Tidynote: Always-Clear Notebook Authoring

Ruanqianqian (Lisa) Huang
University of California San Deigo
La Jolla, California, USA
r6huang@ucsd.edu

Brian Hempel
University of California San Deigo
La Jolla, California, USA
bhempel@ucsd.edu

Yining Cao
University of California San Diego
La Jolla, California, USA
rimacyn@ucsd.edu

James D. Hollan
Design Lab
University of California San Diego
La Jolla, California, USA
hollan@ucsd.edu

Haijun Xia
University of California San Diego
La Jolla, California, USA
haijunxia@ucsd.edu

Sorin Lerner
Cornell University
Ithaca, New York, USA
sorin.lerner@cornell.edu

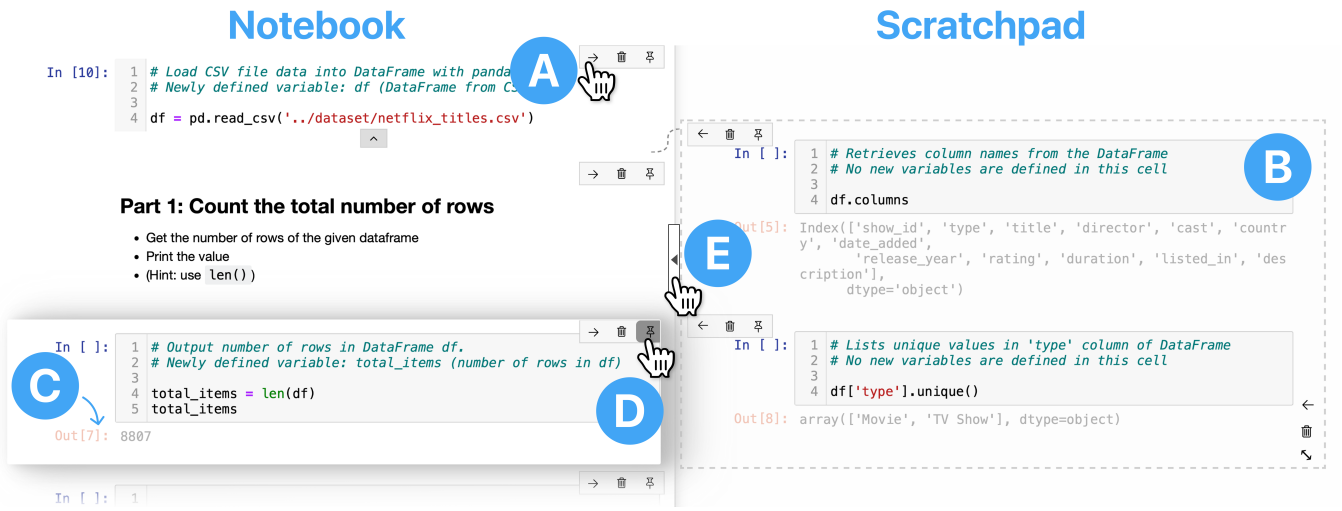

Figure 1: Tidynote divides a Jupyter notebook into clear content (left) and a scratchpad for exploration and references (right). A A notebook cell can be moved to the B scratchpad, where cell execution has no effect on the main notebook. Linear execution ensures clarity in state, so rerunning a cell (A) causes all subsequent cell outputs in the notebook and the scratchpad to gray out C, indicating their staleness. D One can pin a cell so that it floats on top of the screen during scrolling. E A toggle shows/hides the scratchpad as needed. Together, these features enable always-clear notebook authoring.

Abstract

Recent work identified *clarity* as one of the top quality attributes that notebook users value, but notebooks lack support for maintaining clarity throughout the exploratory phases of the notebook authoring workflow. We propose *always-clear notebook authoring* that supports both clarity and exploration, and present a Jupyter implementation called Tidynote. The key to Tidynote is three-fold: (1) a scratchpad sidebar to facilitate exploration, (2) cells movable between the notebook and the scratchpad to maintain organization, and (3) linear execution with state forks to clarify program state. An exploratory study (N=13) of open-ended data analysis tasks shows that Tidynote features holistically promote clarity throughout a notebook's lifecycle, support realistic notebook tasks, and enable novel strategies for notebook clarity. These results suggest that Tidynote supports maintaining clarity throughout the entirety of notebook authoring.

CCS Concepts

• **Human-centered computing** → **Interactive systems and tools**; • **Software and its engineering** → *Development frameworks and environments.*

Keywords

Jupyter notebooks, programming interfaces

ACM Reference Format:
Ruanqianqian (Lisa) Huang, Brian Hempel, Yining Cao, James D. Hollan, Haijun Xia, and Sorin Lerner. 2026. Tidynote: Always-Clear Notebook Authoring. In *Proceedings of the 2026 CHI Conference on Human Factors in Computing Systems (CHI '26), April 13–17, 2026, Barcelona, Spain.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3772318.3791919

CHI '26, Barcelona, Spain

ACM ISBN 979-8-4007-2278-3/26/04
https://doi.org/10.1145/3772318.3791919

1 Introduction

Computational notebooks, such as Jupyter [32], have become widely popular among end-users by enabling storytelling through code [10]. Notebooks divide a program into *cells* that can be interspersed with explanatory text, images, and formulas. Users can execute code cells individually, often out of order [23], and thus easily perform exploratory programming [20]. Explorations are central to construct narratives in notebooks [21], yet they are mostly small code cells [15, 21] and "dirty tricks" [37] that can quickly generate clutter, often with outdated and out-of-order results from prior runs. A notebook filled with such explorations becomes *messy*, difficult for one to understand its narratives [21] and in need of "cleaning" [37] to restore its clarity.

A well-known challenge in notebooks is the tension between exploration and the need for clarity [15, 21, 26, 33, 37]. Clarity is required in many phases throughout the notebook lifecycle, such as sharing results [15, 41], tracking longitudinal findings [15], and providing tutorials [26]. To ready a notebook for these purposes, prior systems support *post-hoc* cleaning by turning portions of a notebook into slides [24, 44, 49], cartoons [18], dashboards [45], and videos [29]; collapsing cells into sections [36]; and gathering cells into complete slices [14]. However, these systems reinstate notebook clarity *long after* the messes have been created.

Although some users may be content with post-hoc cleaning, a recent study [15] found that many users maintain clarity *continuously*, while they edit a notebook, not just after. They want to never let the mess get out of control. However, neither Jupyter nor the post-hoc cleaning systems above provide direct support for continuous cleaning. Consequently, these users adopt tedious manual workarounds, such as copying code to a fresh notebook for debugging, to maintain clarity during authoring [15].

No prior systems support the needs of these users. Post-hoc cleaning systems fail to confine and control messes *as* they are created. Therefore, we propose *always-clear notebook authoring*, an authoring strategy that supports both clarity and exploration throughout the *entire* notebook lifecycle, implemented in a prototype named Tidynote for Jupyter notebooks [10]. In Tidynote, continuous clarity is enabled via three mechanisms. First, while the notebook captures the main narrative, a collapsible *scratchpad* on the side stores work in progress, references, and one-off explorations (Fig. 1 B). Second, cells can *move between* the scratchpad and notebook as explorations progress into clear work or vice versa (Fig. 1 A), with optional cell pinning to reduce scrolling (Fig. 1 D). The ability to move cells back and forth encourages fluid authoring without "premature commitment" [11] to either clarity or exploration. Third, cells are executed linearly, top-to-bottom, to align with the flow of information, with cells out of sync with the state grayed out (Fig. 1 C). This promotes clarity in program state. Using these features, the main notebook can always be clear, in both structure and state: one can easily revisit their work from top to bottom or present their work to others without showing the messy explorations in the scratchpad. Finally, Tidynote and standard Jupyter are mutually compatible: ordinary Jupyter functionality remains available in Tidynote, and notebooks created with Tidynote can be used in standard Jupyter.

We demonstrate how Tidynote supports always-clear notebook authoring via an exploratory study. 13 participants performed open-ended data analysis tasks using Tidynote with the goal of clearly encapsulating their work process for future reference. Our findings show that all participants used all Tidynote features to perform their desired explorations in realistic notebook tasks while maintaining notebook clarity, with various strategies for clarity enabled by Tidynote. Even those participants who were previously careless about clarity were motivated to adopt clarity-driven workflows. We conclude the paper with a discussion of notebook tasks that are best suited by always-clear notebooks and enumerate design opportunities for future notebook systems and information systems.

This paper contributes:

- Tidynote, a prototype system implemented as a Jupyter Notebook extension that supports the always-clear notebook authoring workflow;
- A user study demonstrating Tidynote's effectiveness in supporting always-clear notebook authoring.

2 Related Work

Our goal is to simultaneously support clarity and nonlinear exploration in notebook authoring. To this end, we review (1) challenges in using notebooks—particularly the tension between exploration and clarity; (2) systems for cleaning notebooks; and (3) alternative interfaces that support nonlinear exploration.

2.1 Pain Points in Using Notebooks

Computational notebooks such as Jupyter [10] have become popular for data work, scientific computing, and machine learning [10, 23] due to their support for exploratory programming [20] and narrating through code [22]. Their flat, cell-based structure [15] and nonlinear execution model [30] not only distinguish them from traditional software development environments but also lead to unique usage challenges.

Challenges identified in prior work include low reproducibility [8, 31] and poor code quality [12]. While these findings have motivated best practice recommendations similar to software development guidelines [31], these recommendations often do not align with how notebook users actually work. For instance, Liu et al. [26] found that refactoring in notebooks differs largely from traditional software refactoring, and that notebooks with different focuses (*e.g.*, exploration vs. exposition) exhibit different refactoring patterns.

Additional challenges concern user pain points, such as difficulty understanding program state, managing version history, and deploying to production [5, 15, 19, 41]. These results suggest that notebook users work with different goals and requirements than those supported by software development tools, and yet their needs remain under-supported.

One possible explanation for the above challenges is the tension between exploration and clarity. Corpus studies [26, 33, 37] and direct studies with users [15, 21, 37] highlight how notebooks range from exploratory work full of "messes" to artifacts with clear narratives, but the transition between these modes is effortful and often irreversible. Indeed, cleaning up exploratory work for clarity and presentation can require significant manual effort in content

restructuring and code deletion, with users frequently relying on informal tactics like commenting out code, creating fresh notebooks, or sectioning via markdown headings [15, 19, 21]. This tension is exacerbated in collaborative settings, where disorganized notebooks hinder shared understanding [42]. In reality, a notebook's lifecycle can be filled with back-and-forth iterations between exploratory work and intermediate presentations, and users have created many strategies that adapt to their needs of clarity: some users delay cleanup until sharing [37], some always prioritize finding solutions over maintaining code quality [19], while others always keep a notebook clear even when not collaborating or creating artifacts [15]. Our work aims to embrace the spectrum of notebook clarity by simultaneously supporting clarity and exploration throughout the entire notebook lifecyle.

2.2 Systems for Cleaning Notebooks

Some systems aim to help the user create cleanliness in their notebook. Some provide extra structure to explain notebook organization and some help distill a notebook into a streamlined form for presentation. These are best suited for *post hoc* cleaning. We discuss these systems below, as well as StickyLand [45], which, like our envisioned system, supports cleaning *during* notebook authoring rather than just post hoc.

Several systems augment the notebook structure to facilitate clear organization of work [4, 25, 36]. These structure-focused systems allow users to collapse or hide groups of cells under named headers [36], annotate sections [4], or link explanatory text to corresponding code and outputs [25]. However, these systems provide an overhead in establishing notebook structure in exploratory settings; more importantly, these techniques are best suited for *post hoc* organization because the overhead needs to be repeated as a notebook evolves. Our work aims to maintain clarity in notebook organization *during* fluid exploration with minimal overhead.

