How Scientists Use Jupyter Notebooks: Goals, Quality Attributes, and Opportunities

Ruanqianqian (Lisa) Huang*, Savitha Ravi*, Michael He, Boyu Tian, Sorin Lerner, Michael Coblenz
University of California San Diego
La Jolla, CA, United States
{r6huang, s2ravi, mih024, btian, lerner, mcoblenz}@ucsd.edu

***Abstract*—Computational notebooks are intended to prioritize the needs of scientists, but little is known about how scientists interact with notebooks, what requirements drive scientists' software development processes, or what tactics scientists use to meet their requirements. We conducted an observational study of 20 scientists using Jupyter notebooks for their day-to-day tasks, finding that scientists prioritize different quality attributes depending on their goals. A qualitative analysis of their usage shows (1) a collection of goals scientists pursue with Jupyter notebooks, (2) a set of quality attributes that scientists value when they write software, and (3) tactics that scientists leverage to promote quality. In addition, we identify ways scientists incorporated AI tools into their notebook work. From our observations, we derive design recommendations for improving computational notebooks and future programming systems for scientists. Key opportunities pertain to helping scientists create and manage state, dependencies, and abstractions in their software, enabling more effective reuse of clearly-defined components.**

***Index Terms*—scientific computing, computational notebooks, end-user software engineering**

I. Introduction

Unlike traditional development environments, computational notebooks interleave program text with program output in a linear flow of content. Notebooks divide programs into *cells*, enabling users to execute cells and see output in any order they choose. Computational notebooks, such as Jupyter [1], have enabled thousands of users to create millions of notebooks [2,3] to explore and communicate ideas in a form interleaving code, output, prose, and sometimes multimedia. These tools facilitate *end-user software engineering* [4,5]: creation of critical software systems by people whose main focus is completing tasks, not authoring software.

Despite the broad adoption of computational notebooks, they only provide a thin layer over existing runtime environments. Code in cells can side-effect the environment, impeding reproducibility when users execute cells out of order. Cells provide no abstractions; they are not invokable, for example, and their code shares a scope with surrounding code. Version control is rarely used [6,7], even by notebook users who use version control in their other work. Previous research [8] and commercial implementation [9] exploring alternative designs have not yet produced popular competitors.

Why do computational notebooks still serve the needs of their users better than existing alternatives? We hypothesize that this is not merely a problem of challenges transitioning from popular, well-documented tools; instead, we do not yet understand enough about *how* computational notebooks meet their users' needs. To lay a foundation for a new generation of tools, we seek to understand notebook users' *non-functional requirements—properties that notebook users value* and *how they promote the properties they value*—and *design opportunities* that could lead to more effective notebook systems.

This paper focuses on *scientific users* of computational notebooks, *i.e.*, those who write scientific programming code in notebooks. Prior work suggests that scientists struggle with keeping effective work in notebooks and transitioning from notebooks to other general-purpose programming tools; nonetheless, notebooks are widely popular among scientists [7]. To understand how and why computational notebooks benefit scientists—and identify opportunities for improvement—we focus on three research questions:

RQ1 What goals do scientists pursue in notebooks?
RQ2 What non-functional requirements do scientific users of computational notebooks prioritize?
RQ3 What tactics do scientists use to create notebooks that meet their quality requirements while also facilitating their scientific discovery process?

Past investigations have studied users of Jupyter notebooks, focusing broadly on general data workers [6,10,11]. However, like Ko et al. in their study of Java programming [12], we wanted to understand the interaction-level techniques that scientists use to create software that meets their requirements. Therefore, we conducted an observational study of how individual scientists use Jupyter notebooks to accomplish realistic day-to-day tasks. In our IRB-approved study, we recruited 20 participants to let us watch them do their work. To understand how software-related training affects use, we recruited both computer scientists and non-computer scientists.

We first observed how each participant worked with Jupyter notebooks using their own setup and workflow for up to 45 minutes. Then, because any limited observation period could have resulted in unfinished work, we asked them to wrap up the work for future continuation. Finally, we interviewed them about their experience with Jupyter notebooks during the study as well as overall perceptions.

Leveraging techniques from constructivist grounded theory [13], we noted memos of each observation and coded every notebook-related action as well as every notebook-

*Co-first authors.

related comment made by participants. In Sec. III, we present eight quality attributes that our participants valued: clarity, explorability, reusability, reproducibility, correctness, performance, debuggability, and collaboration. We describe tactics, such as implementing each task in one cell, that participants used to promote quality. Additionally, we show how participants incorporated AI tools into notebook work in scientific settings. Finally, we discuss design opportunities that are revealed by this framing of computational notebook use.

Cell structure and out-of-order execution distinguish notebooks from other programming environments and promote explorability, particularly for code that can be slow to run. Unfortunately, they simultaneously inhibit aspects of correctness and reproducibility, which scientists also value. However, future notebook systems could empower users to choose their tradeoffs more precisely by controlling scope, dependencies, and in what cases cells are evaluated automatically.

While our work may not render a complete description of how people use Jupyter notebooks in every possible way, our findings are contextualized in realistic tasks and can inspire future improvements to Jupyter notebooks and similar end-user programming environments.

In summary, this paper's key contributions are:

- Eight quality attributes that scientists value when creating and maintaining notebooks from an an observational study of 20 participants performing their own tasks;
- Tactics that scientists use to promote software quality in Jupyter notebooks;
- Opportunities for improvement in notebook tools that could promote quality in software written by scientists.

II. Methods

A. Participants

We recruited participants from various disciplines via institutional mailing lists, messaging platforms, and snowball sampling. Since we aimed to study users with different domain expertise, programming experience, and software engineering experience, we did not impose screening constraints other than requiring (1) prior experience with Jupyter notebooks and (2) the ability to demonstrate a realistic task to be done in Jupyter notebooks during the study. We explicitly hoped to recruit people with varying experience in software engineering, so we recruited half of the participants from the computer science discipline and the other half from other scientific disciplines. We stopped recruiting participants after we had achieved saturation, following standard practice [13]: After each study, we compared field notes to existing observations to construct theoretical categories and their properties; the last four participants did not lead to new categories or properties.

B. Procedure

Each study session included two parts: a 60-minute observation and a 30-minute interview. We conducted the studies in person and over Zoom, recording participants' screens for analysis. Two authors ran the studies and took turns to lead. Participants received a $23 gift card after the study.

During the observational portion, participants worked on tasks of their choosing (more in Sec. II-D) using Jupyter notebooks. We prepared a backup task in case participants did not have a task ready, but we never used it. We asked participants to explain their tasks as they worked, in the style of think-aloud. However, past observations of programmers have shown that additional depth is needed [14], so while working on the task, the experimenter asked questions as needed about the subject's workflow when using notebooks and various aspects of the task. After about 45 minutes of observation, we asked the participant to start wrapping up their work to continue later. Most tasks can take many work sessions to complete, and sessions can be hours long. To observe how users initiated the cleanup process in the limited time span of a study, we asked participants to spend up to 15 minutes wrapping up their work so they could easily pick up from where they left off; the 15-minute threshold was determined via pilot studies, and no study participants needed all 15 minutes. In the cases where we observed no wrap-up activities, we asked about participants' usual practices.

We conducted a semi-structured interview afterward. Our questions pertained to five topics: notebook reproducibility, notebook understandability, the ability of the participant or collaborators to continue working on the notebook, the expected longevity of the current notebook, and changes in workflow when using computational notebooks compared to scripts.

C. Protocol Development Process

We started the study with one very open-ended question: *How do people use Jupyter notebooks for their day-to-day tasks?* Prior work on notebook use and pain points [11,15] motivated our initial interview topics. We used a qualitative analysis method with key techniques from constructivist grounded theory [13], including open coding, theoretical coding, and memo writing. Following the *theoretical sampling* approach, we recruited participants in areas in which we lacked data, and we refined the questions we asked participants per Charmaz's guidelines [13] based on emerging patterns to enable a deeper understanding of topics for which we had not yet gathered sufficient data; while this led to new questions asked in later studies, our IRB advised us that new questions within the subject matter of the study did not need further review.

