arXiv:2507.15831v3 [cs.SE] 2 Jul 2026

Observing Fine-Grained Changes in Jupyter Notebooks During Development Time

Sergey Titov[a,*], Konstantin Grotov[b], Cristina Sarasua[e], Yaroslav Golubev[c], Dhivyabharathi Ramasamy[e], Alberto Bacchelli[e], Abraham Bernstein[e], Timofey Bryksin[d]

[a] *JetBrains Research, Amsterdam, The Netherlands*
[b] *JetBrains Research, Munich, Germany*
[c] *JetBrains Research, Belgrade, Serbia*
[d] *JetBrains Research, Limassol, Cyprus*
[e] *University of Zurich, Zurich, Switzerland*

Abstract

In software engineering research, the analysis of fine-grained logs led to significant innovations in areas such as refactoring, security, and code completion. However, even though computational notebooks are a staple of data science and an important tool in machine learning, few similar studies have been conducted in this area.

To help bridge this research gap, this paper makes three scientific contributions. (1) We introduce a *toolset* for collecting code changes in Jupyter notebooks during development time. (2) We use it to collect more than 100 hours of work related to a data analysis task and a machine learning task (carried out by 20 developers with different levels of expertise), resulting in a *dataset* containing 2,655 cells and 9,207 cell executions. (3) Finally, we use this dataset to *investigate* the dynamic nature of the notebook development process and the changes that take place in the notebooks.

In our analysis of the collected data, we classified the changes made to

*Corresponding author

Email addresses: sergey.titov@jetbrains.com (Sergey Titov), konstantin.grotov@jetbrains.com (Konstantin Grotov), sarasua@ifi.uzh.ch (Cristina Sarasua), yaroslav.golubev@jetbrains.com (Yaroslav Golubev), ramasamy@ifi.uzh.ch (Dhivyabharathi Ramasamy), bacchelli@ifi.uzh.ch (Alberto Bacchelli), bernstein@ifi.uzh.ch (Abraham Bernstein), timofey.bryksin@jetbrains.com (Timofey Bryksin)

the cells between executions and found that a significant number of these changes constituted code iteration modifications. We report a number of other insights and propose detailed future research directions on the novel data.

Keywords: Computational Notebooks, Jupyter Notebooks, Software Evolution, Fine-Grained Logs

---

1. Introduction

The analysis of fine-grained software development logs provides information about the development process at a higher level of detail compared to the analysis of version control system (VCS) snapshots [1]. In the last several decades, software engineering research has employed development logs to study, for example, program security [2], performance [3], and code completion [4]. In particular, *fine-grained execution logs* that save the information about various activities during the code writing process [5] have opened up the possibility of studying developer behavior, *e.g.*, how the integrated development environment (IDE) affects developers with ADHD [6].

When it comes to computational notebooks, however, there are almost no studies or datasets on their fine-grained evolution or execution during development time. For this reason, most of the studies analyze computational notebooks using examples downloaded from public version control systems. For example, Pimentel et al. [7] show that a significant portion of notebooks are not reproducible due to incorrect cell ordering. Grotov et al. [8] compare notebooks to Python scripts and find that notebooks generally have lower code quality. Yet, having fine-grained data on notebook development and evolution is critical, because notebooks represent an alternative paradigm of literate programming [9] that possesses unique execution characteristics. This was brought forward by the analysis of fine-grained VCS logs in the work of Raghunandan et al. [10], where the authors examine multiple historical versions of the same notebook from version control systems and demonstrate that notebooks can change drastically between versions.

One study based on truly fine-grained logs is the work of Chen et al. [11], which explores execution logs in Jupyter Notebooks. The authors analyze notebook executions to identify the types of errors made by data science practitioners. The fine-grained resolution of these logs allowed them to examine the specific strategies users employ to fix errors and to suggest ways

to improve automated program repair.

