructions provided by the IoLab extension. It does so by parsing the metadata of executed cells to determine if a state change is needed. If it is, the kernel loads in the previously-saved state requested by the extension. Then, the kernel executes the cell code and finally the new state is saved.

State loading and saving is accomplished with the `dill`[11] Python library which includes functionality for dumping interpreter states to and loading interpreter states from files stored on disk. The kernel modifies the cell's code before sending it to the IPykernel interpreter. First, a prologue is added that imports the `dill` library. If state is to be loaded, the prologue also includes the `dill` call that loads in the correct state file from a temporary directory. This prologue is appended to start the cell code. An epilogue to the cell code is appended to the end of the code including the `dill` call to save the state file to a temporary directory, as shown in fig. 4. The cell code with the attached prologue and epilogue is then sent to the IPykernel interpreter.

```
# Prolouge
%reset -f
import dill
dill.load_module("<prev-state-id>.dill")
# code cell body
...
# Epilouge
dill.dump_module("<new-state-id>.dill")
```

Figure 4: Wrapper user code with prologue and epilogue. The epilogue resets the Python Interpreter and loads the previously cached program state. The prologue caches the interpreter state including the results from executing the code cell's body.

4.3 IoLab notebook file I/O dependencies

As mentioned above, we do not track notebook file dependencies in IoLab. This also means that if a notebook were to write to a file and then read from the file or write to a file then overwrite that file, we would not track that change nor would IoLab store a copy of the original or changed version of the file. While this could lead to cases where users overwrite data in files later on in notebooks, in practice, this is not common. An analysis of notebook file I/O dependencies found that no in studied notebooks overwrote was data read from a file then later overwritten and in only in 27.5% of studied notebooks was data written to a file and then later read[16]. Importantly, none of these file reads and writes are reads or writes that could potentially be broken by "rolling back" program state. Thus, file I/O in practice is not a major challenge for an IoLab-like system.

5 EVALUATION

We conducted a pilot user study to understand how an in-order computational notebook impacts scientists' work. This pilot user study is a preliminary step towards a full formal user study, which is forthcoming. The primary research question we sought to address was:

- **RQ 1**: How does IoLab change user workflows in data tasks?

Additionally, since we are currently developing IoLab, we sought to answer the following questions about the current design of IoLab.

- **RQ 2**: How does IoLab affect scientists' user experience?
- **RQ 3**: How can IoLab be improved to better serve users?

For our Institutional Review Board approved study, we recruited $n = 4$ participants from scientific fields at the University of Washington. All participants were graduate students from the fields of oceanography, psychology, and computer science, and had 8 ± 1.8 years coding experience, including 5.75 ± 2.06 years of Python experience. All participants have used Jupyter Notebooks in Python previously, with most participants actively using Notebooks as a tool in their research.

Participants were then asked to complete a series of coding tasks designed to replicate the experience of developing a physical model and exploring its consequences. All users used IoLab to complete the task. We asked users to explore a common model in population dynamics:

$$\frac{dx}{d\tau} = rx\left(1 - \frac{x}{k}\right) - \frac{x^2}{1 + x^2}$$

where x is the population, τ is time, r is the growth rate, k is the carrying capacity. The task incrementally added complexity by initially excluding terms (such as the predation term $x^2/(1 + x^2)$). This is intended to bias users into revising and rerunning older code cells, creating chances for execution order bugs. Other tasks involved plotting solutions to this model with varying initial conditions $x(0) = x_o$ and parameter values $r \in [0.1, 0.99]$, as well as performing a classic analysis of the "fixed-points" of the model (i.e. the long-term stable populations). We included this analysis because it requires incremental revisions to the data-structures used, again potentially cause execution-order bugs. Although the tasks are designed to cause execution order bugs, all subjects were given IoLab so no execution order bugs should actually occur. During this process, users were asked to verbally share their thoughts and experiences. We took notes on these responses and include them in our analysis. Once users completed the coding portion, they were asked to complete a written questionnaire recording their reflections on the experience.

5.1 Results

We include participants' verbal and written responses to the posed research questions. Verbal responses were collected during the coding task. Participants volunteered these responses without explicit prompting. Participants were asked a version of each research question with different wording but similar semantic content. We include the literal questions below.

*5.1.1 Responses to **RQ1**.* For the written portion we asked participants *"How did the IoLab kernel affect how you would normally approach the challenge problems?"*. Their written responses were:

P1 "Usually, to avoid out of order execution issues, I'll just restart the kernel and run all the above cells. If I have cells that are more computational, I'll just rerun all of the cells underneath the computational cells and have to keep track of which cells I've re-executed and when I executed them. With the Iolab kernel, I just didn't even think about it."