After ten sessions, we analyzed the memos that we had written and found that participants had varying attitudes towards using Jupyter notebooks: some used ad hoc organizational techniques, caring little about readability and/or reproducibility; others attended to organizing the content inside a notebook, ensuring attributes such as modularity, readability, and reproducibility. For example, P1 mentioned *"notebooks have no longevity;"* do all notebook users hold the same low expectations for the longevity of notebooks? Expectations about longevity could have implications on how users prioritize readability and reproducibility. With these questions in mind, starting the 11th session, we added additional questions to our protocol asking participants about their expectations for the notebook's longevity. We also observed that most par-

ticipants conducted cleanup tasks such as adding Markdown headings, leaving notes on to-dos both inside and outside the notebook, and deleting empty cells in their last 15 minutes. For participants who performed little notebook cleaning work at the end of the study, we asked why they chose not to do the cleaning work. We also asked about any cleaning work they would do if they were to share the notebook with others.

After the 15th interview, interested in what users valued in notebook work, we added questions about what they would teach to a novice notebook user.

D. Task

Each participant worked on their own ongoing tasks during the study. Prior to each study, we asked participants to bring their own tasks that could be completed within the time-frame of the study. Although the tasks required domain knowledge, each participant chose tasks in their own field of expertise. As a result, all of the challenges we observed were computational rather than domain-specific. Most participants worked on a single task during the observational session, but three subjects (P8, P15, P20) completed their first task and started a second one. We coded each of the task descriptions to categorize them into seven different task types based on our observations: analysis, visualization, refactoring, reproduction, development, data cleaning, and notetaking. Reproduction tasks involved taking existing data and an expected output and recreating that output from the data in a new notebook. We saw 23 tasks across 20 participants. All except P20 did the tasks in one notebook. To derive the approximate size of work in each notebook, we counted the scrolling activities that occurred during the study to approximate task size (defined as "Scroll Size" in Table I) using the low-level action codes, the derivation of which we describe in Sec. II-E. Table I shows the complete task descriptions, their respective task types, and the scroll sizes of their corresponding notebooks.

E. Data Analysis

We open-coded the study recordings, which resulted in two subsets of *low-level codes*: (1) those pertaining to actions conducted by the participant, and comments and preferences they indicated during the observational portion, and (2) those relating to the interview responses. The first two authors, who also ran the studies, coded the observational components of each study, and the third and fourth authors coded the interviews after reviewing the coded observations and field notes. While coding the observational components, the first two authors also coded *meta-observations*, *i.e.*, high-level observations they made about each participant's behavior implied by their actions or quotes. The four author-coders met weekly to review the low-level codes until achieving agreement.

Finally, to further seek patterns among the low-level codes, the first two authors conducted a second round of coding upon codes of participant comments, preferences, and meta-observations from the observational portion, and codes from the interview portion. This round of coding is top-down as the coders derive *high-level codes* implied by the low-level codes under three categories: quality attributes of notebook content, native notebook attributes related to the quality attributes, and user tactics. We then created memos to relate the user tactics to the quality attributes and the native notebook attributes, following the grounded theory practice. We categorized the tactics as either a direct use of notebook attributes or a workaround for notebook limitations. We report the identified quality attributes, and their associated notebook attributes and user tactics in Sec. III-C. We also include the full list of codes and a replication package in the paper supplement [16].

F. Limitations

While our starting question, *how do people use Jupyter notebooks?* is broad, our participants primarily work in academia either as graduate students or as scientists. As a result, 16 of the 23 tasks were part of ongoing research projects, limiting generalizability to non-research settings. Also, the projects we saw were of limited size; different techniques could be used by those who work on very large codebases. Participants brought in their own tasks, which might result in varying task difficulties albeit improving external validity. A confirmatory survey and member checking could help further validate the findings, but the methods we used reveal the lived experiences of our participants [17]. We met with each participant once, so we were not able to see the notebooks they worked on evolve over time. Furthermore, our participants were all part of the same institution as the authors, limiting external validity. Representativeness may also be limited given that 15 of the 20 participants are graduate students, although many scientists who write code are graduate students [18]. Finally, five studies were done in person instead of on Zoom, and the modality differences could have affected participants' performance.

Although the authors are computer scientists, researchers with other perspectives—particularly with more domain knowledge—could have had different understandings of the work we observed, and our software engineering perspective could have biased our interpretations of the efficacy of the tactics used by our participants. In addition, our study could be biased towards the fact that the participants prioritized Jupyter notebooks over other tools (*e.g.*, MATLAB, command line tools) for computing in the observed contexts.

III. Results

We identified three categories of goals that participants had in their work, ranging from *disposable exploration* to *artifact construction* (**RQ1**, Sec. III-B). Participants valued eight non-functional requirements, which we describe using quality attributes (**RQ2**, Sec. III-C). They promoted those quality attributes using 18 tactics (**RQ3**, summarized in Table II). In addition to our research questions, we report how participants used AI-based tools in their notebook work (Sec. III-D).

A. Participants

We recruited 20 participants (11 identifying as male, 9 as female). Seven had prior work experience in software engineering, one had formal training in software engineering,

six self-reported some familiarity with software engineering, and six reported none. Half of the participants (11) were from computing-relevant domains (10 in Computer Science, one in Data Science) and nine from non-computing-relevant domains across Biology (three), Social Sciences (three), and Earth Sciences (four), where P6 had experience in both Social Sciences and Earth Sciences. Table I displays the participants' backgrounds along with the tasks they worked on.

B. Goals for Using Jupyter Notebooks

11 codes pertained to the kinds of goals that participants had in their work. For example, some participants described focusing on scientific findings, whereas others focused on presenting their work to others. We also identified 15 codes relevant to ensuring the clarity of a notebook. These codes include five categories: removing redundant code and cells, taking notes (as comments or Markdown), refactoring code (*e.g.*, renaming variables), inserting empty cells to separate sections, and reformatting or editing code for code and/or output readability. We also noticed varying expectations for notebook longevity—*i.e.*, whether to revisit the notebook after the task demonstrated in the study is done—within the first ten participants, and we started asking about the expectations for notebook longevity explicitly since P11.

Combining these observations, we found that each task fit into one of three categories: *disposable exploration*, *findings*, and *artifact*. *Disposable exploration* refers to exploratory work that will be discarded immediately after the outcome is achieved. We consider a notebook used for disposable exploration if it does not have any expected longevity. An *artifact* details the process and outcome of problem solving and/or scientific discovery in a clear, descriptive, and potentially reproducible way. We consider a notebook to be an *artifact* if the task where it is used is a cleanup task, or it has expected longevity and its author showed three or more kinds of the clarity-related actions (*i.e.*, more than half of the five available kinds, to demonstrate sufficient effort in ensuring clarity from multiple aspects). Finally, a *findings* notebook documents the process and outcome of problem solving and/or scientific discovery but not necessarily in a structured way—its main purpose is to expose information to the notebook author for them to decide on next steps of work. Although a *findings* notebook has some expected longevity, its author used less than half of the possible kinds of clarity-related actions.

Out of 20 participants, we have 21 notebooks; P20 worked on two notebooks during the study, which we denote as P20A and P20B as necessary. We found four notebooks for disposable exploration, nine for findings, and eight for artifact. The first two authors compared the results with each individual study and field notes and agreed upon the categorization. While only P20 worked on more than one notebook during the observations, several participants (P3, P5, P11) shared the context of their notebook work via multiple notebooks prior to the observation or during the post-observation interview.

C. Software Quality Attributes

Participants valued eight quality attributes: clarity, reproducibility, explorability, debuggability, reusability, correctness, performance, and collaboration. We detail each quality attribute, how notebooks promoted inhibited quality, and user tactics that promoted quality (results summarized in Table II).