Another class of tools turn notebooks into presentable formats. From notebooks, prior systems auto-generate slide decks [49] and associated bullet points [44], dashboards with storytelling annotations [24], data comics [18], or videos [29], but they assume the notebooks are already clear and organized. Code gathering tools [14] slice execution logs to produce a minimal, presentable notebook slice from a selected output; while this reduces clutter within one notebook, repetitive cells could persist across notebook slices, and cleaned slices quickly fall out of sync with ongoing edits in the original notebook. Compared to the above systems, which focus on *post hoc* mess management to support clear presentations from messy explorations, we propose controlling notebook mess *during* exploration while enabling easy transition to presentation.

Perhaps closest to our work in terms of functionality, StickyLand [45] allows pinning cells in a floating dashboard on top of a messy notebook, maintains provenance between the source notebook and the dashboard, and enables quick dashboard-based presentation without explicit cleaning. However, StickyLand differs from our work in design philosophy: "sticky" cells are flexible, but ambiguous in their purpose because they can range from transient reference material to presentable results, whereas Tidynote's scratchpad is less ambiguously intended for scratch work. If a StickyLand user wants the main notebook to be their presentable artifact instead of the sticky space, then the sticky space is less useful because it does not make sense to put all exploratory code into a floating space. We analyze StickyLand and other closely-related systems in more depth in Sec. 3, in relation to our design goals.

2.3 Notebook State Management and Exploratory Interfaces

Some non-Jupyter systems provide alternative state management models to mitigate state messes [3, 27, 28, 34]. Providing automatically managed state, such as automatically rerunning dependent cells, can alleviate state confusion, but does not necessarily promote exploration or provide a way to separate presentation-ready content from archival or temporary work. Thus, other systems aim to improve nonlinear exploration itself. For traditional scripting, Variolite puts in situ version control on code blocks to facilitate variants and comparisons [19], but does not have the benefit of cell-based execution as in notebooks. In the context of notebooks, Fork It [46] enables state forking and backtracking and displays branches side by side, and Kishuboard [9] brings version control to the data level. However, the clarity of program state can still remain problematic in these systems due to the standard nonlinear execution. 2D canvas nodes-and-wires notebook systems like natto [40] let users spread alternative explorations along a second spacial dimension, but wires can become messy "spaghetti" [47], and organizing the 2D space requires deliberate and careful user intervention. Instead of building another 2D nodes-and-wires system that could lead to spaghetti, we are inspired by the inline annotation interface of TextTearing [48] and in-situ branching interfaces [16, 19, 46] to preserve the original linear display of cells as much as possible. We thus minimally augment the linear cell order with *one level of branching* to allow for intuitive nonlinear exploration.

3 Always-Clear Notebook Authoring

The goal of our work is to support the clarity of a notebook *throughout its lifecycle*—from initial creation, to any number of intermediate presentations, and to continued editing after each presentation. This goal entails clarity requirements in both the content and the program state.

While there are approaches attempting to manage notebook clarity, primarily after messes are created, there are no prior systems that fully achieve the wholistic nature of the above goal. To inform the design of our envisioned "always-clear" notebook authoring, we first examine how existing tools and practices manage the tension between exploration and clarity during notebook authoring. We analyze three representative strategies—out-of-notebook cells, post-hoc cleaning, and state forks, examining how each supports and/or fails to support the clarity needs throughout the notebook authoring process (Sec. 3.1). This analysis reveals three critical weaknesses that prevent existing approaches from keeping notebooks *consistently* interpretable and presentable. These weaknesses directly motivate our design goals and the design decisions behind Tidynote (Sec. 3.2).

3.1 Comparative Analysis

Through the literature [5, 15, 21, 37], we first identify five main user actions in keeping a notebook clear throughout its life cycle:

Strategies (w/ exemplar work)		Global Exploration A new exploration of the imported dataset	Cell-Based Exploration A different distance calculation method	Clear Unused Exploratory Work Preparing for results presentation	Iterate Between Clear & Exploratory Work Intermediate sharing & receiving feedback	Reduce Cell State Messes Restoring state after out-of-order execution or deleting exec'd cells
Out-of-Notebook Cells	Scratchpad Extension [35]	A hidable cell on the side	DISALLOWED	NO ADDITIONAL SUPPORT	NO ADDITIONAL SUPPORT	NO ADDITIONAL SUPPORT
	StickyLand [45]	NO ADDITIONAL SUPPORT		A floating window	Move between Notebook & floating window	
Post-hoc Cleaning	Janus [36]			Hide cells to the side	NO ADDITIONAL SUPPORT	
	Gather [14]			Slice cells to new Notebook		
State Forks	Fork It [46]	Create cell branches		Delete branches	DISALLOWED	
Always-Clear Authoring	*Tidynote*	Scratchpad on the side	Cell-based scratch work	Hide Scratchpad	Move cells between Notebook and Scratchpad	Linear execution with state fork

Figure 2: *Comparative analysis of different strategies for managing notebook messes.* We compare the mechanisms of three existing strategies—out-of-notebook cells [35, 45], post-hoc cleaning [14, 36], and state forks [46]—with our envisioned *always-clear notebook authoring*, across five actions relevant to keeping a notebook clear throughout its lifecycle—performing exploration (global and cell-based), clearing unused exploration, iterating between exploration and clarity, and reducing state messes. "No additional support" means the actions are achievable manually, but the corresponding approach does not provide additional support beyond the base, traditional Jupyter; "Disallowed" means such actions are unattainable at all. Our approach is featured in the last row, the only strategy that supports keeping a notebook clear throughout its entire lifecycle.

structuring exploration (1) globally and (2) cell-wise, (3) cleaning up exploration, (4) maintaining exploration while keeping the notebook interpretable, and (5) reasoning about program state. We review how each relevant prior strategy [14, 35, 36, 45, 46] supports these user actions or fails to do so.

To explain how our proposal compares to prior approaches, Fig. 2 shows our comparative analysis, showing the above user actions as column headers and the reviewed strategies in each row; our approach is shown in the last row. We now go into more details of each strategy and their support for each relevant user action.

Out-of-Notebook Cells. Both the Jupyter scratchpad extension [35] and StickyLand [45] introduce auxiliary spaces—a side cell or floating windows—to give users room to explore without cluttering the main notebook. While these structures offer an external workspace, they do not improve the clarity of the main notebook itself, which remains the primary "battlefield" where users read, write, and present results. Furthermore, these systems address content clarity but not state clarity. They retain Jupyter's global execution semantics: all cells, regardless of their location, share the same underlying state. Exploratory code in a scratchpad or floating window can therefore modify variables and outputs throughout the notebook, disallowing any exploration that is meant for just individual cells. As such, users remain responsible for preventing state inconsistencies with these approaches, limiting the usefulness of these auxiliary spaces for maintaining overall clarity.

Post-hoc Cleaning. While providing no additional support for exploration, post-hoc cleaning approaches allow creating clarity *after* exploration: Janus lets users move cells to a folded tab on the side [36], while Gather allows slicing out a subset of cells into a separate notebook [14]. These systems envision cleanup as a final step, rather than an intermittent process, so their support for juggling between exploration and clarity is limited to manual copy/paste of code: Janus disallows moving the folded side-cells back, and slices out of Gather are independent from the source notebook. The post-hoc nature of the cleanup does not suit users who want to maintain clarity *throughout* authoring [15].

State Forks. Fork It [46] supports state branching, where each branch contains a fork of the notebook state. Users compare branches during exploration and remove unnecessary branches to restore clarity. However, only one branch point is allowed at a time, so explorations must be deleted before creating branches at a different point. This limitation discourages juggling between exploration and clarity, and impedes the user from archiving explorations. And although each branch forks the notebook state at the branch point, Fork It keeps the default Jupyter execution semantics where one can run cells out of order, which led to confusion in their user study. Because of these limitations in branching and state clarity, maintaining clarity throughout notebook authoring still requires considerable manual effort.

Summary of Insights & Weaknesses. Taken together, these strategies provide valuable insights into promoting clarity in notebook—auxiliary exploration spaces, post-hoc cleaning mechanisms, and state forking. However, we also identify three key weaknesses that keep these prior techniques from enabling clarity throughout a notebook's *entire* lifecycle.

Lack of flexible, structured exploratory spaces. In practice, exploratory work is often scoped to certain cell(s) (which is part of the motivation for PageBreaks [34]). Existing auxiliary spaces operate only at the global level and remain decoupled from the cells where exploration originates, both visually and state-wise, offering no structured way to support cell-level or multi-scale explorations.

Insufficient support for fluid iteration between exploratory and clear work. During notebook authoring, exploration might mature into clear content, while previously clear content could regress to exploration (*e.g.*, archived but not removed [15, 21]). Such limited support thus contradicts the dynamic nature of how a notebook evolves and inhibits always-clear authoring.

No support for ensuring clarity in program state. All existing strategies reuse the default cell execution mechanism in Jupyter notebooks that could result in inconsistent state and unreproducible code [31, 42]. In the long term, these challenges lead to difficulty in understanding state and in longitudinal notebook work.

3.2 Design Goals

Based on the weaknesses above, we derived the following design goals for always-clear notebook authoring, and we explain how each goal informed the corresponding Tidynote design decisions.

DG1: Providing flexible structures for exploratory workflows. Exploratory work should be easy to initiate, localize, and organize via temporary and flexible structures, specific to a cell or general to the entire notebook, to facilitate its progression into clear work.

Tidynote introduces a scratchpad attached to the notebook, where each exploratory section is initialized by moving a specific cell into the scratchpad. Cells can also be pinned to remain visible while scrolling, supporting multi-step reasoning without losing context. These structures let users keep relevant explorations close to their originating cells.

DG2: Enabling rapid iteration between exploratory and non-exploratory activities. Cells should be easy to move between exploratory and clear portions of a notebook as it matures.

Tidynote enables bidirectional movement of cells between the main notebook and the scratchpad. This supports the natural progression and flow of notebook work, allowing explorations to be hidden or promoted without tedious manual effort.

DG3: Promoting clarity in program state. Execution should behave predictably and align with the notebook's visual layout, preventing hidden or inconsistent states.

Tidynote implements linear execution semantics and state forking aligned with the user's visual structure of notebook work. This maintains reproducible state while still allowing exploratory divergence when needed.

Guided by these decisions, we implement Tidynote as an extension to Jupyter notebook. Sec. 4 demonstrates the use of Tidynote, and Sec. 5 details its technical implementation.

4 Tidynote Walkthrough

Ali is a data scientist who also teaches introductory data analysis with Jupyter notebooks at a university. Today, she is developing a notebook that analyzes a dataset of Netflix shows as lecture material.[1] As this notebook is meant to be shared with students, Ali wants it to be clear while also exploring different ways of designing the material. As such, she decides to use Tidynote, an always-clear notebook system for Jupyter.[2]

[1]This is one of the tasks from our user study, and Ali's story and interactions with Tidynote are based on the experience of some participants

[2]Demo video in supplementary and https://tinyurl.com/tidynote-study-rep

Scratchpad. The display of Tidynote behaves like an ordinary Jupyter notebook, as shown in Fig. 3 A. Ali loads her dataset into a Pandas dataframe and computes the number of items in the dataset with `total_items = len(df)`. From this starting point, Ali would like to teach her students how to count the number of rows matching a particular condition. She needs to explore to find a suitable column and condition for the demonstration, so she creates a new cell and writes `df.columns` to list the columns in the dataset. She does not want this exploration to be part of the main narrative of this notebook. Tidynote provides a *scratchpad* in the notebook's right margin: a parallel set of cells for holding temporary, old, exploratory, or alternative computations that the user does not want to include in the main narrative. In the upper right corner of the `df.columns` cell, Ali presses the → right arrow button B which expands the right margin of the notebook to reveal the scratchpad and moves the cell into it C. Cells in the scratchpad behave as normal, but are "aside" from the main notebook narrative. A dotted line on the cell connects it to the main notebook. A scratchpad cell can reference variables defined in the main notebook before this line, but new and changed variables in the scratchpad do not affect the main notebook. The user may also hide the whole scratchpad, leaving only the main narrative visible.