P2 "I didn't realize what the new IoLab kernel's functionality was until [an author] told me. From my previous experience working with Jupyter notebooks, I formed a habit of copy and pasting nearly all code relevant to a cell, such as constants, to avoid dependencies on previous cells."

P3 "[IoLab] enforced cleaner coding. I really had to think through some of what I wanted to try instead of just trying them because I know that some cells are re-executed. I like moving all around the notebook and using the interact-ability of Jupyter notebooks right now so this felt a little bit more rigid. I'm not good at clean coding so this is a good learning opportunity."

P4 "I don't trust notebooks, so I usually program defensively when using them. I did like that the IoLab kernel seemed to have more safety mechanisms built-in, like invalidating cells that need to be re-run"

Participant verbal comments with respect to **RQ1** were:

P1 "Usually, to avoid out of order execution issues, I'll just restart the kernel and run all the above cells… [or] keep track of which cells I've re-executed and when… With [Iolab], I just didn't even think about it."

P4 "I've develop a style to avoids the pitfalls of [traditional] notebooks [including copying and pasting code and a Rust-like ownership model for parameters]... I still do it out of habit."

IoLab changed the workflow of participants by preventing the possibility of out-of-order execution errors. Each participant stated that they various utilized workaround strategies to avoid out-of-order execution errors, which are baked-in to their Jupyter notebook workflows. Although some participants (e.g. P3) expressed that IoLab did feel more rigid, overall users appreciated that IoLab would allow them to simplify their workflow and reduce the need for these workaround strategies to avoid errors. This results in IoLab improving user's trust in the results of their Jupyter Notebook code.

5.1.2 Responses to ***RQ2****.* For the written portion we asked participants *"Overall, how did the IoLab kernel affect your experience in Jupyter Notebooks? Feel free to address your ability to explore different ideas/calculations, work efficiently, avoid bugs, and understand the code of the notebook."*. Their written responses were:

P1 "It seemed like it gave me a more normal coding experience and had behavior I expected more compared to traditional notebook engines."

P2 "I liked it a lot! I'd definitely try it out in my workflow to see how quickly I could adapt to it - I am used to just mentally keeping track of what cells I've ran and I don't tend to form new habits super well, but I'd definitely install it and try it out!"

P4 "I think it's a cool idea and addresses my biggest gripe with Jupyter notebooks, which is how running cells out of order results in an implicit DAG of data dependencies that the user has to track. However, it honestly didn't affect my experience much because I normally work around the dependency issue by repeating as much context as possible, and the study tasks did not require repeated modifications to something like a helper function. But I think this is an important part of the experience to address."

Verbal comments with respect to **RQ2**:

P1 "I liked it a lot! I'd definitely try it out in my workflow to see how quickly I could adapt to it!"

P2 "It gave me a more normal coding experience and had behavior I expected more compared to traditional notebook engines"

P4 "Over time I've tried to stop using notebooks for my work because their [reproducibility] is really bad."

Overall, IoLab was received positively by study participants, who felt that it enhanced their user experience in Jupyter notebooks. Users appreciated that IoLab made extra steps to avoid out-of-order execution errors unnecessary, and were excited to integrate it into their workflow. However, participants also acknowledged that adapting to new habits might take time, especially since strategies to avoid out-of-order execution errors are so deeply baked-in to their usage of Jupyter Notebooks. Participants expressed enthusiasm to use IoLab in both teaching and research settings, where ensuring standardized execution order is paramount.

5.1.3 Responses to ***RQ3****.* For the written portion we asked particiants *"What features did you wish were present in the notebook to help solve the problem?"*. Their written responses were:

P1 "I didn't like that the cell output from cells below was removed when you execute a cell above it, so I couldn't compare the value of a variable before and after I made a change to it."

P2 "I wish there was more autocomplete or code completion suggestions. I can never remember if matplotlib expects "set_xlabel" or "xlabel" or what order things should be passed to numpy functions."

P3 "I wish I could see which cell it was executing instead of just waiting."

P4 "I'd like more visual feedback as to which cells are invalid and need to be re-run, that sort of thing."

Users suggested several ideas for desired features in IoLab. Most user suggestions concerned providing more visual feedback on the program state of cells, including clearer indications of which cells require re-execution and which cells are currently running. Some users also wished that IoLab retained previous cell outputs when executing earlier cells, because this allows for the comparison of output under different conditions. Users also requested enhanced autocomplete and code suggestions for function names and arguments.