Clarity. 17 participants described tactics that they used to improve the clarity and presentability of their notebooks. Ten of our participants planned to present their work to their colleagues using their notebooks, so they took extra care to ensure that the notebook was readable and could even be edited and recomputed on the fly. Others, including P17, P18, P19, and P20 reported that they often refer back to code written in previous notebooks and needed to be able to understand and potentially reuse code from them. P17 described reusing code from notebooks dating back to 2018 and said he would continue writing new code in the notebook in the future.

Seven of our participants highlighted that they appreciated the ability to interleave Markdown notes with their code. P10 said that these notetaking abilities in notebooks make it easier for others to understand the code in notebooks, and P20 stressed that *"the markdown function is extremely important to [her]"* when writing exploratory code. For some users like P1, too many notes can have the opposite effect and hinder clarity. Instead, he wrote high-level to-dos elsewhere. P7 had a similar strategy and explained that she writes notes in a separate notetaking software because it was better at tracking history compared to notes inside the notebook. The Markdown capabilities of notebooks also allow users to create sections in their notebooks by creating headings for certain groups of cells, which both P1, P2, and P20 used. Alternatively, P9, P16, and P19 split up their notebooks using multiple empty cells to segregate tested code from exploratory code and to separate different paths of exploration.

Over half of the participants used the flexibility of the cell structure to organize content for clarity. They adopted the heuristic of "one task per cell" to keep relevant lines of code together while maintaining the ability to see the outputs of intermediate computations; however, P1 noted that there was a tension between wanting to group related code and wanting to break apart and inspect inside a cell: *"On one hand, I want the flexibility to be able to look inside a cell and really get into its pieces. But, on the other hand, I also want to be able to flip it over and be like, okay, I've iterated on some kind of structure, and I have this modular building block."*

This form of content organization aids the reuse of code across notebooks since notebook code cannot be exported and must be reused via copy/paste. However, copy-pasting code can impede readability when unnecessary or redundant code is added to a notebook. P18 found some vestigial code copied from another notebook and noted *"sometimes I'll copy and paste old code into here and then I'll just forget to delete it."*

The flat, cell-based structure increased the participants' cognitive load as inspection code and outputs interleaved. P16 remarked that *"once there are too many things [...] happening*

TABLE I
PARTICIPANT BACKGROUNDS AND TASKS.
IN "FIELD", "COMP"=COMPUTING, "BIO"=BIOLOGY, "SOC"=SOCIAL SCIENCES, AND "EARTH"=EARTH SCIENCES.
"&" IN TASK TYPE DENOTES MORE THAN ONE TASK DONE IN THE STUDY.
EACH PARTICIPANT WORKED ON ONE NOTEBOOK EXCEPT P20, AS ";" DENOTES DATA FOR SEPARATE NOTEBOOKS IN GOALS AND SCROLL SIZE.
IN GOALS, "A" REPRESENTS *artifact*, "F" REPRESENTS *findings*, AND "DE" REPRESENTS *disposable exploration*.

ID	Gender	Occupation	Field	SE Experience	Task Description	Task Type	Goals	Scroll Size
P1	M	PhD Student	[Comp]CS	Work	Refactoring a data analysis notebook	Refactoring	A	33
P2	F	PhD Student	[Comp]CS	Work	Algorithm & data comparison	Analysis	F	61
P3	M	PhD Student	[Comp]CS	Training	Data visualization	Visualization	A	65
P4	M	PhD Student	[Comp]CS	Knowledge	Exploratory data analysis	Analysis	F	11
P5	M	PhD Student	[Bio]Bioinformatics	Knowledge	Exploratory algorithm analysis	Analysis	F	50
P6	F	Data Analyst & Researcher	[Soc]Economics & [Earth]Oceanography	Knowledge	Refactoring a data analysis notebook	Refactoring	A	16
P7	F	PhD Student	[Soc]Neuroscience	Knowledge	Reproducing an existing notebook	Reproduction	DE	17
P8	M	MS Student	[Comp]CS	Work	Data cleaning & developing a machine learning model	Data Cleaning & Development	F	47
P9	M	Scientist	[Earth]Geoscience	None	Reproducing data visualizations	Reproduction	F	12
P10	M	Scientist	[Bio]Microbiology	None	Migrating a script to a notebook for documentation	Refactoring	A	71
P11	F	PhD Student	[Earth]Oceanography	None	Data visualization homework assignment	Visualization	A	35
P12	M	PhD Student	[Comp]CS	Work	Code cleanup	Refactoring	F	24
P13	F	Lab Assistant	[Soc]Psychology	None	Data visualization	Visualization	DE	13
P14	F	PhD Student	[Comp]CS	None	Algorithm implementation	Development	F	9
P15	F	PhD Student	[Earth]Geoscience	Knowledge	Data analysis & visualization	Analysis & Visualization	F	21
P16	M	PhD Student	[Comp]CS	Work	Analysis of machine learning models	Analysis	A	32
P17	M	PhD Student	[Comp]CS	Work	Data analysis	Analysis	A	13
P18	F	PhD Student	[Comp]CS	Work	Testing different machine learning models	Analysis	F	35
P19	M	Undergraduate	[Comp]DS	Knowledge	Drafting a programming assignment	Development	DE	30
P20	F	PhD Student	[Bio]Bioinformatics	None	Annotating a notebook & reproducing visualizations (two notebooks)	Notetaking & Reproduction	DE; A	3; 5

in a notebook it becomes hard to follow." To manage information load, P14 would reuse cells for multiple inspections, and both P13 and P16 would delete inspection code to reduce visual clutter. P16 also preserved inspection code in comments for later use to avoid retaining the inspection output.

Five of our participants (P12, P16, P17, P18, P20) reported that they create new notebooks to explore new ideas, debug, and clean up code, which required copy and pasting code from notebooks. P17 explained that he sometimes creates a new notebook and copies over his code cell-by-cell to debug; P2 completed the study using a notebook she created exclusively for debugging. P5 inspected a dataset in an existing, cleaned-up notebook instead of in his current notebook.

Two participants wrote code in functions to explicitly promote clarity in their notebook. P18 explained that she abstracts code into functions to help her focus on relevant information while reading through her notebook. P14 explained in her interview that *"at the end it's nice to have a bunch of functions when the code is cleaned,"* but like P1, she preferred to lose the clarity of functions when developing to ease debugging.

Takeaway 1: Users put effort into ensuring notebook clarity and often use Markdown cells in Jupyter. Whereas typical software engineers rely on abstraction and structure to make code understandable, notebooks' flat structure does not provide these capabilities and often hinders clarity.

Reusability. Traditionally, software engineers leverage modularity and abstraction to promote reusability [19], since abstractions can be reused across contexts without understanding modules' implementation details. In contrast, Jupyter's flat namespace and single scope for all cells both facilitated reuse (by avoiding the need to pass parameters or change representations) and inhibited reuse (by enabling bugs caused by variables having meanings that pertain to irrelevant parts of the program). Participants reported writing functions when code would otherwise be duplicated, but P3 and P11 preferred to duplicate code unless it would result in more than several copies; P11 explained that repetitive code can be easier to read than non-repetitive code that invokes functions.