To go further and broaden this research, we propose three contributions. As our first contribution, we devised the tooling necessary to obtain fine-grained log data. The main tool is a plugin for the Web-based Jupyter Notebook IDE that tracks and logs executions of the opened Jupyter notebook upon every action. Additionally, we developed a server-side application that receives and stores this data, as well as a collection of post-processing scripts for parsing, analyzing, and visualizing the collected data. The tooling is designed to be convenient to use, adaptive to running live experiments, and easy to extend, leading to further data collection and studies in this area. It is available online [12].

As our second contribution, using the developed tools, we collected the dataset of fine-grained execution logs for Jupyter notebooks — Jupyter Notebooks Executions (**JuNE**). To this end, we organized a series of day-long experiments where we asked participants to solve at least one of two development tasks — one related to data analysis and one related to machine learning. Overall, 11 students (enrolled in Master's and Bachelor's studies that include Data Science-related topics) and 9 industry professionals (with a minimum of one year of experience in a Data Science-related role) participated and sent us their data. The resulting dataset contains the information about a total of 2,655 developed cells and 9,207 cell executions, and includes timestamps, the content of the cells on every execution, the labeling of the data science steps in the cell using an annotation model, the type of the implemented action (*e.g.*, cell creation, cell deletion, or cell execution), and the cell execution order. We release this dataset to enable further studies taking advantage of these fine-grained execution logs [12].

As our third contribution, having obtained the dataset, we carried out a fine-grained analysis of the development process in Jupyter notebooks. We focus on the code changes made to cells during the iterative process of editing and execution. By analyzing this data, we investigate how the creation and evolution process in the notebooks unfolds—what notebook users do while working on cells and how we can support them more effectively.

In order to analyze changes, we transformed the dataset to represent notebook development as a series of transitions from one cell to another. We identified two types of transitions: *inter-transitions*, where the developer moves from one cell to another, and *self-transitions*, where the developer re-executes the same cell. We found that 39% of transitions are self-transitions, while the remaining 61% are inter-transitions. Self-transitions represent the

continuous development of a cell and will be the primary focus of our empirical analysis, as we are interested in code changes.

Additionally, we introduced two new annotations for the cells. First, we annotated each self-transition with one of 11 possible *purposes* (*e.g.*, "fix", "clean code", "explore variable", etc.). We conducted a manual open coding on 400 examples and then used GPT-4o to annotate the remaining transitions. Second, we annotated each cell with its corresponding *data science step*, based on the classification introduced by Ramasamy et al. [13] (*e.g.*, "evaluation" or "load data"). To annotate the data science steps, we developed our own model tailored to our data format, which performs slightly better than the original.

Through our analysis we observe that the changes in notebooks are on average relatively small (with only about 13% of a cell being modified) and that this size decreases with consecutive re-executions of the same cell. We found that most changes revolve around the process of code iteration, with 72.2% of the change purposes in self-transitions aimed at fixing, debugging, editing, cleaning code, improving readability, etc., while exploration is a relatively rare goal, accounting for only 14.1% of transitions. Lastly, we found evidence that data science steps are highly stable during self-transitions—the type of a cell rarely changes during cell development, but changes more frequently in the case of inter-transitions.

The results lead us to hypothesize that the interactive nature of notebooks is leveraged for debugging rather than for exploration with rich outputs. This highlights the need for more convenient debugging tools and IDE-like inspection features to check the code prior to execution.

In summary, this work makes the following main contributions:

- **Tooling** for collecting fine-grained execution logs in Jupyter notebooks, consisting of a plugin to capture user activity, a server to receive and store the data, and a collection of post-processing scripts to parse and analyze it. We also provide a tool to replay notebook executions, effectively allowing to explore the notebook's state at various points. Our data collection effort also ensured that the tooling can handle real-world, extensive usage.

- **Dataset** of fine-grained execution logs that we collected from 11 students and 9 industry professionals. In total, the dataset contains the data about 2,655 developed cells and 9,207 cell executions—for a total

of more than 100 hours of data science and machine learning related work—together with detailed information about their evolution and content.