Ali thinks the columns `'type'` and `'release_year'` might be useful. She clicks Jupyter's ✚ in the top tool bar that adds a cell below the active cell, in this case, in the scratchpad below `df.columns`, allowing her to continue her exploration outside the main narrative. She writes `df['type'].unique()` to discover there are only `'Movie'` and `'TV Show'` types in this data set. She adds another cell to similarly check `'release_year'` and sees many more unique values. Ali decides to go with the `'type'` column. She deletes the cell checking the unique values of `'release_year'`.

Ali will include `df['type'].unique()` in her lecture, so D she presses the ← button to move that cell from the scratchpad back to the main notebook.

Cell Pinning. She expects it may be useful to reference the column names later. To do so without needing to scroll back in the notebook, she presses the pin button 📌 on the `df.columns` cell, which causes the cell to float and stick to the top or bottom of the screen when it would otherwise scroll off the viewport (Fig. 4 E).

Sandboxed Scratch Sections. In a new cell in the main notebook, Ali writes `num_movies = len(df['type'] == 'Movie')` hoping to count the number of rows of the movie type. However, upon inspecting the variable, she realizes that she made a mistake: `num_movies` outputs the same value as `total_items`. She is confused about the mistake and presses the → button to pull out the cell to the scratchpad, expecting to potentially need several cells to debug. Testing the output of just `df['type'] == 'Movie'` shows that her code did not filter out a subset of the data based on the condition, instead just producing a list of boolean flags (Fig. 4 F). She thinks she might want to save these flags as a new column, but she does not (yet) want to mutate `df` in the main notebook or for the existing scratch cells. She creates a new cell in the notebook and moves it to the scratchpad: this creates a new "scratch section" whose state is independent of other sections, and changes to its state will not affect the main notebook. The scratchpad in Fig. 4 has three scratch sections G H I, each with its own gray dotted border. Multiple

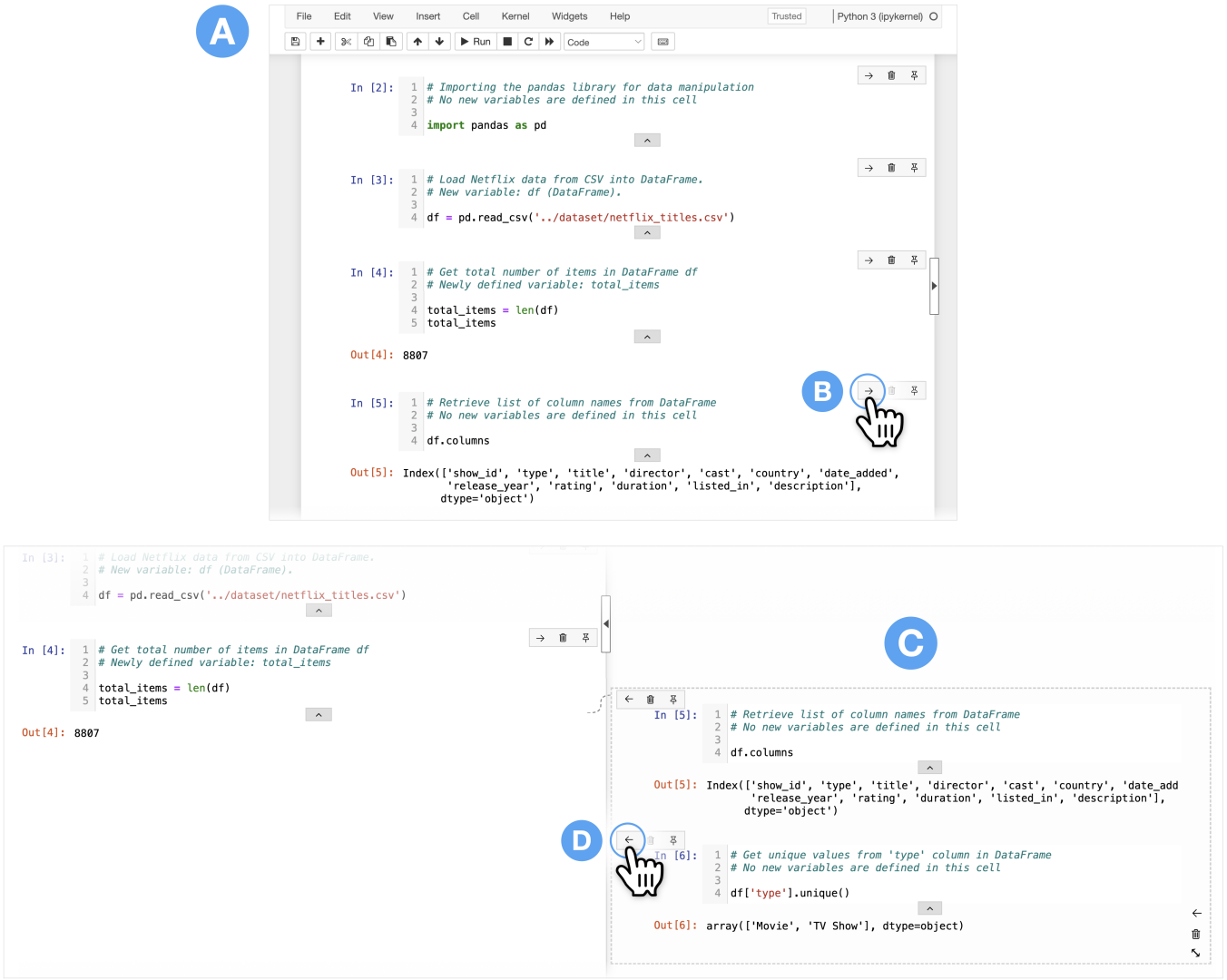

Figure 3: A The full Tidynote interface, showing Ali's initial notebook with the scratchpad hidden. B Clicking the → button expands the right margin to reveal the C scratchpad and moves the cell into it. D The ← button moves a cell back to the notebook. (Each code cell starts with two lines of automatically generated comments.)

scratch sections can branch off the same main notebook cell: as shown by the dotted lines, both H and I shared the same parent cell J, but the state of H and of I are independent of each other.

In the new scratch section I Ali adds an `isMovie` column to `df` with `df['isMovie'] = df['type'] == 'Movie'`. Ali then remembers she can count after filtering, which she tries with `len(df[df['isMovie']])`. This produces the desired result, but now she decides that creating a new column just to count may be a bit excessive. Happily, scratch sections are sandboxed so her mutated dataframe is scoped only to I: *e.g.*, re-running `df.columns` in G will not show the new column, nor will the abandoned exploration pollute the main notebook.

Ali settles on `num_movies = len(df[df['type'] == 'Movie'])`. She adds this as a new cell K in the main notebook. She repeats the computation with the `'TV Show'` type of shows. She confirms, using the scratchpad, that the sum of the two values is indeed the same as `total_items` (not shown).

Linear Execution. Ali decides the raw input data is too large to distribute to her students, so in a separate text editor she removes most of the CSV rows from the input file. In her notebook she re-runs the cell with `read_csv`. Tidynote enforces linear execution to insure integrity of the notebook narrative: all subsequent cells in the main notebook and scratchpad have their outputs grayed out to indicate they are stale and must be re-run to update (Fig. 5 L). Ali does so.

Sharing. Although most of Ali's cells are short, long cells are common. Like JupyterLab notebooks, Tidynote supports cell folding, but instead of showing the default "···" after folding, Tidynote

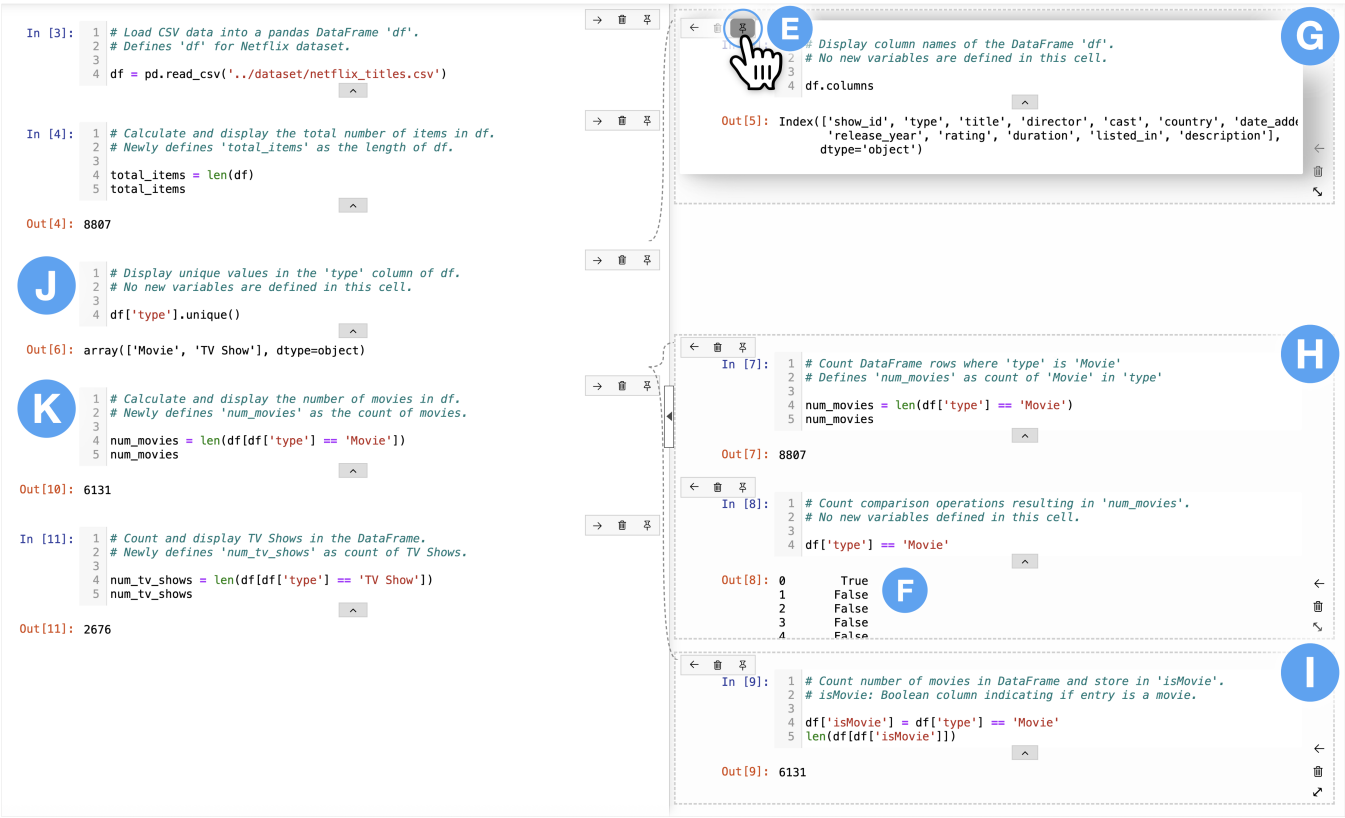

Figure 4: Cells can be pinned (E) to prevent them from scrolling offscreen. The scratchpad can contain multiple scratch sections (G) (H) (I), each with independent state from each other and from the main notebook. Multiple scratch sections might branch from the same main notebook cell, *e.g.* (H) and (I) from (J).