Jupyter's single scope caused problems for P12, who copied

TABLE II
SUMMARY OF HOW NOTEBOOKS AFFECT QUALITY ATTRIBUTES AND TACTICS PARTICIPANTS USED TO IMPROVE QUALITY

Attribute	Ways notebooks inhibit	Ways notebooks promote	Tactics used to promote
Clarity	Flat, cell-based structure of notebooks makes it difficult to organize information	Markdown notetaking	Abstraction; sectioning; maintaining one task per cell; creating new notebooks
Reusability	Single scope results in accidental variable reuse	Single scope avoids need for parameter passing	AI-based explanations of unfamiliar code
Reproducibility	Out of order cell execution; no built-in package management	Broad usage of Jupyter enables viewing and running others' notebooks	Virtual environments; re-running from the top; readability and cleanliness; storing notebooks alongside data
Explorability	Lack of support for output comparison across runs; information overload; manual cell execution and state management; inability to inspect in the middle of a cell or in the middle of a loop	Cell structure-based interactions and code-ouptut correspondences	Writing intermediate outputs to disk; exploring new ideas in fresh, short notebooks; merging cells with state dependencies
Correctness	Error-prone manual state management	Cell-by-cell execution allows users to check the validity of each line	Cell-based risk management; restart and run all; enforcing linear execution
Performance	Error-prone manual state management	Caching data; saving outputs; cell-by-cell execution	Reusing data; sectioning
Debuggability	Out of order cell execution; single scope when debugging inside a function; difficulty with navigating to relevant buggy cells; inability to inspect in the middle of a cell; enforced kernel restart with changes in dependencies	Cell-based structure promotes small inspection, code-output correspondence, and rapid edit-run cycles	Restart and run all; avoid debugging within function definitions; notetaking
Collaboration	Limited compatibility with file diffing utilities	Broad usage of one notebook tool (Jupyter) makes collaboration easier	Sectioning

and pasted code within a notebook but forgot to update a variable, which was still bound due to its previous use. Fortunately, after seeing plot emitted by a cell, P12 debugged and fixed the problem. Later, P12 encountered another instance of the same problem, but did not notice the bug.

Because of mutation, code reuse within a notebook even for the same purpose is unsafe. Participants often invoked Pandas functions that, for example, renamed columns, so code that is correct before the renaming operation would be incorrect afterward. P1 became unsure which lines of code would change the structure of a dataframe, restarting evaluation to be sure: *"okay, let's start from the top."*

Some participants wanted to reuse code between notebooks and Python scripts. For example, P10 adapted code from a standalone script for use in Jupyter. However, Jupyter couldn't invoke the script's `main` function. P10 refactored the code to use variables instead of command-line arguments. P7 worked with example code that reflected this same pattern: it included a function called `mainfunction`.

Some participants wanted to reuse *unfamiliar* code from other contexts, which required understanding the code to be reused—at least to some extent. P7 relied on ChatGPT to explain unfamiliar code from an example that she wanted to reuse, even though the code included various comments. But these tools were not integrated into Jupyter, so P7 had to copy and paste the code into another window, leaving ChatGPT without the code's surrounding context. P7's query to ChatGPT was only the source code followed by `explain`, missing a possible opportunity to ask a more specific question.

P16 described converting notebook code to scripts, which execute outside Jupyter; this process is facilitated by the fact that Jupyter provides only a thin interface on top of Python.

Takeaway 2: Jupyter's lack of abstraction promotes frequent copy/paste. The flat namespace and single global scope makes copy/paste convenient but error-prone.

Reproducibility. Reproducibility concerns the ability for the developer or others to reproduce the same notebook output in the future. Reproducibility is important in replicating scientific analyses and extending prior work with new analytic techniques. Unfortunately, reproducibility can be a real problem for notebook users. P17 recounted a situation in dealing with a non-reproducible notebook: *"I simply created a blank notebook and copied section by section, because I think* this *section would run [. . .] if it does run, then I move on to the next section, and that does identify the problem."*

Because notebooks permit out-of-order cell execution, re-executing notebooks in order can produce different results than users first observe. At the end of each session, we asked participants to re-run their notebooks in order. P2 and P9 were unable to reproduce their earlier work this way, suggesting that out-of-order execution is a real threat to validity. Some participants (e.g., P3) were careful to write code in order of dependencies, but this process was manual. P6 and P8 used "restart and run all" to make sure their notebooks would run in order; P17 complained about how out of order execution threatens reproducibility. P16 cited mutation and order of execution when explaining why he kept imports at the top.

P1, who had a computing background, used a package manager, Poetry, to create virtual environments for notebooks, enabling specification of dependencies. In contrast, most participants did not appear to be concerned with library versioning, which could threaten reproducibility.

Nine participants considered reproducibility to include read-

ability and cleanliness, since readers might need to understand the code to reproduce the analyses. Matters of readability are discussed under the *Clarity* heading in this section.

Takeaway 3: Scientists value reproducibility, but out-of-order execution and lack of package management hinder it.

Explorability. Exploratory programming is about prototyping ideas and iterating on implementations through code without pre-defined specifications or goals [20]. Regardless of their goals, our participants valued explorability; indeed, explorations are prevalent in programming, and even the process of creating an *artifact* involves exploration. Jupyter includes features that promote exploration: cell-based interactions, facilitating exploration through small inspection (P8, P11, P17), nonlinear execution (P20B), and interleaving code and output for correspondence and quick comparison (P12, P17).

Participants often needed to compare outputs between different versions of their code, but Jupyter did not facilitate this: every time they re-ran a cell, the old output was overwritten. To work around this problem, P1 saved output outside of the notebook for comparison: *"I [would] have code blocks output their result to a file or [. . .] save it somewhere [. . .] and then I'll copy paste that result into like a constant in the code block [for comparison].* P8 chose another approach, leveraging out-of-order cell execution: putting various implementations of an algorithm in different cells enabled comparing the outputs.

Notebooks truncate cell output if it is too long, even though the output could include important information that is easily missed. P9, for example, missed a message indicating an installation failure because it was buried in a long output, instead believing that installation had succeeded. This caused persistent failures when he ran other cells that used functions in that package. Long outputs can also cause difficulties when testing out multiple new ideas by reducing clarity. As a workaround, P12 and P16 started fresh notebooks for exploring new ideas to avoid a notebook getting long due to too many inspection cells and outputs, *"just [making] a new notebook [...] if something gets messy"* (P12).

Jupyter requires users to manually rerun a cell after it has been edited, but users sometimes forgot to do so, leading to unexpected output and changes to the global state. These changes made it difficult to assess the validity of exploratory code. For example, P2 changed the input file she was using to a truncated version in order to do more explorations. However, she forgot to rerun this edited cell and operated under the impression that she was working with the truncated files. This caused her to both waste extra time waiting for the runs to finish and created an extra bug for her to solve, distracting her from her original task. P8 tried to avoid such hiccups by packaging exploratory code into a function with exploratory parameters as the arguments and putting a call to that function in the same cell, so that he could repeatedly call the function with different parameter values to explore outputs.

Finally, similar to how they used markdown notes and annotations to keep notebook content clear, participants used notetaking to facilitate explorations so that they could freely explore without getting lost. This way, P20 said, *"I [could] know that [which] is the the newest exploration [...] and these are all the file paths [...] that I wanted to use [for it]."*

Jupyter's limitation of only showing values of expressions that are at the *ends* of cells frustrated P13, who expected to inspect an expression in the middle of a cell without using the `print` function. Likewise, the promise of expression-based interactivity enabled by the cell structure breaks with loops: P4 could not inspect expressions within a loop unless printing them out and was forced to rerun the whole loop, as opposed to individual iterations, to gain any feedback on code change.

Takeaway 4: Cell-by-cell execution can help users iterate in straight-line code, but the cell model has difficulty scaling to more complex workflows.

Correctness. In traditional software engineering, systematic testing is used to evaluate the correctness of code. In contrast, our participants found it difficult to concisely describe expected results. P17 notes that *"it's not like you can write unit tests to see if things are correct, sometimes you can tell by the data. If the data distribution doesn't look right, then [I] realize maybe I should have done it differently."* Lack of functional decomposition makes it difficult to write unit tests: only five out of 20 participants wrote new functions. In addition to inspecting output manually, participants adopted notebook-related strategies to leverage the notebook environment and mitigate its risks, including *cell-based risk management* (described below), *enforcing linear execution*, and *restarting and rerunning notebooks* for ensuring the correctness of their code.