- **Empirical findings** about the changes that occur in notebooks during their development, which reveal important insights. We annotated the collected data based on the purpose of the changes in code transitions and the data science step they refer to. Our analysis shows that the development process in notebooks is highly non-linear and lacks proper support for development tools. We suggest that further analysis of this data could provide valuable insights for improving notebook tooling.

The remainder of the paper is organized as follows. In Section 2, we describe the existing work studying Jupyter notebooks and collecting fine-grained software logs. In Section 3, we present the tooling we developed for collecting fine-grained logs in Jupyter notebooks. We detail our data collection process in Section 4 and then present the obtained data in Section 5. In Section 6, we describe the methodology of our analysis, and present the findings in Section 7. Then, we discuss the implications of our study and the threats to its validity in Section 8. Finally, we present future research directions in Section 9 and conclude in Section 10.

2. Background and Related Work

2.1. Code Evolution in Software Engineering

More than ten years ago Negara et al. [1] raised questions about the reliability of VCS data for investigating software evolution and what it lacks. Using an Eclipse plugin, they collected fine-grained development logs and compared them with VCS data from the same project. They found that 37% of changes occur during development and never reach VCS. Additionally, they discovered insights in development logs that are difficult to glean from VCS data alone. For example, they found that 24% of the changes eventually committed to VCS had not been tested prior to their inclusion.

The analysis of development logs has led to multiple innovations in the current generation of IDEs. For example, in another paper [14], the same authors provide a comprehensive analysis of refactoring techniques employed by developers. They demonstrated that half of all refactorings are done manually, even though many could be automated by the IDE. For instance, their

study highlights that the frequency of automated *Extract Method* refactoring does not correlate with the size of the code, suggesting usability issues. Developers tend to make manual changes, even when they involve rewriting larger pieces of code.

Another significant work in the field is the study of Yoon et al. [15], in which the authors provided evidence of a significant number of backtracking actions during the development process and a lack of tools to support them. The authors collected log data about code editing in Eclipse and characterized the different types of backtracking, as well as the frequency and size of the backtracking. They also proposed a tool to selectively undo the desired previous edits without undoing certain intermediate changes.

There are other studies that leverage development logs for code completion [4], various types of refactoring [16], as well as test design and generation [17]. Due to the changes in the availability of detailed development logs, the number of studies conducted in this area has recently declined. With increasing technical complexity of the systems, as well as legal complexity of the entire field, collecting such data becomes more problematic [18].

At the same time, collecting and analyzing fine-grained development logs in the context of Jupyter Notebooks is crucial for advancing our understanding of how notebooks are authored, used, and evolved over time. Unlike traditional software development environments, notebooks offer a unique blend of narrative, code, and execution semantics, which likely leads to distinct development and maintenance practices. For instance, it remains unclear whether and how refactoring strategies differ in notebooks. As observed by Titov et al. [19], notebook cells can be conceptualized as proto-functions. Fine-grained logs could thus reveal notebook-specific refactoring patterns—such as the extraction or reorganization of cells—that are not observable through coarse-grained data alone.

2.2. Coding in Jupyter Notebooks

Studies of computational notebooks are an established subfield in software engineering research [20], with important empirical insights. Research showed that notebooks have a lower rate of reproducibility [21, 7], significantly differ in terms of code structure from Python scripts [8], and frequently contain code clones [22]. These results show that the analysis of code in notebooks is challenging, which affects the performance of tools for assisting developers, such as code completion or code linting.

Recently, Ramasamy et al. [13] analyzed the developers' workflow in 470 notebooks and showed the probability of transitions between different data science steps in subsequent cells of the final snapshot of a notebook (*e.g.*, data pre-processing, modeling, evaluation, etc.). The results revealed that 53.25% of the code cells in the analyzed data science workflows focused on the tasks of data pre-processing and data exploration. In our paper, we used the classification of data science steps from this work because this taxonomy is the most complete. Other taxonomies, *e.g.*, from the work of Wang et al. [23], do not distinguish between model training and model evaluation, while loading the data is not captured at all.