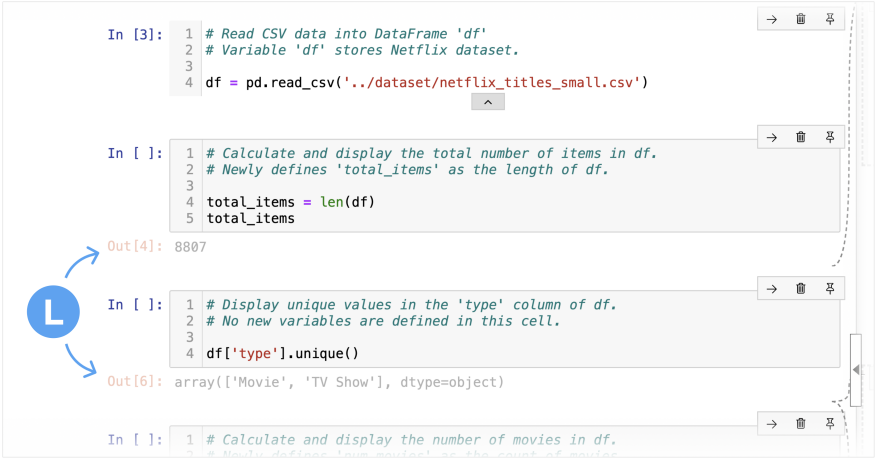

Figure 5: Rerunning the first cell (`df = ...`) is a nonlinear execution: the output of later cells are grayed out (L) to indicate staleness.

collapses the cell to only its first two lines: the first line displays an AI-generated summary of the cell's operation and the second lists any new variables defined in the cell.[3] Before sharing the notebook with her students, Ali presses collapse buttons ^ to fold the cells

[3]Implementation with GPT-4o [1] detailed in Appendix A

that import Pandas and load the CSV, as the students are already familiar with this code (collapsing not shown in figure). Finally, Ali saves the notebook as is and shares it. The students can open it in Tidynote and, by default, see only the notebook portion with Ali's explorations in the scratchpad hidden away, but they can still open the scratchpad and experiment on their own for more practice.

Summary. Ali developed a data analysis notebook for instruction using Tidynote to simultaneously explore ideas while keeping the main notebook portion for presentation clean. The primary enabler of this dual-use was Tidynote's *scratchpad*, a space that allows for side thoughts, explorations, alternative computations, and archived old ideas to separate from the notebook's main narrative. The scratchpad, along with *cell pinning* for quick reference, enabled Ali to explore and experiment with a record of her work she could consult now (with pinning) or later, without muddying her public-facing narrative (*DG1: Providing flexible structures for exploratory workflows*).

With the ability to move cells between the notebook and the scratchpad, Ali could carefully craft that public-facing narrative within her notebook without requiring an explicit cleaning phase of her notebook authoring. She could move explorations or archived code to the scratchpad and hide the mess at any time to immediately present the content to her students or co-teachers (*DG2: Enabling rapid iteration between exploratory and non-exploratory activities*).

By enforcing *linear execution*, with each *scratch section* branching off from the main narrative in a sandbox, Tidynote eliminates the overhead of reasoning about out-of-order notebook state, freeing Ali to explore fearlessly, *e.g.*, mutating the dataframe as she worked out the best way to count rows (*DG3: Promoting clarity in program state*).

Had Ali not used Tidynote, she would have spent more time deleting exploratory cells, more time scrolling to reference cells (instead of pinning them), more mental energy thinking about the kernel state, more time re-running prior cells to reset her dataframe after mutating it, and likely more time re-writing code she had previously deleted (because she could not stash it). Tidynote streamlines the process of keeping a tidy notebook.

5 Tidynote Implementation

Tidynote is an extension of Jupyter Notebooks v6 written in about 4,500 lines of TypeScript for GUI rendering and about 250 lines of Python for code execution.[4] We implement cell pinning and folding with CSS manipulation. Below we explain scratchpad rendering, linear execution, indicating stale cells, compatibility with standard Jupyter, and generalizability.

Scratchpad Rendering. The scratchpad is placed next to the main notebook and contains scratch sections attached to the main notebook, similar to text annotated with comments in a rich text editor. A scratch section must be initialized with a notebook cell moving into the scratchpad: given two consecutive notebook cells C_k and C_{k+1}, when C_{k+1} is moved to the scratchpad, it is placed in a new scratch section attached to C_k. All sections attached to C_k will be displayed in order, top to bottom, with the first aligned to the bottom of C_k. A scratch section can have one or more cells of any type: we extended cell creation to place a new cell in its parent container, be it a scratch section or the main notebook.

Linear Execution. Tidynote maintains Jupyter's cell-based interaction model, where cells are executed individually, but enforces a top-to-bottom execution order. Regardless of a cell c's location or execution history, Tidynote ensures that executing c yields a state consistent with the notebook's linear structure. To achieve this, when executing c, Tidynote clears the kernel's global namespace (via `%reset -f`) to remove effects from prior executions, computes c's *prefix* (defined below), and evaluates the code sequence *prefix*+c as the output for c.

We compute the *prefix* of c depending on its location: (1) if c is in the notebook, *prefix* is the code of all executed cells visually above c concatenated in their visual ordering; (2) if c is in the scratchpad, then c must be in a scratch section s attached to a notebook cell c_{nb}, and so *prefix* is the concatenation of all executed notebook cells from the beginning of the notebook through c_{nb}, and all executed scratch cells in s above c. Finally, with a kernel already reset, we run the code of c prefixed by *prefix*. The above computation for *prefix* along with kernel resetting means that running a cell in the main notebook will *not* involve any code in the scratchpad, effectively sandboxing scratchpad execution from the notebook; similarly, running a cell in a given scratchpad section will *not* involve any code from other scratchpad sections. This full-rerun approach is similar to the "No Checkpoint" approach in Multiverse [38]. It involves no process forking and has reasonable performance (potentially faster than the serialization approach in Fork It [46]; see Table 6 in Multiverse).

This rerun approach could cause code with side effects (*e.g.*, printing) to repeat. For example, if a cell c contains a `print` statement, and we run c with the prefix *prefix* that contains another `print` statement, we get two print outputs on the cell instead of one; in regular Jupyter, we would have gotten only one output from c. As a limited workaround, we modify *prefix* as follows to avoid redundant output: (1) we remove calls that display data to `stdout`, including `print` statements, plotting calls (*e.g.*, `plt.show()`), and dataframe summary calls (*e.g.*, `df.info()`); (2) we turn off `matplotlib`'s interactive mode and enforce explicit calls of `.show()` for plot display.

Note that our implementation of linear execution is less strict than the regular linear execution (in scripts and notebook systems like Polynote [3]): we allow users to *skip* cells, *i.e.*, running a cell with one or more cells above it unrun.

Visualizing Stale Cells. Recall that the cell-based interaction model allows flexible execution orders, both (a) *linearly*: running a new, previously-unrun cell at the end of the notebook or a scratchpad section with every cell above it already run in sequential order, and (b) *nonlinearly*: everything else, including moving a cell from the scratchpad into the notebook. In case of a nonlinear execution order upon cell c, all cells that come after (in the notebook or scratchpad) become *stale*, and Tidynote informs the user as follows: (1) clearing the execution results of c (if any) and all cells occurring later in the notebook as well as their attached scratch sections, (2) regenerating results for c, and (3) graying out the later cells (in the notebook and scratchpad) to indicate their staleness.

Compatibility with Standard Jupyter. Tidynote GUI information is written to the notebook metadata (JSON blob), including cell

[4] Tidynote source and examples: https://github.com/rlisahuang/tidynote-artifact

state (executed/folded/placement), code, scratchpad state (open/hidden), and scratch sections. When a Tidynote notebook is opened in standard Jupyter, the main notebook cells come before the scratchpad cells, and changes to a cell's content will not affect its placement (notebook/scratchpad) when re-opening the notebook in Tidynote.

Generalizability and Limitations. Tidynote implements always-clear notebook authoring on top of the classic Jupyter notebook (v6), but the implementation should extend to any computational notebook system that allows modifications to its GUI and the execution model. However, there are several limitations of our approach that we leave for future work. First, our implementation of linear execution is incomplete as we only manually removed common calls that produce redundant output to `stdout`. Second, the implementation of linear execution could be further improved with a more efficient, time traveling kernel [38] or caching large computation to the disk [13].

6 Exploratory User Study

We ran an IRB-approved exploratory user study to answer the following research questions:

RQ1. How do Tidynote features support maintaining clarity throughout notebook authoring?

RQ2. How does Tidynote support realistic notebook tasks?

RQ3. What strategies for notebook clarity does Tidynote enable?

The full study materials are available in the supplements and at https://tinyurl.com/tidynote-study-rep.

Participants. We recruited 13 participants who were proficient with Jupyter notebooks via social media and emails. Table 1 summarizes their demographics, frequency of Jupyter use, and attitude towards notebook clarity prior to the study.

Tasks. Since Tidynote aims to promote notebook clarity *during* the development of notebooks, we choose to evaluate it with data analysis tasks, as they are often exploratory and prone to the accumulation of messes. We provide a starter notebook for each participant. The starter notebook imports one of the following datasets, both popular on Kaggle: "Netflix Movies and TV Shows"[5] or "Most Streamed Spotify Songs 2023"[6]. The notebook continues with a tutorial of Tidynote in the context of data exploration. Following the tutorial is a question about the given dataset to be answered with programming. After the initial question, participants come up with their own question(s) about the dataset that they write code to answer; if they could not come up with any questions, we provide a list of questions for them to select. As such, the tasks are open-ended: we do not restrict the answer to the initial question (*e.g.*, participants could answer it with a plot, print statements, or something else, with or without library use) or the follow-up questions to resemble realistic data analysis work and to encourage in-depth usage of Tidynote.

Procedure. Each study took about 90 minutes on Zoom with informed consent. Each participant was randomly assigned to one of the two starter notebooks (size of assignments balanced) and accessed the notebook in the browser. First, participants went through a Tidynote tutorial and practiced Tidynote features to familiarize with both the data and the environment. Participants then spent up to 40 minutes (or until 20 minutes before the end of the study, whichever came first) writing code to answer the initial question provided and (if time permitted) their own questions about the dataset, with the ability to use Google search and AI assistants such as ChatGPT. Participants were expected to keep the notebook as clear as possible, to the extent that they would be comfortable with continuing the analysis if leaving the notebook for weeks or sharing the work with collaborators. For the remaining 20 minutes, participants went through a semi-structured interview to reflect on their experience.

Data Collection. From the pre-study screening survey, we collected participants' existing attitudes towards notebook clarity. Throughout the study, we logged notebook actions (through Tidynote) and noted clarity-relevant behavior and quotes; the first author revisited notebook recordings to validate action logging. Finally, the first author conducted open coding [6] of the types of participants' own tasks from the final notebooks and field notes, as well as the post-study interviews (auto-transcribed by the conferencing software).

To measure how Tidynote helped maintain notebook clarity during realistic tasks, we collected the final notebooks created by the participants and evaluated the importance of each Python code statement (*e.g.*, assignment, function definition, import) to their analysis tasks. Though there is no standard rubric for such evaluation, following practitioners' definitions for notebook cleaning of "keeping a desired subset of results while discarding the rest" [14], the first two authors created three categories for the code statements in the final notebooks: (1) *Relevant* if directly generating results for the analysis tasks; (2) *Necessary* if not relevant but syntactically necessary for the code to run (*e.g.*, function definitions and library imports); (3) *Transient* if neither relevant nor necessary, such as one-off exploration. The first author reviewed all final notebooks to categorize all statements after the tutorial portion. Table 2 reports the results of this analysis, which we discuss in Sec. 7.2.