5.1.4 Visual aids for invalidated cells. In general, users found visual indication that a cell needed to be rerun to be useful. In IoLab this is indicated by removing it's run number (the brackets in fig. 2) and clearing a cell's output. However, the nature of work in a notebook is exploratory so many code changes are made to examine the difference in results. For this reason, participants expressed conflicting opinions about the clearing of output cells. P1 preferred that outputs persist longer so that they might examine the difference between cells whereas P3 preferred the clear indication that a cell had not been run. A potential line of future work could be an extension to the notebook where users can hypothesis a code change and run it in a "fork" of the original program state and then "commit" that code change after inspecting it's output would resolve both users concerns.

5.1.5 Caching. A major limitation in the designed tasks is that all tasks were both relatively quick to recompute as well as used relatively little memory so the effects of caching were not apparent. For example, the largest computation asked of participants could run in 15 seconds on an M1 MacBook, but could also be easily optimized to take < 1s on the same hardware. Rerunning the entire notebook could take 5-20 seconds but was never required by the tasks as there were no "notebook-wide" dependencies. Most code dependencies were designed to be within 4 cells but the actual number varies on user coding style. The participants that tended to copy-and-paste cells would have no code dependencies. Follow-up experiments on IoLab could include tasks that involve making large, in-place updates to arrays which would stress the caching in IoLab.

6 RELATED WORK

6.1 Reactive execution

Reactive computational notebooks provide an alternative approach to the traditional REPL-based notebooks. Unlike traditional Jupyter Notebook kernels or REPLs, reactive kernels such as PlutoJL[15], marimo[4], IPyFlow [1], and Observable[2] track cells' dependencies and reactively update all dependencies if a cell is updated—similar to the execution model of spreadsheets such as Excel and Google Sheets. This model of notebook prevents out-of-order execution by maintaining a single, consistent state across all cells of a notebook at all time. However, reactive notebooks lack any concept of mutable state and sequential order of execution. Furthermore, the order of cells on the page in a reactive notebook does not reflect the order in which cells are executed.

6.2 Gathering execution trace

Nbgather is a tool developed to help manage messes in Jupyter Notebooks[8]. Nbgather keeps a log of cells as they are executed, tracking the execution history of a notebook. Then, it generates minimal program slices from the execution log to present to programmers so they can clean notebooks and discover the derivation of their results. Nbgather presents programmers with in-order, minimal slices of notebooks; however, it does not preserve the on-the-page execution order for cells. The minimal slice may have cells in a different order than they appeared in the original notebook since slices originate from the kernel's execution log, not the notebook's program text.

6.3 Forking and backtracking

Fork It! is a Jupyter Notebook tool that allows users to manually backtrack and fork notebook execution at a chosen cell to evaluate different possibilities[17]. With Fork It!, programmers are able to fork the notebook at a given point, reverting the state of the notebook back to the state at that point and allowing users to take an alternate approach. Fork It! implements manual backtracking by loading state from disk at manually set fork points in the notebook.

6.4 Lineage tracking

NBSafety is a tool that provides programmers with visual feedback when Jupyter Notebook cells may be "stale" due to the effects of executing other notebook cells[10]. NBSafety traces the lineages of notebook cells using static and dynamic analysis techniques to identify when cells may need to be re-executed due to updates elsewhere in a notebook. NBSafety does so while preserving existing notebook semantics, thereby notifying Jupyter Notebook users when cells may present safety errors while leavings programmers to determine when to re-execute cells.

7 CONCLUSION

We presented IoLab, a JupyterLab extension and Jupyter kernel that eliminates out-of-order execution in computer notebooks in JupyterLab by enforcing a linear order of execution. We described the design principles behind IoLab and detailed the technical implementation of the system. In a pilot qualitative user study with scientists at the University of Washington, we investigated how scientists change their workflows when using an in-order computational notebook.

REFERENCES

[1] [n. d.]. *ipyflow/ipyflow.* https://github.com/ipyflow/ipyflow original-date: 2020-01-14T18:02:25Z.

[2] [n. d.]. *Observable: Build expressive charts and dashboards with code.* https://observablehq.com/

[3] 2024. *rmarkdown: Dynamic Documents for R.* https://github.com/rstudio/rmarkdown R package version 2.29.1.