In *cell-based risk management*, used by four participants (P1, P2, P8, P17), users manage the risks presented by new code by writing in separate cells, which they later combine. For example, P8 created a new cell to remap a categorical column of a dataframe to numerical values. He checked that his code worked as expected by inspecting the datatype of the column. As he needed to do the same for three other columns, he created a new cell and wrote similar code for all three remappings in the same cell and ran the cell without additional inspection. When asked why he chose not to inspect, he said *"I did my proof of concept for the first thing... I know it's going to work because it worked for one of them."*

Because Jupyter requires users to manually manage state, some participants took time to ensure that they had the correct mental model of their notebook's execution. When trying to understand a collaborator's notebook, which involved a lot of variable mutation, P1 said *"I get nervous about this stuff because I don't know if I've reset the state. So, my way of handling that is just restart and run it from the top."* P6, a newcomer to notebooks and programming, runs each of her cells again whenever she makes a major change in her code to ensure that there are no new errors. Others (P2, P3, P8, P12, P17) enforced linear execution in their notebooks.

Restarting and running all notebook cells is not always desirable, especially when working with large data. P4 said,

"the nice thing about Jupyter [...] is like just loading the data and not having to load it every single time when I run a script." Restarting and running all the cells to ensure correctness in this case would counteract the performance benefits of notebooks.

Takeaway 5: Traditional software testing methods are difficult to incorporate in notebooks, so users mitigate risk through inspection and cell-by-cell execution.

Performance. Some participants praised notebooks for facilitating working with large datasets. P4 talked about how his dataset takes 2-3 minutes to load, but using Jupyter allows him to just load it once and run his computations as many times as he needs. P2 explained, *"the point of the Jupyter Notebook is that I have [computations] saved so I can use them later."*

Participants promoted performance in their notebook work by limiting the number of times they loaded data and ran unnecessary computations. Out-of-order execution and splitting up cells allow both reusing loaded data and running the code efficiently. P11 preferred to complete multiple tasks in the same notebook and created sections to separate them. To run the code for a single task, she would run the first notebook cell containing all the import statements she needed, and skip to only the cells in the relevant section.

Initially, P2 wrote code following the "one task per cell" principle. However when debugging code that took a long time to run, she split up her cells based on how often she needed to recompute certain lines and how long they took to run. Cycles of editing and rerunning the split up cells caused some confusion about the current state of the notebook and whether certain cells had been ran after changes. Ultimately, P2 had to rerun each of the expensive computations again in order to confirm that she was working with the most up-to-date outputs. This tactic that was intended to aid performance ended up hindering it when used in the context of debugging.

Takeaway 6: Notebooks benefit data-heavy tasks by enabling partial execution of programs, but users must carefully manage state to leverage this feature.

Debuggability. Debuggability refers to the ability to determine the cause of a bug. All tasks but cleanup tasks P6 and P20A involved some debugging. The cell-based model enabled users to see output of small portions of the program, making the edit-run cycle much faster than in traditional IDEs. Cells enabled participants to compare and connect code to output (P12), run cells out of order (P2, P8, P13), and isolate code for debugging errors (P9, P19). However, out-of-order execution, single global scope, difficulty in finding relevant cells, requiring reloading when dependencies are changed, and expression-based inspection having to be at the end of a cell all interfered with participants' ability to debug efficiently.

Out-of-order execution in notebooks required the user to manually rerun cells that had been edited (and all other cells that depended on them). For example, P15 had written a `for` loop that was supposed to update values in rows of a Pandas data frame. Unfortunately, P15 neglected to index *into* the frame, accidentally rewriting the entire column in every iteration. The first iteration ran with some output, but the second iteration failed due to the unexpected change in the entire column. Confused, P15 decided to rerun the cell to replicate the error, only to see the `for` loop fail immediately during the first iteration: it was now operating on data that had been mutated in the previous (failed) run. 15 minutes into the situation, the participant sought help from the interviewer, who explained that P15 had to reload the data frame, resolve the bug in the loop, then rerun the cell with the loop to finally see the expected output. Neglecting an index may be common in dynamically-typed languages, but a scripting setting would not have produced the output that misled the participant across runs as every run executes the entire script, not just snippets of code. Like P1 pointed out, one must resolve some debugging scenarios in Jupyter notebooks by restarting the kernel and rerunning all cells to enforce the script-like execution linearity.

The single global scope in notebooks also makes debugging and inspecting local variables in a function hard (P3, P10). For example, P10 considered returning local variables he wanted to inspect from a function to use the expression-based inspection in a cell. Instead, P14 simply avoids debugging inside a function. When processing large files inside a function, the edit-run-inspect cycles can take a long time. By moving code out of a function, she could see intermediate output without having to stop, add `print` statements, and rerun the code.

Compared to an IDE for scripts, where one could easily go to the definition of a function or simply a specific line of code to localize the bug, notebooks provided no easy way to navigate to relevant cells (P3, P9, P12). In particular, P9 could not locate the cell he just ran after he scrolled through the notebook to read other cells while waiting for the execution to complete. As such, participants spent a lot of time scrolling through the notebook: in fact, scrolling was the most prevalent action across all studies, with 603 coded instances out of 4195 total action instances (14.3%). To complement the lack of navigation aid in notebooks, participants (P14, P16) took more notes in the notebook to facilitate cell navigation in debugging: *"If you leave notes [...] then if something doesn't work, at least I can go back and look at my notes, [...] start with the things that looked weird intermediately, and go from there"* (P14).

Notebooks required restarting the kernel for changes in the dependencies to take effect, which severely slowed down the edit-run debugging cycles for P7 when some of the debugging-related code changes occurred in an imported module.

Jupyter shows the value of the last expression in each cell. P13 wanted to inspect arbitrary expressions in cells without inserting `print` statements. Enabling easy inspection of all values could further promote debuggability.

Takeaway 7: Cell structure helps isolate errors, but out-of-order execution and single scope impedes debugging because debugging work can mutate state needed elsewhere.

Collaboration. Collaboration is an important part of the sci-

entific process. As students and researchers, our participants needed to ensure that their work could be understood by others, and when needed, could be collaboratively written. Two of the twenty participants planned to co-author their notebooks with their colleagues (P6, P14), but many planned to share and iterate on their notebook work with the input of others (P1, P3, P5, P10, P11, P12, P16, P17). However, Jupyter does not natively provide many tools for facilitating collaboration on the same notebook, so users rely on ad hoc methods such as splitting a notebook into separate sections and storing the notebook on a shared drive. Version control systems have limited benefit because they do not integrate nicely with notebook cells. This also affected our participants' willingness to collaborate with others on notebooks. P8 said that he collaborates with others when working with Python scripts, but chooses not to collaborate with others because of the difficulty of resolving editing conflicts in notebooks. To avoid messy conflicts, P20 elaborated in interview that she had once split a notebook into two sections by adding empty cells in the middle, and she and her collaborator worked on the cells on opposite sides of the divide.

Takeaway 8: Scientists often work together, but a lack of version control integration or other collaboration tools for notebooks makes collaboration difficult.

D. Use of AI tools

11 participants used AI-generated code during their respective studies. Three (P1, P2, P5) used GitHub Copilot [21], which was integrated into their notebook environment (*i.e.*, VS Code). Nine participants (P5, P7, P9, P10, P13, P15, P17, P19, P20) used ChatGPT for help when writing notebook code, and two (P12, P16) mentioned using it in the past.

The participants using Copilot largely used it as an autocomplete tool to accelerate their productivity [22] since it was *"a good time saver,"* (P2). In 18 of the 21 instances of Copilot usage, Copilot would complete their line of code, and 12 instances were accepted without changes. Two were considered unhelpful after reading and were deleted, and three were accepted and edited. P1 also used Copilot to explore solutions by prompting it in the comments to create a plot in a new cell. Once he ran the cell, he saw that the axes labels were unreadable. He tried to fix this issue by prompting Copilot to fix the code, but when unsuccessful, moved on.