Study Limitations. First, although we recruited participants from various backgrounds, there were only 13 participants, most of whom worked in computing-relevant fields with self-reported Jupyter experience. Second, although participants worked on their own analyses of the given data that could improve external validity, our findings were limited to data-driven notebook work and the two datasets involved. Third, there was no explicit comparison between Tidynote and a baseline system, as we decided from the comparative analysis (Sec. 3.1) that no prior system was intended for maintaining notebook clarity throughout the authoring process, *i.e.*, the goal of Tidynote. To mitigate this, because participants were familiar with the regular Jupyter, we asked them to retroactively compare their Tidynote experience to prior Jupyter experience in the post-study interview. Finally, our study required participants to keep their notebooks as clear as possible to stress-test Tidynote features as opposed to assessing its most natural use.

As such, our study findings should be viewed as an early step towards understanding the effects of always-clear notebook authoring and generalizations to notebook systems beyond Jupyter.

[5] https://www.kaggle.com/datasets/shivamb/netflix-shows/data
[6] https://www.kaggle.com/datasets/nelgiriyewithana/top-spotify-songs-2023/data

Table 1: Left: Participant demographics, occupation, Jupyter use, and attitude towards notebook clarity prior to the study ("Reluctant"=would reluctantly clean a notebook, "Glad"=would gladly clean a notebook, "Always"=would always keep a notebook clear. Right: Number of tasks beyond the provided task performed during the study, and task types.

	Occupation	Jupyter Use	Clarity Preference	Num "Other" Tasks and Task Types
P1 (M)	PhD student, systems	Daily	Reluctant	5: wrangling (2); summarizing (2); plotting (1)
P2 (M)	PhD student, AI/ML	Daily	Reluctant	1: modeling (1)
P3 (M)	PhD student, security	Daily	Always	2: plotting (2)
P4 (M)	Software developer	Daily	Always	0: N/A
P5 (M)	Professor, Chemistry	Daily	Reluctant	1: wrangling (1)
P6 (M)	ML engineer, finance	Daily	Glad	4: summarizing (4)
P7 (M)	ML engineer	Daily	Always	7: summarizing (6); filtering (1)
P8 (M)	PhD student, IoT	Daily	Reluctant	1: plotting (1)
P9 (F)	Student, data science	Regularly	Always	1: wrangling (1)
P10 (F)	Student, AI/ML	Regularly	Always	1: plotting (1)
P11 (F)	Researcher, CS education	Occasionally	Reluctant	2: language processing (1); filtering (1)
P12 (F)	Student, computer science	Occasionally	Always	3: summarizing (1); plotting (2)
P13 (F)	Hydrologist, geoscience	Regularly	Glad	2: summarizing (1); wrangling and plotting (1)

7 Results

Below we report the user study results according to our three RQs in terms of behavioral data, quotes, and comparison to regular Jupyter.

7.1 RQ1: How do Tidynote features support maintaining clarity throughout notebook authoring?

We describe how participants in our study used various features of Tidynote that achieve our three Design Goals (DGs) to maintain notebook clarity. First, we report the use of features that implement *DG1: Providing flexible structures for exploratory workflows.*

[DG1] All participants alternated notebook and scratchpad work to structure exploration. In Tidynote, one could *actively work* in the notebook or the scratchpad by creating, running, or (vertically) moving cells. We identified *container switch* events—when the locations of two consecutive actions differ—and computed the proportions of time each participant actively worked in the notebook/scratchpad. The colored blocks in Fig. 6 show the data, participants are ordered from top to bottom by the percentage of active work in the scratchpad. The figure shows that all participants actively worked in both containers, and that they all alternated their active work between the containers throughout the tasks. In addition to alternating between containers, four participants (P1, P3, P7, P11) further alternated between at least two scratch sections in the scratchpad, using the additional sections for reference (P1, P7), one-off explorations (P3, P11), reminders for future tasks (P3), and archival of outdated or irrelevant code (P7).

[DG1] Most participants pinned at least one cell to minimize scrolling during exploration. 11 out of 13 participants (all but P1, P2) used cell pinning to keep inspection results or reference material visible as they sifted through long notebooks. During the post-study interview, P6 further showed a personal notebook to suggest pinning non-code cells like markdown checklists, something the study did not demonstrate but Tidynote already supports. Two participants (P1, P2) did not pin any cell, either feeling the lack of need with all the work within the viewport (P2) or having forgotten about it (P1). P1 recognized this as a *"user error"*, but suggested *"automatically pinning [in] the scratchpad"* as he had to scroll repeatedly through the scratchpad during the study.

Below is the use of features relevant to *DG2: Enabling rapid iteration between exploratory and non-exploratory activities.*

[DG2] All participants moved cells between the notebook and scratchpad throughout the study as exploration progressed into clarity. Fig. 6 also visualizes the cell movement events as triangles in their temporal order, showing that all participants moved cells at least once throughout their tasks. The scratchpad is designed to be used in conjunction with the notebook: one can create new cells in existing scratch sections, but can only create a new scratch section by moving a notebook cell over. Even with this requirement, all participants except P12 moved cells in both directions. Most participants except for P8 found moving cells intuitive. P8 initially struggled with restoring the ordering of scratchpad cells in the notebook, but later realized that moving an entire scratchpad section was meant for moving a group of cells. Indeed, six participants (P1, P2, P3, P4, P8, P13) moved entire scratchpad sections to the notebook when they wanted to include entire sections of scratch work in the main notebook narrative.

[DG2] Some participants toggled the scratchpad to focus on the notebook or confirm notebook clarity. Eight participants (P1, P2, P3, P7, P8, P10, P11, P13) toggled the scratchpad at least once during the study. Most of them hid the scratchpad to temporarily focus on the notebook (P1, P2, P3, P7, P10, P11, P13), and three participants (P8, P10, P11) explicitly closed the scratchpad at the end of all the tasks to read through the notebook and ensure it was clear and presentable. Among the participants who did not toggle the scratchpad, they either found it unnecessary to hide it for purposes other than presentation (P5, P6, P9) or they misinterpreted the button as deleting the scratchpad (P4, P12); P12 *"would definitely use that if [she] knew that next time."*

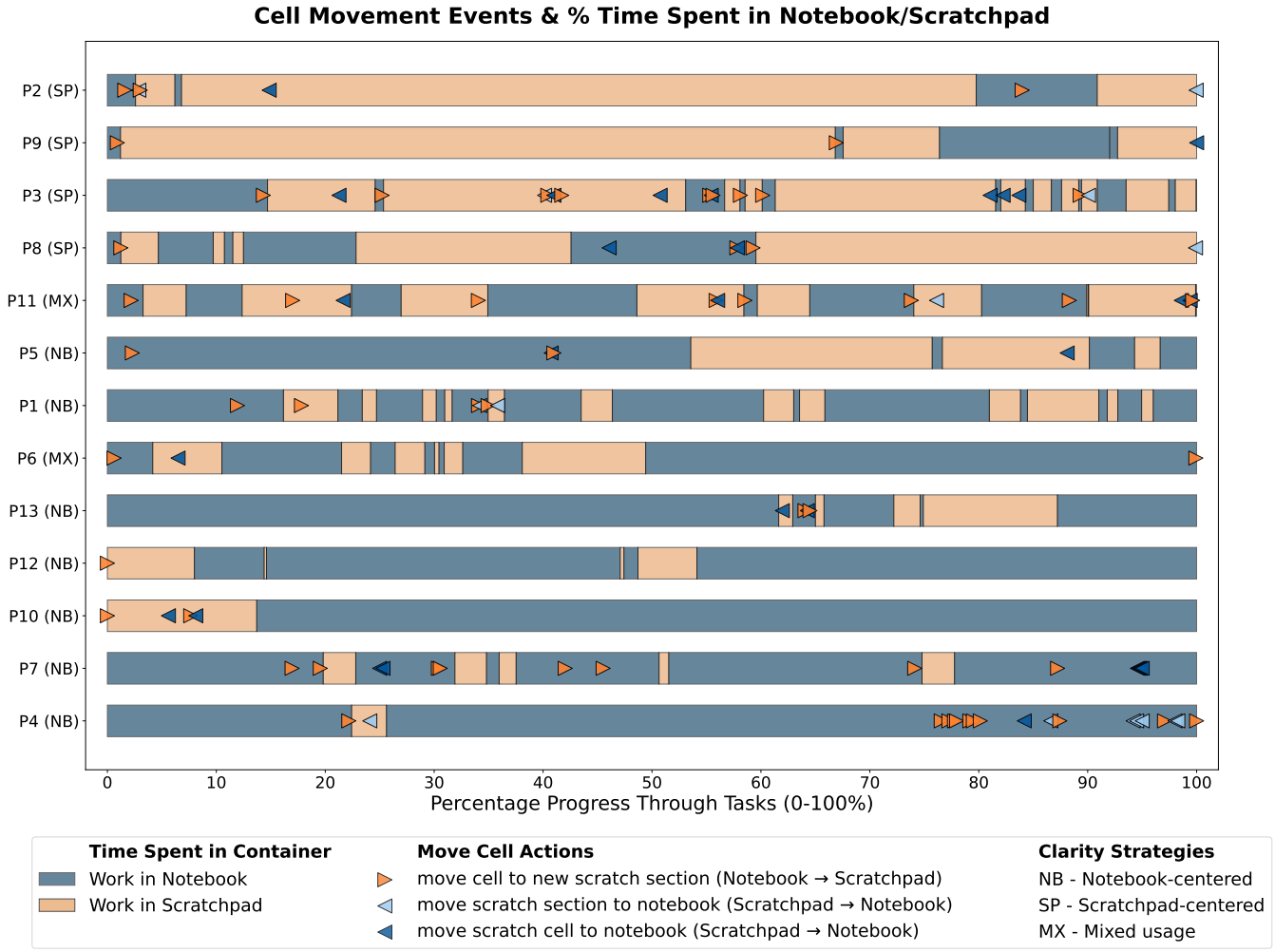

Figure 6: Cell movement events & percentage time spent in notebook/scratchpad, ordered by % time spent in scratchpad from top to bottom. "Time through tasks" is the interval between the onset of the first notebook action after the tutorial, and the ending of the last action before the end of the tasks. Time progress is normalized across participants, labeled by their strategies for clarity (Sec. 7.3).

Finally, we report on the use of linear execution for *DG3: Promoting clarity in program state.*

[DG3] All participants ran cells out of order without experiencing confusion with state. Tidynote enforces linear execution, a big deviation from the regular Jupyter semantics that allow execution in any order. We analyzed how linearity affected (a) participants' execution behavior and (b) state clarity. For (a), we recorded three types of cell execution: (1) "linear"—running a new, previously unrun cell at the end of the notebook or a scratchpad section with all cells above it already run in sequence (2) "last cell"—rerunning the cell at the end of the notebook or a scratchpad section (cells above it already run in sequence) (3) "nonlinear"—every run that does not fall under the previous categories. Fig. 7 visualizes the distribution of these actions, showing that out-of-order execution ("nonlinear" and "last cell") remained prevalent regardless of where the execution occurred (notebook or scratchpad). For (b), we measured how participants used Jupyter's "restart and run all" feature, a common technique to restore state clarity by restarting the kernel and running all cells [15]. We found that *no participant* ever had to use this feature during the study to resolve state confusion; only P2 did so intentionally to confirm his notebook was reproducible at the end of the tasks. In addition, although we did not ask explicitly, no participant mentioned confusion about state, and no participant had to manually trace/debug the state. Finally, we reran copies of the 13 notebooks used in the study as a sanity check, finding that all of them reproduced the same results as seen in the study.