[4] Akshay Agrawal and Myles Scolnick. [n. d.]. *marimo - an open-source reactive notebook for Python.* https://github.com/marimo-team/marimo original-date: 2023-08-14T18:56:20Z.

[5] J.J. Allaire, Charles Teague, Carlos Scheidegger, Yihui Xie, Christophe Dervieux, and Gordon Woodhull. 2024. *Quarto.* https://doi.org/10.5281/zenodo.5960048

[6] Brian E. Granger and Fernando Pérez. 2021. Jupyter: Thinking and Storytelling With Code and Data. *Computing in Science Engineering* 23, 2 (2021), 7–14. https://doi.org/10.1109/MCSE.2021.3059263

[7] Jesse Harden, April Yi Wang, Rebecca Faust, Katherine E. Isaacs, Nurit Kirshenbaum, John Wenskovitch, Jian Zhao, and Chris North. [n. d.]. Human-Notebook Interactions: The CHI of Computational Notebooks. In *Extended Abstracts of the 2024 CHI Conference on Human Factors in Computing Systems* (New York, NY, USA, 2024-05-11) *(CHI EA '24)*. Association for Computing Machinery, 1–6. https://doi.org/10.1145/3613905.3636318

[8] Andrew Head, Fred Hohman, Titus Barik, Steven M. Drucker, and Robert DeLine. [n. d.]. Managing Messes in Computational Notebooks. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems* (New York, NY, USA, 2019-05-02) *(CHI '19)*. Association for Computing Machinery, 1–12. https://doi.org/10.1145/3290605.3300500

[9] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, and Carol Willing. 2016. Jupyter Notebooks – a publishing format for reproducible computational workflows. In *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, F. Loizides and B. Schmidt (Eds.). IOS Press, 87 – 90.

[10] Stephen Macke, Hongpu Gong, Doris Jung-Lin Lee, Andrew Head, Doris Xin, and Aditya Parameswaran. [n. d.]. Fine-grained lineage for safer notebook interactions. 14, 6 ([n. d.]), 1093–1101. https://doi.org/10.14778/3447689.3447712

[11] Michael M. McKerns, Leif Strand, Tim Sullivan, Alta Fang, and Michael A. G. Aivazis. 2012. Building a Framework for Predictive Science. arXiv:1202.1056 [cs.MS] https://arxiv.org/abs/1202.1056

[12] Fernando Pérez and Brian E. Granger. 2007. IPython: a System for Interactive Scientific Computing. *Computing in Science and Engineering* 9, 3 (May 2007), 21–29. https://doi.org/10.1109/MCSE.2007.53

[13] Jeffrey M. Perkel. [n. d.]. Why Jupyter is data scientists' computational notebook of choice. 563, 7729 ([n. d.]), 145–146. https://doi.org/10.1038/d41586-018-07196-1 Bandiera_abtest: a Cg_type: Toolbox Publisher: Nature Publishing Group Subject_term: Computational biology and bioinformatics, Computer science.

[14] João Felipe Pimentel, Leonardo Murta, Vanessa Braganholo, and Juliana Freire. 2019. A Large-Scale Study About Quality and Reproducibility of Jupyter Notebooks. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. 507–517. https://doi.org/10.1109/MSR.2019.00077

[15] Fons van der Plas, Michiel Dral, Paul Berg, , Rik Huijzer, Mikołaj Bocheński, Alberto Mengali, Connor Burns, Hirumal Priyashan, Benjamin Lungwitz, Jerry Ling, Eric Zhang, Felipe S. S. Schneider, Ian Weaver, Xiu-zhe (Roger) Luo, Shuhei Kadowaki, Gabriel Wu, Timothy, Luis Müller, Zachary Moon, Supanat, Sergio A. Vargas, Rok Novosel, Jelmar Gerritsen, Vlad Flore, Jeremiah, Ciarán O'Mara, and Michael Hatherly. [n. d.]. *fonsp/Pluto.jl: v0.20.4.* https://doi.org/10.5281/zenodo.14537201

[16] Shigeyuki Sato and Tomoki Nakamaru. 2024. Multiverse Notebook: Shifting Data Scientists to Time Travelers. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 121 (April 2024), 30 pages. https://doi.org/10.1145/3649838

[17] Nathaniel Weinman, Steven M. Drucker, Titus Barik, and Robert DeLine. [n. d.]. Fork It: Supporting Stateful Alternatives in Computational Notebooks. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems* (New York, NY, USA, 2021-05-07) *(CHI '21)*. Association for Computing Machinery, 1–12. https://doi.org/10.1145/3411764.3445527