Table III shows 14 instances ChatGPT usage, 11 of which had a participant successfully integrate ChatGPT-generated code into their notebook. To validate the AI-generated code, each of these participants first read over the code, and all but one (P5) copied and ran the code inside their notebook to determine if it met their needs. However, this strategy did not always succeed: P9 could not download the dependencies needed to run the generated code. In addition, P7 relied on ChatGPT twice to explain code from the notebook, but noted *"I don't know this code well enough to tell if ChatGPT is giving me something wrong."* All but one ChatGPT user worked in a non-computer science field, which highlights the important role of AI tools for scientists in programming.

TABLE III
USAGE OF CHATGPT

ChatGPT Usage Scenario	Participants	Success Rate
Explaining code	P7	2/2
Fixing errors	P13, P17	2/2
Generating visualization code	P9, P13	3/3
Using unfamiliar libraries	P5, P9, P15, P19	1/5
Other programming tasks	P10, P13, P20	3/4

The exploratory nature of notebooks eases validating AI-generated code because, as P12 puts it, *"if it gives me code that might be wrong, I can just try it."* Still, for those with less coding expertise, it may take them more effort to integrate the code into notebook before validation by execution is possible.

Takeaway 9: Scientists rely on AI-based tools even though they are not integrated into their environments, but sometimes lack the programming knowledge to understand or incorporate AI-based suggestions successfully.

IV. Discussion and Future Work

Identifying quality attributes that *scientists* value exposes opportunities to build theory and deepen our understanding of prior work, which focused on *data workers* more generally. First, our study provides a framework for understanding notebook use and challenges in scientific settings through the lens of quality attributes and identifies tactics scientists use to promote the reported quality attributes. This could enable theory-building opportunities; data collection could focus on the root causes of priorities and motivations, which would enable a stronger theoretical perspective. For example, our study identified conflicting quality attributes for certain goals and contexts (*e.g.*, clarity could conflict with debuggability when a notebook aims to show more intermediate computations), and in-depth data collection could better explain how scientists navigated quality attributes in conflict. Second, with a specific focus on scientific users, while our study reveals notebook usage goals and pain points similar to prior results (of data workers) [6,11,23,24], we found that priorities for notebook quality attributes depended on context and goals (rather than a general prioritization of exploration over explanation).

We noted highlights in the current design of notebooks that supported scientific work and its associated quality attributes. Support for dividing code into cells that can be executed in any order and share an execution environment is the hallmark of the computational notebook paradigm. As we observed, this promotes explorability, which our participants prized. Combined with the support for Markdown-based headings and explanations, which promotes clarity, computational notebooks focus on exactly the quality attributes most valued by our participants. Support for easily seeing output also promotes

correctness and debuggability, and out-of-order execution facilitates work with very expensive analyses (performance). Our participants were less concerned with reproducibility and reusability, which are weaknesses of computational notebooks.

How, then, could future tools for scientists do better? Key design opportunities pertain to reusability and reproducibility, which are inhibited by out-of-order execution, the global scope for all cells, and lack of built-in package management. Here, we believe an opportunity lies in enabling more separation between the cells. Cells could have their own scopes, with explicit control over which variables are imported from and exported to global scope. This could enable a kind of reactivity, similar to spreadsheets or Observable [9], in which cells are automatically re-evaluated when their inputs change.

Similar to that shown in other work [15], we observed a spectrum from exploratory work to explanatory work. Systems need to support work that spans this spectrum over time, but these are sometimes in conflict: out-of-order execution, which promotes explorability, inhibits reproducibility; also, single scope promotes explorability but inhibits some aspects of reusability. Also, some tactics promote some quality attributes at the expense of others, leading to refactoring needs when goals change. For example, a very exploratory notebook might include a large number of short cells to enable quick edit/debug cycles; as the notebook becomes more explanation-oriented, we found that participants were more likely to combine related code into larger cells. Future notebook tools could provide refactoring tools that make transitioning between goals more convenient, when users are ready—and even enable *backward* transitions. Existing tools enable splitting cells, which helps, but documentation can increase viscosity [25]; tools that track a tighter relationship between code and documentation and between different regions of code could make it easier to revise mature code. In our study, we repeatedly observed participants starting over with fresh notebooks when they needed to make these kinds of transitions, creating a mess of different files with unclear histories and necessitating rework to construct each new version.

Modular notebooks. Notebooks have a linear structure: a sequence of code cells, interspersed with Markdown cells. The result is that notebooks *hide dependencies*, inhibiting modularity [25]: code cells can rely on variables that were bound or whose values were mutated in other cells in the notebook. Then, reusing a cell requires first identifying its dependencies. Tools could help find required code [26], but it could be more effective to promote modularity. Dependencies between cells could be restricted; lexical and semantic dependencies could be made explicit (adding annotations of preconditions and postconditions on cells [27]). Gradual approaches could promote formal abstraction mechanisms: refactoring tools could make it easy to extract cells as functions, and notebooks could adapt tools like Projection Boxes to enable live exploration of function behavior [28] (which Engraft [29] has attempted). Another axis of modularity concerns data: code that reads from files or Internet data sources can be buried in notebooks, again hiding dependencies. Notebook environments and languages could improve modularity by making these dependencies, including requirements regarding file formats, explicit.

When executing a notebook, users can choose to run all cells, a single cell, or all cells up to a point. But this inhibits explorability because executing all of these cells can be very expensive. Alternatively, running just one cell is risky because the environment does not track which cells need to be re-run after recent changes. Modularity could enable analyses that let users explore more safely and efficiently.

Organizing large notebooks. Cells in notebooks are in a fixed, scrollable view, making it difficult to see portions of the notebook that pertain to specific tasks. Other tools that facilitate data analysis are more flexible. Spreadsheets include multiple sheets, enabling users to divide analyses into sections. LabVIEW [30], a graphical programming environment that targets scientists and engineers, breaks projects into separate files and libraries. Although Jupyter notebooks can call functions in other notebooks via the `%run` command, doing so pollutes the namespace with all of the referenced functions. Future computational notebooks could be more flexible, allowing users to organize their cells according to their content.

Parallel evaluation. Some cells can take a long time to run. When those cells are executing, no other cells can be evaluated. This restriction relates to notebooks' inability to analyze dependencies: from the environment's perspective, any cell can produce state that is needed for any other cell. But in general, this is not the case, and restricting progress in an unrelated part of the notebook inhibits exploration. A few notebook improvements have adopted dataflow analysis to report or indicate dependencies across cells [31,32]. Based on these works, we could identify non-dependent cells to parallelize their evaluation to address this problem.

Caching partial results. Cells can include some very expensive lines of code, whereas other lines of code are very cheap. Better analysis of dependencies in notebooks [31,32] could enable caching partial results, promoting exploration.

Packaging dependencies. Notebooks often depend on installed packages and data files. Current notebooks do not support specifying dependencies and their versions in metadata. Although users can work around this via virtual environments, notebooks could make this easier with explicit support. Likewise, users must manually bundle input data with notebooks, something that automated support could help with.

Better AI Integration. Our study shows how scientists used AI assistants in notebook work. Our findings align with prior evidence of AI use in IDEs [22,33] and further unveil the need for built-in AI support in notebook environments, which prior work has proposed for general notebook use [34]. Specific to scientists, future AI integrations in notebooks should emphasize supporting the *validation* of AI-generated code [33] (*i.e.*, checking if the generated code matches one's intent) while taking into account their programming and domain expertise.

V. Related Work

Since Knuth's proposal [35], numerous literate programming environments have enabled end-users to incorporate more storytelling [3] into their code [3,9,36]–[39]. Jupyter notebooks allow quickly prototyping ideas through cells and interleaving narratives with code, helping document scientific discoveries and analyses [3]. Indeed, among the long line of computing environments scientists use [3,9,30,37]–[42], Jupyter notebooks [3,39] have become very popular [7].

Corpus Studies. With millions of public Jupyter notebooks on GitHub [3], multiple corpus studies [2,10,15,43]–[45] gained insights into the usage patterns in Jupyter notebooks via such data. These studies identified the tension between exploration and explanation in constructing and sharing notebooks [15], their lack of reproducibility [2], and the lack of good coding practices in notebooks [44]. Building upon prior results, recent work proposed a linear regression model to predict the level of exploration vs. explanation in a notebook [45], developed a taxonomy of bugs in notebooks [10], and analyzed refactoring behavior across the evolution of notebooks [43].