Comparison to Regular Jupyter. Overall, participants recognized that Tidynote features work together to improve the experience of flexible exploration and maintaining clear notebooks throughout the notebook authoring workflow. As P7 noted, *"I don't think*

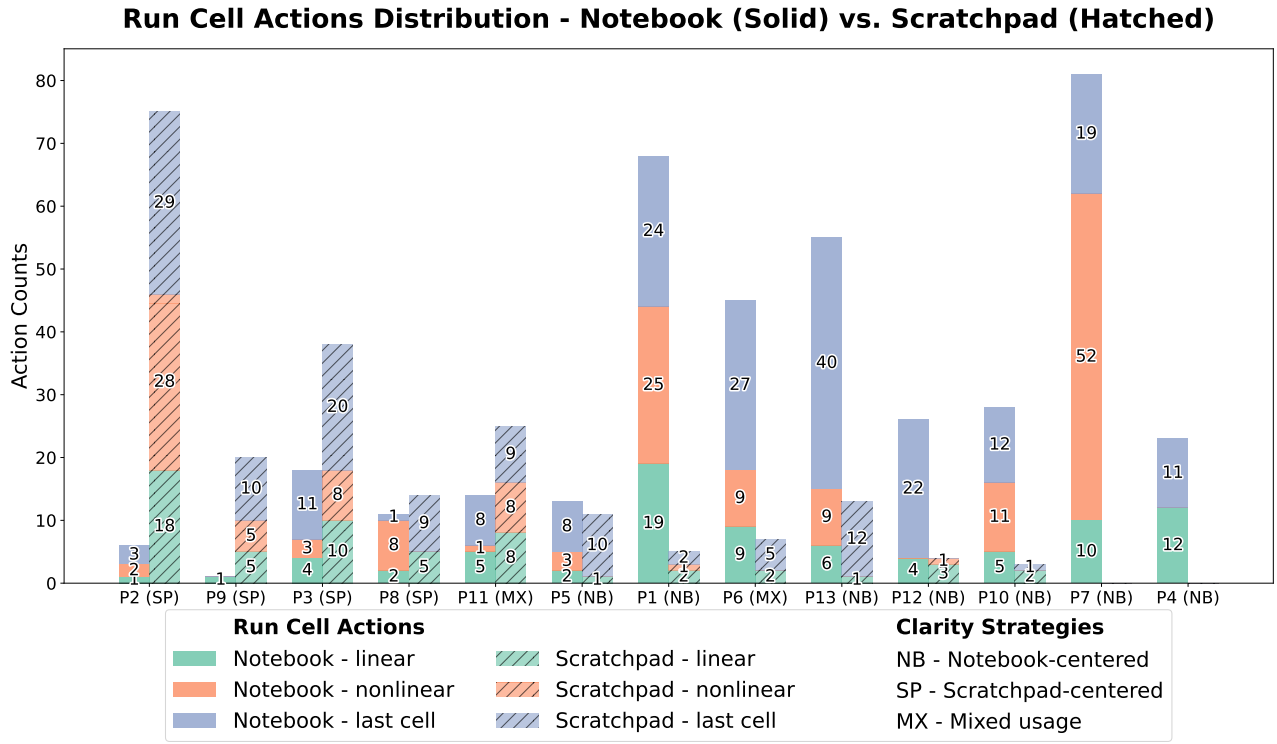

Figure 7: Distribution of run cell actions by container. "linear" = running a new, previously unrun cell at the end of the notebook or a scratchpad section with all cells above it already run in sequence, "last cell" = rerunning the (previously-run) cell at the end of the notebook or a scratchpad section (cells above it already run in sequence), "nonlinear" = everything else. Participants are labeled with their strategies for clarity (Sec. 7.3) and ordered the same as Fig. 6.

any of these features are unnecessary—it just depends on which use case you're working on." Participants commented on the following improvements compared to regular Jupyter:

- **Scratchpad and movable cells encourage exploration without compromising notebook clarity**. The scratchpad enables temporary archival of prior exploration or messy code that could be useful in the future (P1, P3, P4, P7, P8), which if created in the notebook can be easily moved out (or back) to keep the notebook clear. In addition, the scratchpad separates exploration and clutter (P2, P3, P5, P9, P10, P11, P12, P13), supports comparative data exploration (P2, P3, P9, P12, P13), and eases hypothesis testing (P5, P11) such as validating AI-generated results before incorporating them (P11). All of the above are not available in regular Jupyter.
- **Cell pinning eases navigation during exploration**. P10, who pinned the most cells among all participants (five), explicitly called out its benefit for avoiding scrolling, while she *"would have had more redundant cells to avoid the scrolling"* in regular Jupyter. Scrolling is indeed the activity with the most time spent in regular Jupyter [15], and as P3 put it, *"I've died so many times scrolling around my notebook and trying to find [something]. [Pinning] is perfect."* Six participants (P3, P6, P7, P8, P10, P12) highlighted how pinning reduced the need to repeat inspection code like `df.head()` at multiple points.
- **Linear execution ensures clarity of program flow** by maintaining a clear and consistent program state, which was especially helpful for workflows like machine learning. As P6 demonstrated in his personal Jupyter notebook at several places, forgetting to re-run key steps caused errors. In addition, 10 participants (P1, P3, P4, P5, P6, P7, P10, P11, P12, P13) thought that linearity helped avoid confusion and encourage regular testing, and consequently led to less error-prone notebooks.
- **Tidynote can enhance teaching with notebooks**, an opportunity recognized by two participants with relevant teaching experience (P5, P11). They envisioned live coding [39] or peer instruction [7] scenarios, where both the instructor and the students could experiment in the notebook while keeping it *"nicely structured"* (P5). P11 liked how Tidynote supports one-off explorations without compromising notebook clarity: she taught in scenarios where students sometimes did not have Internet access and could not explore relevant concepts on their own, so they relied on her to demonstrate the "what-if" scenarios in the notebook used for lecture.

Still, some participants felt Tidynote features did not change or even worsened certain scenarios that did not require clarity, such as using a notebook entirely for scratch work (P8, P9) and small tasks that might eventually be scripted (P9). For these contexts,

Table 2: Evaluation of the final notebooks created by participants in the study. Each code statement was categorized as: (1) *Relevant* (R) if directly generating results for the analysis tasks; (2) *Necessary* (N) if not relevant but syntactically necessary for the code to run; (3) *Transient* (T) if neither relevant nor necessary, such as one-off exploration.

	Notebook			Scratchpad		
ID	**R**	**N**	**T**	**R**	**N**	**T**
P1	10	10	2	0	0	1
P2	25	44	0	0	0	0
P3	12	14	1	0	0	0
P4	6	8	0	0	0	3
P5	2	2	0	0	0	1
P6	6	2	1	0	0	2
P7	9	4	2	0	0	0
P8	6	9	1	0	0	0
P9	2	5	0	0	0	1
P10	9	14	0	0	0	0
P11	6	10	1	0	0	10
P12	4	3	0	0	0	0
P13	15	11	3	0	0	5
Min.	*2*	*2*	*0*	*0*	*0*	*0*
Mean	*8.6*	*10.5*	*0.8*	*0.0*	*0.0*	*1.8*
Max.	*25*	*44*	*3*	*0*	*0*	*10*

the enforced linearity limits the ability to explore freely (P8, P9) and compromises performance when repeatedly importing large datasets (P8), and the scratchpad could have been just a sandbox independent from the notebook as opposed to having sections attached to different places (P9).

7.2 RQ2: How does Tidynote support realistic notebook tasks?

Table 1 (column "Num Other Tasks and Task Types") shows that, during the study, Tidynote supported participants in six types of data work: wrangling (manipulating data in place), summarizing (using library functions to summarize data without manipulation), plotting (using libraries to plot data), modeling (creating models to predict unseen data based on the given data), filtering (obtaining a subset of data based on conditions), and language processing (analyzing patterns in natural language data). All participants felt that their tasks during the study resembled their usual notebook tasks. In addition to the tasks performed during the study, several participants imagined using Tidynote to teach intro programming (P5, P11) and document longitudinal findings (P3, P8, P13).

Our analysis of the code statements in the participants' final notebooks (Table 2) further suggests that Tidynote helped participants maintain clear notebooks. On average, the notebook portion had 8.6 *Relevant* statements (assignments, function definitions, etc.) and 10.5 *Necessary* statements, compared to only 0.8 *Transient* statements. In contrast, the scratchpad portion had zero *Relevant* and *Necessary* statements, and 1.8 *Transient* statements, across all participants. This indicates that the *Transient* code that did not get deleted along the way mostly ended up in the scratchpad, suggesting that the resulting notebook portion could be considered more narrative focused and thus more "tidy," while the scratchpad contained extraneous computation. Moreover, six participants had no code left in the scratchpad in their final notebooks, either deleting them all (P2, P3, P8, P12) or moving them back into the notebook (P7, P10) before the end of their tasks. The notebook and scratchpad separation may have assisted this final cleaning: had those cells been in the notebook, these participants would have had a harder time deciding which cells were deletable or movable.

Comparison to Regular Jupyter. In addition to benefiting from the Tidynote features, most participants appreciated the ability to reuse familiar Jupyter workflows in Tidynote, which made it learnable and adaptable to regular notebook tasks. Specifically, while participants felt *what* they did in the study resembled their usual work, when asked to compare *how* they did the tasks during the study with their usual Jupyter workflow, 12 participants (all except P9) felt that their workflows with Tidynote were similar to their usual workflows. Indeed, in addition to the Tidynote features aimed for always-clear authoring, participants had access to all standard Jupyter features in Tidynote. P4 particularly called out that Tidynote's tight integration into Jupyter eased adjusting to the new interaction paradigm because most prior Jupyter usage patterns remained available.

7.3 RQ3: What strategies for notebook clarity does Tidynote enable?

To understand how participants used Tidynote to maintain clarity throughout notebook authoring, we revisited their notebook activities from the video recordings and instrumentation data. The analysis showed two main strategies: (1) a *notebook-centered strategy*, if the user keeps their work within the main notebook in Tidynote and wraps up a task by deleting the messy cells or moving them to the scratchpad; and (2) a *scratchpad-centered strategy*, if the user keeps their work within the scratchpad and only moves it back into the main notebook when wrapping up.

Notebook-centered. Seven participants (P1, P4, P5, P7, P10, P12, P13) adopted the notebook-centered strategy to keep their notebook clear. Instrumentation data shows that all these participants spent a greater portion of their time in the notebook than in the scratchpad, creating and executing more cells in the notebook, as is illustrated in both Fig. 6 (more blocks in blue) and Fig. 7 (higher solid bars). Notably, P4 and P7 never ran cells in the scratchpad (no corresponding hatched bars in Fig. 7), only moving cells back and forth as needed (Fig. 6).

Scratchpad-centered. Four participants (P2, P3, P8, P9) used the scratchpad-centered strategy, spending more time creating and running cells in the scratchpad (top four rows with more orange blocks in Fig. 6, and higher hatched bars in Fig. 7). Two participants (P2, P9) *cherry-picked* their work from the scratchpad: P2 manually copied and pasted code line by line into the main notebook; P9 cherry-picked similarly but in a new cell within a scratch section before eventually moving the cell back into the main notebook (few blue triangles for P2 and P9 in Fig. 6, indicating fewer cell movements). P3 did not cherry-pick by copy-paste but instead directly moved individual cells and scratch sections back into the

notebook (more blue triangles than P2, P9, and P8 in Fig. 6). P8 adopted the P2's cherry-picking strategy in the initial task and P3's strategy of direct cell movement in his own task.

Mixed usage. Two participants (P6, P11) used a mix of both strategies. P6 started his work with the scratchpad-centered strategy for the initial task but eventually leaned toward the notebook-centered way (more orange blocks in the beginning than later in Fig. 6); as he worked on four additional tasks (Table 1), he ran more cells in the main notebook (higher solid bar in Fig. 7). P11, in contrast, started her work in a notebook-centered way but gradually moved data exploration and hypothesis testing into the scratchpad before moving the concluding results back into the notebook, and she indeed ran cells more in the scratchpad (higher hatched bar in Fig. 7).

Comparison to Regular Jupyter. In the interview, we asked all participants how Tidynote affected their attitudes towards notebook clarity. With the above Tidynote-inspired strategies for notebook clarity, *all* participants who were previously unmotivated about clarity (Table 1: P1, P2, P5, P6, P8, P11, P13) felt that clarity could be easily achieved in Tidynote within their preferred strategy. Because Tidynote *"doesn't really introduce that much overhead [of cleaning] in [one's] workflow"* (P1), P2's workflow was completely changed: *"Right now, I would clean my notebook. Before, I would not clean my notebook."* Before with regular Jupyter, participants delayed cleanup until necessary (P6) with manual post-hoc cleaning (P6, P10) or help from AI assistants (P1, P2), or simply ignored notebook clarity (P1, P2, P11). Meanwhile, those already committed to notebook clarity (P3, P4, P7, P9, P10, P12) maintained the attitude after using Tidynote, although P9 would rather not use Tidynote for clarity out of her distaste for enforced linearity. Finally, there are cleaning workflows that participants would reuse from prior Jupyter experience, such as adding manual documentation and markdown headings (P3, P13).

8 Discussion and Future Work

Based on our study results, below we discuss how Tidynote meets its design intention (Sec. 8.1), implications for clarity workflows in regular Jupyter (Sec. 8.2), design opportunities for future notebook clarity support (Sec. 8.3-Sec. 8.4), and design implications for future information systems (Sec. 8.5).

8.1 Always-Clear Authoring: Does Tidynote support it?

Our main goal is to support always-clear notebook authoring, by meeting our three Design Goals to better manage exploration (DG1), enable rapid iteration between exploratory and non-exploratory activities (DG2), and promote clarity in program state (DG3). In Tidynote, we proposed the scratchpad and cell pinning to support DG1, cells movable between the notebook and scratchpad to support DG2, and linear execution with state branching in the scratchpad to support DG3. Our study showed that participants used all these features interchangeably to maintain clarity throughout a notebook's lifecycle (Sec. 7.1) for realistic notebook tasks (Sec. 7.2), and developed strategies that suited their workflows for maintaining clarity (Sec. 7.3). These results highlighted that Tidynote is a promising step towards supporting always-clear authoring.

There are two possible explanations for the generally positive findings. First, we grounded the design in empirical findings about notebook use and a thorough comparative analysis of existing tools. Second, instead of building a brand new system, we implemented an augmentation of Jupyter largely compatible with existing features. Although a further discussion on interface learnability is beyond the scope of this work, prior work assessing the learnability of program synthesizers [17] highlighted the importance of supporting pre-existing experience. As a Jupyter extension, Tidynote allows participants to leverage pre-existing knowledge about Jupyter notebooks (Sec. 7.2). And Tidynote's linear execution, though different from Jupyter, is close to the regular linear execution in scripting, where code is executed top to bottom.

8.2 Generalizability of Findings

To stress-test Tidynote features for maintaining notebook clarity, our study required participants to keep their notebooks as clear as possible. This may differ from notebook authoring in the real world where clarity may not be enforced. Our results suggested that, once leaving the Tidynote bubble, participants would likely resume their prior clarity habits: those reluctant about maintaining clarity (with regular Jupyter) only felt more motivated about clarity during the study because Tidynote eased achieving the goal (Sec. 7.3).

Still, the study results and Tidynote remain relevant to practical notebook authoring. A meaningful portion of notebook users genuinely value clarity but have to rely on manual tactics to meet their goals in regular Jupyter. Six out of 13 participants in our study always kept clear notebooks in their ordinary work (Table 1). Moreover, an earlier study found that 17 out of 20 participants adopted tactics to clean their notebooks, and some of these participants cleaned continuously [15]. But, cleaning is only indirectly supported in ordinary Jupyter, so many of the user tactics have more of the character of "workarounds" rather than "using Jupyter features as intended". Table 3 shows several of these previously reported manual tactics for clarity [15], as well the corresponding features in Tidynote that streamline or subsume the tactic. For example, during the study P11 leveraged scratchpad's support for the debugging (third tactic in Table 3), which inspired part of our walkthrough example in Sec. 4. The tactic "merging cells relevant to one task" (fourth tactic in Table 3) comes at the cost of losing intermediate cell outputs [15], which scratchpad sections avoid by allowing relevant cells to be in a section. Abstracting code into a function to avoid polluting the global state (last tactic in Table 3), in addition to requiring a tedious refactor, also sacrifices the ability to display intermediate outputs within the function for exploration and debugging; in comparison, Tidynote's linear execution keeps the global state cleaner, potentially delaying the need to abstract code into a function.

8.3 A Lack of Standard for Clarity

Our study also revealed an interesting contrast between some participants' own feature preferences and the resultant clarity of their notebooks. A notable example is the difference between P2 and P9: P9 disliked linearity but favored the scratchpad, while P2 was the opposite. Despite the contrasting feature preferences and resultant differences in Tidynote usage, in the post-study interview, they

Table 3: User Tactics for Notebook Clarity Supported by Tidynote

Clarity Requirement	Manual Tactics in Jupyter [15]	Tidynote Support
Content	Removing/commenting out redundant code and cells	Scratchpad
	Using empty cells to separate groups of cells	
	Debugging in a fresh notebook to keep the original intact	
	Merging cells relevant to one task	Scratch sections
State	Using abstractions (*e.g.*, function definitions)	Linear execution

both felt highly confident with their notebook clarity, and their notebooks (the notebook portion) contained entirely *Relevant* and *Necessary* code, according to our evaluation (Table 2). These results suggest that the interpretation of "clarity" could be subjective: *e.g.*, clarity in state was important to P2 but not P9.

Indeed, there is no standard of what makes a notebook "clear". Notebooks lack something similar to software development guidelines despite empirical results on how notebook users maintain clarity [15, 46]. We observed varying standards for clarity among participants: P4 thought that code readability was the top criterion for notebook clarity; P8 had a vague concept for clarity as he typically did not keep a "clear" notebook; others (*e.g.*, P3 and P13) valued the ability to maintain a cohesive narrative more. Tidynote does not directly account for the varying standards for clarity, although it still caters to different needs by providing multiple features aimed for different aspects of clarity (*e.g.*, content vs. state), as illustrated by the example of P2 and P9 above. Given the variety of views and strategies for notebook clarity (Sec. 7.3), we believe that clarity should be judged relative to the user's task and preference, and future always-clear notebook authoring systems should provide support for multiple aspects of clarity, adequate for the user's desired level and preference.

8.4 Handling the Happy (and Sad) Side Effects of Clarity

Clarity is one quality attribute among many others that users value in their notebooks [15], notable others including correctness and reproducibility. Although not our design intention, by promoting clarity in the program state (mainly out of concerns for user comprehension), Tidynote led to the happy side effects of supporting correctness and reproducibility—our sanity check on the 13 notebooks used in the study showed that they all reproduced the same results after the study (Sec. 7.1).

Maintaining clarity still comes with sad side effects. First, although our main goal was to alleviate the well-known tension between clarity and explorability [15, 21, 26, 33, 37], the tension remains in situations where clarity is not the main goal. For example, clarity need not be maintained in using notebooks entirely for scratch work (P8, P9) or small tasks that might eventually be scripted (P9). Second, there are also situations where other attributes like debuggability and efficiency take precedence over clarity. In these scenarios, users might prefer nonlinear execution for quick debugging and avoiding re-processing large data (P2, P8, P9), where the enforced linearity can become restrictive.

Perhaps the solution is to, *e.g.*, offer a toggle back to ordinary Jupyter execution semantics, and to incorporate other adjustable clarity support suggested by participants:

(1) Better visual layout of the scratchpad: When one has a lot of scratch work, having better auto-layout algorithms (P1, P7, P10) (*e.g.*, like Enso [2]) could help better organize the scratchpad;
(2) Placing pinned cells and scratch sections more freely depending on the notebook task: For more exploratory work, scratch sections need not be attached to any part of a notebook, so they could instead become resizable and floating windows that can be placed anywhere on the screen (P2, P5, P9, P10, P12) just like in StickyLand [45];
(3) More integrated AI support that facilitates notebook clarity, particularly in documentation, such as documenting multiple cells (P7), deriving hypotheses from code (P2, P9), summarizing output (P2) and errors (P12), refactoring scratch work into a notebook cell (P3), and code generation (P6).

Furthermore, future notebook systems could incorporate a "slider" that corresponds to a spectrum of support for clarity, with the most relaxing that optimizes for rapid exploration to the most restricted that prioritizes notebook interpretability, depending on user preferences and task requirements.

8.5 Broader Design Implications for Information Systems

Our system and study results can shed light on future information systems that need to balance exploration and clarity, such as word processors and presentation programs. Specifically, our three design goals directly inform the following general design goals for such information systems: **providing structural flexibility**, **enabling both ephemeral and evolving ideation**, and **aligning visual layout with internal logic**.

Our study further revealed opportunities for **supporting on-the-fly information sharing**. Specifically for notebook sharing, this entails teaching, collaborative editing, and output-driven sharing scenarios that require little overhead in cleaning. For teaching, Tidynote already supports effort-free notebook cleaning, a big alleviation for instructors (P5, P11) who often have little time for cleaning up notebooks that become messy during lectures but then need to be shared with the class. For collaborative editing, there could be better support for documentation, such as headings for scratch sections, similar to Janus [36] but automated like Themisto [43], to facilitate understanding the scratchpad explorations from one other. Finally, when clarity matters less for the code but more for the output (P6), we could incorporate functions such as output-only dashboards [45] or high-level notebook summaries. The above notebook sharing scenarios apply to general information systems and can benefit from similar support for always-clear authoring.

9 Conclusion

We proposed always-clear notebook authoring, a paradigm that that supports both clarity and exploration throughout the entire notebook lifecycle, instantiated in a prototype for Jupyter called Tidynote. An exploratory study (N=13) showed that Tidynote features collectively improved the experience of maintaining clarity, maintained support for realistic notebook tasks, and enabled novel strategies for clarity. Our study highlights the promise of Tidynote in supporting always-clear notebook authoring and reveals key design opportunities for future notebook systems that provide configurable clarity support, and for general information systems that balance exploration and clarity.

Acknowledgments

Our thanks to Savitha Ravi, Saketh Kasibatla, and the Foundation Interface Lab for helpful feedback on earlier prototypes. This work was supported by U.S. National Science Foundation Grants No. 2107397 (*Human-Centric Program Synthesis*) and No. 2432644 (*Direct Manipulation for Everyday Programming*).