These studies show that while notebooks can evolve from exploration-focused to explanation-oriented by introducing more clarity [45], and notebook users do attempt debugging [10] and refactoring [43], notebook code may be of low quality according to traditional metrics, such as presence of unused module imports [2,44]. These quality metrics, however, may be less relevant for scientists than the higher-level quality attributes that we identified. In addition, these studies analyze notebooks on GitHub, which might miss some insights since scientists often choose not to publish notebooks on GitHub except for sharing [7]. Our study revealed notebook-specific quality attributes that scientists valued, tactics they performed to promote quality, and the difficulty in achieving quality goals without support for modularity, scoping, and refactoring—a cost they had to bear to optimize exploration.

Studies with Humans. While corpus studies derive notebook usage patterns through notebook artifacts, interviews, observational studies, and surveys seek the answer directly from notebook users (mainly data scientists). Wang et al. [46] conducted the first observational study where pairs of data scientists collaborated in a notebook task, focusing on collaboration patterns but not the low-level notebook-related actions.

Four prior works, although targeting general data workers as opposed to scientsts, are particularly relevant to ours. Through interviews and surveys, Kery et al. [6] found that data scientists prioritized exploration over explanation, and this prioritization often backfired when revisiting work later; in contrast, we found that scientists' priorities depend on context and goals. For example, P17 carefully documented exploratory code with Markdown notes, expecting to revisit his code years later, even though documentation could inhibit exploration later by increasing modification costs. As Subramanian et al. [23] found from screen recordings of nine data workers performing their own tasks, our study showed that scientists used notebooks for both experimentation and results sharing; additionally, while this work showed that notebooks could easily become disorganized, most scientists in our study valued clarity in notebooks. Our participants adopted tactics to promote clarity in their notebooks, but these same tactics often inhibited other quality attributes they valued (*e.g.*, avoiding debugging within function definitions, which maintained clarity but limited debuggability). Chattopadhyay et al. [11] conducted observations, interviews, and surveys with industrial data scientists and engineers, revealing nine pain points when using notebooks. Although their study population differs from ours, our participants encountered similar issues, including a lack of built-in (AI) code assistance tools and difficulty refactoring code, hitting all pain points except data security; in addition, our work surfaced tactics scientists adopted to work around the pain points (*e.g.*, P20 split a notebook into two sections to address limitations in collaborative work). Finally, in addition to conducting two corpus studies, Rule et al. [15] interviewed 15 academic data analysts who felt that although messes built up easily, notebook clarity was unnecessary unless for sharing; in contrast, we found clarity to be a top quality attribute that scientists cared about, even without sharing or collaboration, with four participants explicitly using tactics to promote notebook clarity in non-collaborative work, such as adding documentation and writing modular code.

Notebook Improvements and Novel Systems. Driven by the existing challenges with Jupyter notebooks, researchers have reviewed the design of computational notebooks [8] and proposed analysis techniques [31,47], novel notebook systems [9,29,48], and Jupyter extensions [26,31,32,49]–[51] to improve the notebook quality and user experience. Some systems incorporate informal version control into notebooks to help users explore and compare code alternatives [49]–[51]. Some tackle the error-prone manual state management issue in Jupyter by reporting unsafe notebook executions that lead to out-of-sync data dependencies [31] or allowing dataflow execution across cells [9,32,48]. Other systems aim to reduce the potential clutter created during the exploration process by providing more live feedback [29] and cleaning up redundant code with program slicing [26]. Our study complements these systems by proposing design opportunities for future improvements to Jupyter notebooks grounded in observational data.

VI. Conclusion

Scientists value eight different quality attributes in their work and use 18 tactics to promote those quality attributes. The cell model in computational notebooks promotes key quality attributes, such as *explorability*, that scientists value, partly explaining their dominance among scientists. Although the model also inhibits other valued quality attributes, such as reproducibility, future changes to the notebook model could enable scientists to meet their quality goals and navigate the spectrum from *exploration* to *explanation* more effectively.

Acknowledgments

We thank the anonymous reviewers for their suggestions. This work was supported in part by NSF grant 2107397.

References

[1] Project Jupyter, "Jupyter," 2024. [Online]. Available: https://jupyter.org

[2] J. F. Pimentel, L. Murta, V. Braganholo, and J. Freire, "A Large-Scale Study About Quality and Reproducibility of Jupyter Notebooks," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. Montreal, QC, Canada: IEEE, May 2019, pp. 507–517. [Online]. Available: https://ieeexplore.ieee.org/document/8816763/

[3] B. E. Granger and F. Pérez, "Jupyter: Thinking and Storytelling with Code and Data," *Computing in Science & Engineering*, vol. 23, no. 2, pp. 7–14, 2021.

[4] M. Burnett, "End-User Software Engineering and Why it Matters," *J. Organ. End User Comput.*, vol. 22, no. 1, p. 1–22, January 2010. [Online]. Available: https://doi.org/10.4018/joeuc.2010101904

[5] B. R. Barricelli, F. Cassano, D. Fogli, and A. Piccinno, "End-User Development, End-User Programming and End-User Software Engineering: A Systematic Mapping Study," *Journal of Systems and Software*, vol. 149, pp. 101–137, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121218302577

[6] M. B. Kery, A. Horvath, and B. Myers, "Variolite: Supporting Exploratory Programming by Data Scientists," in *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, ser. CHI '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 1265–1276. [Online]. Available: https://doi.org/10.1145/3025453.3025626

[7] E. Pertseva, M. Chang, U. Zaman, and M. Coblenz, "A Theory of Scientific Programming Efficacy," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3639139

[8] S. Lau, I. Drosos, J. M. Markel, and P. J. Guo, "The Design Space of Computational Notebooks: An Analysis of 60 Systems in Academia and Industry," in *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2020, pp. 1–11.

[9] Observable, "Observable." [Online]. Available: https://www.observablehq.com

[10] T. L. De Santana, P. A. D. M. S. Neto, E. S. De Almeida, and I. Ahmed, "Bug Analysis in Jupyter Notebook Projects: An Empirical Study," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 4, April 2024. [Online]. Available: https://doi.org/10.1145/3641539

[11] S. Chattopadhyay, I. Prasad, A. Z. Henley, A. Sarma, and T. Barik, "What's Wrong with Computational Notebooks? Pain Points, Needs, and Design Opportunities," in *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, ser. CHI '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1–12. [Online]. Available: https://doi.org/10.1145/3313831.3376729

[12] Ko, Amy J. and Myers, Brad A. and Coblenz, Michael J. and Aung, Htet Htet, "An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks," *IEEE Transactions on Software Engineering*, vol. 32, no. 12, pp. 971–987, 2006.

[13] K. Charmaz, *Constructing Grounded Theory*, 2nd ed. Sage, 2014.

[14] M. Coblenz, G. Kambhatla, P. Koronkevich, J. L. Wise, C. Barnaby, J. Sunshine, J. Aldrich, and B. A. Myers, "PLIERS: A Process that Integrates User-Centered Methods into Programming Language Design," *ACM Trans. Comput.-Hum. Interact.*, vol. 28, no. 4, jul 2021. [Online]. Available: https://doi.org/10.1145/3452379

[15] A. Rule, A. Tabard, and J. D. Hollan, "Exploration and Explanation in Computational Notebooks," in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, ser. CHI '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 1–12. [Online]. Available: https://doi.org/10.1145/3173574.3173606

[16] R. Huang, S. Ravi, M. He, B. Tian, S. Lerner, and M. Coblenz, "Artifact: How Scientists Use Jupyter Notebooks: Goals, Quality Attributes, and Opportunities," February 2025. [Online]. Available: osf.io/c759a

[17] L. Yardley, "Demonstrating Validity in Qualitative Health Research," *Qualitative Psychology: A Practical Guide to Methods*, pp. 235–251, 2008.