References

[1] 2024. Hello GPT-4o. https://openai.com/index/hello-gpt-4o/
[2] 2025. Enso Analytics | Self-Service Data Prep and Blend built for Data Teams. https://ensoanalytics.com
[3] 2025. Home - Polynote. https://polynote.org/latest/
[4] Souti Chattopadhyay, Zixuan Feng, Emily Arteaga, Audrey Au, Gonzalo Ramos, Titus Barik, and Anita Sarma. 2023. Make It Make Sense! Understanding and Facilitating Sensemaking in Computational Notebooks. arXiv:2312.11431 [cs.HC] https://arxiv.org/abs/2312.11431
[5] Souti Chattopadhyay, Ishita Prasad, Austin Z. Henley, Anita Sarma, and Titus Barik. 2020. 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* (Honolulu, HI, USA) *(CHI '20)*. Association for Computing Machinery, New York, NY, USA, 1–12. doi:10.1145/3313831.3376729
[6] Victoria Clarke and Virginia Braun. 2017. Thematic analysis. *The journal of positive psychology* 12, 3 (2017), 297–298.
[7] Catherine H Crouch and Eric Mazur. 2001. Peer instruction: Ten years of experience and results. *American journal of physics* 69, 9 (2001), 970–977.
[8] Taijara Loiola De Santana, Paulo Anselmo Da Mota Silveira Neto, Eduardo Santana De Almeida, and Iftekhar Ahmed. 2024. Bug Analysis in Jupyter Notebook Projects: An Empirical Study. *ACM Trans. Softw. Eng. Methodol.* 33, 4, Article 101 (April 2024), 34 pages. doi:10.1145/3641539
[9] Hanxi Fang, Supawit Chockchowwat, Hari Sundaram, and Yongjoo Park. 2025. Enhancing Computational Notebooks with Code+Data Space Versioning. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems (CHI '25)*. Association for Computing Machinery, New York, NY, USA, Article 154, 17 pages. doi:10.1145/3706598.3714141
[10] Brian E Granger and Fernando Pérez. 2021. Jupyter: Thinking and Storytelling with Code and Data. *Computing in Science & Engineering* 23, 2 (2021), 7–14.
[11] Thomas R. G. Green and Marian Petre. 1996. Usability Analysis of Visual Programming Environments: A Cognitive Dimensions Framework. *J. Vis. Lang. Comput.* 7, 2 (1996), 131–174. doi:10.1006/jvlc.1996.0009
[12] Konstantin Grotov, Sergey Titov, Vladimir Sotnikov, Yaroslav Golubev, and Timofey Bryksin. 2022. A Large-Scale Comparison of Python Code in Jupyter Notebooks and Scripts. In *Proceedings of the 19th International Conference on Mining Software Repositories* (Pittsburgh, Pennsylvania) *(MSR '22)*. Association for Computing Machinery, New York, NY, USA, 353–364. doi:10.1145/3524842.3528447
[13] Philip J Guo and Dawson Engler. 2011. Using automatic persistent memoization to facilitate data analysis scripting. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. 287–297.
[14] Andrew Head, Fred Hohman, Titus Barik, Steven M. Drucker, and Robert DeLine. 2019. Managing Messes in Computational Notebooks. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems* (Glasgow, Scotland Uk) *(CHI '19)*. Association for Computing Machinery, New York, NY, USA, 1–12. doi:10.1145/3290605.3300500
[15] Ruanqianqian (Lisa) Huang, Savitha Ravi, Michael He, Boyu Tian, Sorin Lerner, and Michael Coblenz. 2025. How Scientists Use Jupyter Notebooks: Goals, Quality Attributes, and Opportunities. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '25)*. 13 pages.
[16] Sungwon In, Eric Krokos, Kirsten Whitley, Chris North, and Yalong Yang. 2024. Evaluating Navigation and Comparison Performance of Computational Notebooks on Desktop and in Virtual Reality. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) *(CHI '24)*. Association for Computing Machinery, New York, NY, USA, Article 606, 15 pages. doi:10.1145/3613904.3642932
[17] Dhanya Jayagopal, Justin Lubin, and Sarah E. Chasins. 2022. Exploring the Learnability of Program Synthesizers by Novice Programmers. In *Proceedings of the 35th Annual ACM Symposium on User Interface Software and Technology* (Bend, OR, USA) *(UIST '22)*. Association for Computing Machinery, New York, NY, USA, Article 64, 15 pages. doi:10.1145/3526113.3545659
[18] DaYe Kang, Tony Ho, Nicolai Marquardt, Bilge Mutlu, and Andrea Bianchi. 2021. ToonNote: Improving Communication in Computational Notebooks Using Interactive Data Comics. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems* (Yokohama, Japan) *(CHI '21)*. Association for Computing Machinery, New York, NY, USA, Article 727, 14 pages. doi:10.1145/3411764.3445434
[19] Mary Beth Kery, Amber Horvath, and Brad Myers. 2017. Variolite: Supporting Exploratory Programming by Data Scientists. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems* (Denver, Colorado, USA) *(CHI '17)*. Association for Computing Machinery, New York, NY, USA, 1265–1276. doi:10.1145/3025453.3025626
[20] Mary Beth Kery and Brad A. Myers. 2017. Exploring Exploratory Programming. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 25–29. doi:10.1109/VLHCC.2017.8103446
[21] Mary Beth Kery, Marissa Radensky, Mahima Arya, Bonnie E. John, and Brad A. Myers. 2018. 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*. ACM, Montreal QC Canada, 1–11. doi:10.1145/3173574.3173748
[22] Donald Ervin Knuth. 1984. Literate Programming. *Comput. J.* 27, 2 (1984), 97–111.
[23] Sam Lau, Ian Drosos, Julia M. Markel, and Philip J. Guo. 2020. 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)*. 1–11. doi:10.1109/VL/HCC50065.2020.9127201
[24] Haotian Li, Lu Ying, Haidong Zhang, Yingcai Wu, Huamin Qu, and Yun Wang. 2023. Notable: On-the-fly Assistant for Data Storytelling in Computational Notebooks. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems* (Hamburg, Germany) *(CHI '23)*. Association for Computing Machinery, New York, NY, USA, Article 173, 16 pages. doi:10.1145/3544548.3580965
[25] Yanna Lin, Leni Yang, Haotian Li, Huamin Qu, and Dominik Moritz. 2025. InterLink: Linking Text with Code and Output in Computational Notebooks. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems (CHI '25)*. Association for Computing Machinery, New York, NY, USA, Article 51, 15 pages. doi:10.1145/3706598.3714104
[26] Eric S. Liu, Dylan A. Lukes, and William G. Griswold. 2023. Refactoring in Computational Notebooks. *ACM Transactions on Software Engineering and Methodology* 32, 3 (July 2023), 1–24. doi:10.1145/3576036
[27] marimo. 2025. marimo | a next-generation Python notebook. https://marimo.io
[28] Observable. 2025. Observable. https://www.observablehq.com
[29] Yang Ouyang, Leixian Shen, Yun Wang, and Quan Li. 2024. NotePlayer: Engaging Computational Notebooks for Dynamic Presentation of Analytical Processes. In *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology* (Pittsburgh, PA, USA) *(UIST '24)*. Association for Computing Machinery, New York, NY, USA, Article 9, 20 pages. doi:10.1145/3654777.3676410
[30] Fernando Perez and Brian E. Granger. 2007. IPython: A System for Interactive Scientific Computing. *Computing in Science & Engineering* 9, 3 (2007), 21–29. doi:10.1109/MCSE.2007.53
[31] Joao 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)*. IEEE, Montreal, QC, Canada, 507–517. doi:10.1109/MSR.2019.00077
[32] Project Jupyter. 2024. Jupyter. https://jupyter.org
[33] Deepthi Raghunandan, Aayushi Roy, Shenzhi Shi, Niklas Elmqvist, and Leilani Battle. 2023. 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* (Hamburg, Germany) *(CHI '23)*. Association for Computing Machinery, New York, NY, USA, Article 863, 12 pages. doi:10.1145/3544548.3580997
[34] Eric Rawn and Sarah E. Chasins. 2025. Pagebreaks: Multi-Cell Scopes in Computational Notebooks. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems (CHI '25)*. Association for Computing Machinery, New York, NY, USA, Article 53, 16 pages. doi:10.1145/3706598.3713620
[35] Min RK. 2024. minrk/nbextension-scratchpad. https://github.com/minrk/nbextension-scratchpad
[36] Adam Rule, Ian Drosos, Aurélien Tabard, and James D. Hollan. 2018. Aiding Collaborative Reuse of Computational Notebooks with Annotated Cell Folding. *Proceedings of the ACM on Human-Computer Interaction* 2, CSCW (Nov. 2018), 1–12. doi:10.1145/3274419

[37] Adam Rule, Aurélien Tabard, and James D. Hollan. 2018. Exploration and Explanation in Computational Notebooks. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems* (Montreal QC, Canada) *(CHI '18)*. Association for Computing Machinery, New York, NY, USA, 1–12. doi:10.1145/3173574.3173606

[38] 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. doi:10.1145/3649838

[39] Ana Selvaraj, Eda Zhang, Leo Porter, and Adalbert Gerald Soosai Raj. 2021. Live coding: A review of the literature. In *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1*. 164–170.

[40] Paul Shen. 2024. natto.dev - write JavaScript on a 2D canvas. https://natto.dev/

[41] Krishna Subramanian, Nur Hamdan, and Jan Borchers. 2020. Casual Notebooks and Rigid Scripts: Understanding Data Science Programming. In *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, Dunedin, New Zealand, 1–5. doi:10.1109/VL/HCC50065.2020.9127207

[42] April Yi Wang, Anant Mittal, Christopher Brooks, and Steve Oney. 2019. How Data Scientists Use Computational Notebooks for Real-Time Collaboration. *Proceedings of the ACM on Human-Computer Interaction* 3, CSCW (Nov. 2019), 1–30. doi:10.1145/3359141

[43] April Yi Wang, Dakuo Wang, Jaimie Drozdal, Michael Muller, Soya Park, Justin D. Weisz, Xuye Liu, Lingfei Wu, and Casey Dugan. 2022. Documentation Matters: Human-Centered AI System to Assist Data Science Code Documentation in Computational Notebooks. *ACM Trans. Comput.-Hum. Interact.* 29, 2, Article 17 (Jan. 2022), 33 pages. doi:10.1145/3489465

[44] Fengjie Wang, Xuye Liu, Oujing Liu, Ali Neshati, Tengfei Ma, Min Zhu, and Jian Zhao. 2023. Slide4N: Creating Presentation Slides from Computational Notebooks with Human-AI Collaboration. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems* (Hamburg, Germany) *(CHI '23)*. Association for Computing Machinery, New York, NY, USA, Article 364, 18 pages. doi:10.1145/3544548.3580753

[45] Zijie J. Wang, Katie Dai, and W. Keith Edwards. 2022. StickyLand: Breaking the Linear Presentation of Computational Notebooks. In *Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems* (New Orleans, LA, USA) *(CHI EA '22)*. Association for Computing Machinery, New York, NY, USA, Article 269, 7 pages. doi:10.1145/3491101.3519653

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

[47] K. N. Whitley and Alan F. Blackwell. 1997. Visual Programming: The Outlook From Academia and Industry. *Workshop on Empirical Studies of Programmers* (1997). https://doi.org/10.1145/266399.266415

[48] Dongwook Yoon, Nicholas Chen, and François Guimbretière. 2013. TextTearing: opening white space for digital ink annotation. In *Proceedings of the 26th annual ACM symposium on User interface software and technology*. 107–112.

[49] Chengbo Zheng, Dakuo Wang, April Yi Wang, and Xiaojuan Ma. 2022. Telling Stories from Computational Notebooks: AI-Assisted Presentation Slides Creation for Presenting Data Science Work. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems* (New Orleans, LA, USA) *(CHI '22)*. Association for Computing Machinery, New York, NY, USA, Article 53, 20 pages. doi:10.1145/3491102.3517615

A AI-Generated Cell Summary for Cell Folding in Tidynote

To facilitate cell folding (*i.e.*, showing a more meaningful information than "···" as in JupyterLab), Tidynote generates a summary for every executed cell. When a code cell is executed, Tidynote records the cell code, summarizes the code and variables newly defined with GPT-4o, and inserts the summary at its beginning. Tidynote only re-summarizes if the cell code (with comments and whitespace removed) differs from prior record. The prompt is a formatted string in TypeScript:

```
const priorCodePrompt = prevCode ? `The code prior to
    ↪ this cell that has been executed
    ↪ is:\n\n\`\`\`\n${prevCode}\`\`\`\n\n` : '';

const prompt =
`I am writing code in this notebook cell:
\`\`\`
${origCodeSansOutdatedComments}
\`\`\`

${priorCodePrompt}

Summarize the code in this cell in a comment with no
    ↪ more than ten words.

Summarize the variables newly defined in this cell in
    ↪ another comment with no more than ten words.

Return only the two comments starting with #, separated
    ↪ by a newline.`;
```

`origCodeSansOutdatedComments` is the code of the cell to be summarized with existing summaries removed, if any. `prevCode` is code from cells linearly above to the given cell that have been executed. We only include `prevCode` in the prompt (*i.e.*, in `priorCodePrompt`) if it is non-empty.