[18] J. E. Hannay, C. MacLeod, J. Singer, H. P. Langtangen, D. Pfahl, and G. Wilson, "How do scientists develop and use scientific software?" in *2009 ICSE Workshop on Software Engineering for Computational Science and Engineering*, 2009, pp. 1–8.

[19] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 2nd ed. Addison-Wesley, 2003.

[20] M. B. Kery and B. A. Myers, "Exploring Exploratory Programming," in *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2017, pp. 25–29.

[21] N. Friedman, 2021. [Online]. Available: https://github.blog/2021-06-29-introducing-github-copilot-ai-pair-programmer/

[22] S. Barke, M. B. James, and N. Polikarpova, "Grounded Copilot: How Programmers Interact with Code-Generating Models," *Proc. ACM Program. Lang.*, vol. 7, no. OOPSLA1, Apr. 2023. [Online]. Available: https://doi.org/10.1145/3586030

[23] K. Subramanian, N. Hamdan, and J. Borchers, "Casual Notebooks and Rigid Scripts: Understanding Data Science Programming," in *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. Dunedin, New Zealand: IEEE, Aug. 2020, pp. 1–5. [Online]. Available: https://ieeexplore.ieee.org/document/9127207/

[24] A. Rule, I. Drosos, A. Tabard, and J. D. Hollan, "Aiding Collaborative Reuse of Computational Notebooks with Annotated Cell Folding," *Proceedings of the ACM on Human-Computer Interaction*, vol. 2, no. CSCW, pp. 1–12, Nov. 2018. [Online]. Available: https://dl.acm.org/doi/10.1145/3274419

[25] A. F. Blackwell, C. Britton, A. Cox, T. R. Green, C. Gurr, G. Kadoda, M. S. Kutar, M. Loomes, C. L. Nehaniv, M. Petre, and C. Roast, "Cognitive Dimensions of Notations: Design Tools for Cognitive Technology," in *Cognitive Technology: Instruments of Mind: 4th International Conference, CT 2001 Coventry, UK, August 6–9, 2001 Proceedings*. Springer, 2001, pp. 325–341.

[26] A. Head, F. Hohman, T. Barik, S. M. Drucker, and R. DeLine, "Managing Messes in Computational Notebooks," in *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, ser. CHI '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 1–12. [Online]. Available: https://doi.org/10.1145/3290605.3300500

[27] C. A. R. Hoare, "An Axiomatic Basis for Computer Programming," *Commun. ACM*, vol. 12, no. 10, p. 576–580, oct 1969. [Online]. Available: https://doi.org/10.1145/363235.363259

[28] S. Lerner, "Projection Boxes: On-the-fly Reconfigurable Visualization for Live Programming," in *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, ser. CHI '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1–7. [Online]. Available: https://doi.org/10.1145/3313831.3376494

[29] J. Horowitz and J. Heer, "Engraft: An API for Live, Rich, and Composable Programming," in *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*, ser. UIST '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3586183.3606733

[30] National Instruments. (2024) What is NI LabVIEW? [Online]. Available: https://www.ni.com/en/shop/labview.html

[31] S. Macke, H. Gong, D. J.-L. Lee, A. Head, D. Xin, and A. Parameswaran, "Fine-Grained Lineage for Safer Notebook Interactions," *Proc. VLDB Endow.*, vol. 14, no. 6, p. 1093–1101, February 2021. [Online]. Available: https://doi.org/10.14778/3447689.3447712

[32] D. Koop and J. Patel, "Dataflow Notebooks: Encoding and Tracking Dependencies of Cells," in *9th USENIX Workshop on the Theory and Practice of Provenance (TaPP 2017)*, 2017.

[33] K. Ferdowsi, R. L. Huang, M. B. James, N. Polikarpova, and S. Lerner, "Validating AI-Generated Code with Live Programming," in *Proceedings of the CHI Conference on Human Factors in Computing Systems*, ser. CHI '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3613904.3642495

[34] A. M. Mcnutt, C. Wang, R. A. Deline, and S. M. Drucker, "On the Design of AI-powered Code Assistants for Notebooks," in *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, ser. CHI '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3544548.3580940

[35] D. E. Knuth, "Literate Programming," *The Computer Journal*, vol. 27, no. 2, pp. 97–111, 1984.

[36] J. Wilkin, "Software Reviews," *The College Mathematics Journal*, vol. 29, no. 1, pp. 62–65, 1998. [Online]. Available: https://doi.org/10.1080/07468342.1998.11973917

[37] B. Hayes, "Thoughts on Mathematica," *Pixel*, vol. 1, pp. 28–34, 1990.

[38] B. Baumer and D. Udwin, "R Markdown," *Wiley Interdisciplinary Reviews: Computational Statistics*, vol. 7, no. 3, pp. 167–177, 2015.

[39] F. Perez and B. E. Granger, "IPython: A System for Interactive Scientific Computing," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 21–29, 2007.

[40] S. Matlab, "MATLAB," *The MathWorks, Natick, MA*, vol. 9, 2012.

[41] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A Fresh Approach to Numerical Computing," *SIAM review*, vol. 59, no. 1, pp. 65–98, 2017.

[42] Spyder, "Spyder IDE," 2024. [Online]. Available: https://www.spyder-ide.org/

[43] E. S. Liu, D. A. Lukes, and W. G. Griswold, "Refactoring in Computational Notebooks," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 3, pp. 1–24, Jul. 2023. [Online]. Available: https://dl.acm.org/doi/10.1145/3576036

[44] K. Grotov, S. Titov, V. Sotnikov, Y. Golubev, and T. Bryksin, "A Large-Scale Comparison of Python Code in Jupyter Notebooks and Scripts," in *Proceedings of the 19th International Conference on Mining Software Repositories*, ser. MSR '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 353–364. [Online]. Available: https://doi.org/10.1145/3524842.3528447

[45] D. Raghunandan, A. Roy, S. Shi, N. Elmqvist, and L. Battle, "Code Code Evolution: Understanding How People Change Data Science Notebooks Over Time," in *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, ser. CHI '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3544548.3580997

[46] A. Y. Wang, A. Mittal, C. Brooks, and S. Oney, "How Data Scientists Use Computational Notebooks for Real-Time Collaboration," *Proceedings of the ACM on Human-Computer Interaction*, vol. 3, no. CSCW, pp. 1–30, Nov. 2019. [Online]. Available: https://dl.acm.org/doi/10.1145/3359141

[47] S. Titov, Y. Golubev, and T. Bryksin, "ReSplit: Improving the Structure of Jupyter Notebooks by Re-Splitting Their Cells," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Mar. 2022, pp. 492–496, iSSN: 1534-5351. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9825898

[48] P. Shen, "natto.dev - write JavaScript on a 2D canvas," 2024. [Online]. Available: https://natto.dev/

[49] M. B. Kery, M. Radensky, M. Arya, B. E. John, and B. A. Myers, "The Story in the Notebook: Exploratory Data Science using a Literate Programming Tool," in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. Montreal QC Canada: ACM, Apr. 2018, pp. 1–11. [Online]. Available: https://dl.acm.org/doi/10.1145/3173574.3173748

[50] Z. J. Wang, K. Dai, and W. K. Edwards, "Stickyland: Breaking the linear presentation of computational notebooks," in *Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems*, ser. CHI EA '22. New York, NY, USA: Association for Computing Machinery, 2022. [Online]. Available: https://doi.org/10.1145/3491101.3519653

[51] N. Weinman, S. M. Drucker, T. Barik, and R. DeLine, "Fork It: Supporting Stateful Alternatives in Computational Notebooks," in *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, ser. CHI '21. New York, NY, USA: Association for Computing Machinery, May 2021, pp. 1–12. [Online]. Available: https://dl.acm.org/doi/10.1145/3411764.3445527