There were also attempts to look at the notebook's evolution over time. Raghunandan et al. [10] looked at how a set of 2,574 notebooks changed over different commits on GitHub. More specifically, they analyzed the extent to which the notebooks change or maintain their purpose (exploratory or explanatory). Their study revealed that most of the notebooks in their dataset (60.1%) keep their explanatory nature over the commits, 22.6% keep their exploratory purpose, while 15.1% of the notebooks went from being exploratory to being explanatory, and only 2.1% transitioned from being explanatory to being exploratory.

In a recent work, Chen et al. [11] analyzed fine-grained Jupyter notebook traces. The authors collected fine-grained execution logs to develop a taxonomy of the errors that occur during data science work and to examine how users fix them. Their analysis shows that most errors during notebook development occur in the data exploration and preprocessing phases, and are usually fixed by changing parameters or correcting syntax errors. They argue that these findings could significantly improve automated program repair for Jupyter notebook use cases, since the range of errors is relatively small.

We believe that this field needs to be extended further to move away from coarse-grained commit-centered dynamics and support better developer tooling. Our work focuses on wider patterns, not just errors, presents mature tooling suitable for large-scale experiments and replaying collected notebooks, a novel dataset, and more empirical findings about the nature of work in Jupyter notebooks.

3. Tooling

To collect fine-grained execution logs of Jupyter notebooks and to conduct our empirical study, we developed the necessary tooling. It consists of four

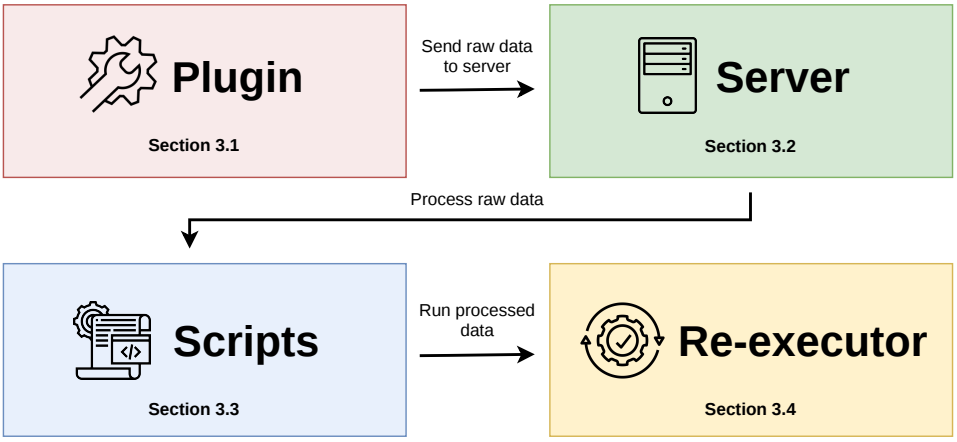

Figure 1: The pipeline of the developed tooling.

main parts, visualized in Figure 1:

- A **plugin** to collect the necessary data.
- A **server** to receive and store the data.
- A collection of **post-processing scripts** to process the data and conduct the analysis.
- A **tool to re-execute notebooks** from obtained logs, given the correct environment.

3.1. Activity Plugin

To collect user activity, we developed a JavaScript plugin as an extension for the Jupyter Notebook Web application. The plugin tracks the initial launch of a Python notebook, as well as its interruption and restarting. Additionally, the extension tracks certain events that occur during the development process in the notebook, including *creating a new cell*, *deleting a cell*, *executing a cell*, *rendering a Markdown cell*, *finishing executing*, *changing a cell type* (from code to Markdown or vice versa), and *errors* (when the executed cell finished by raising an error). For each action, we save all the possible information about the cell, the notebook, and the user. For

example, when a cell is executed, the plugin collects the index of the cell, its ordinal number in the notebook, and the cell's source code. Also, for all types of events, we save information about the session, the notebook kernel, the notebook name, and the timestamp. Upon collection, the data is stored locally and, if necessary, sent to a remote server, the address of which can be set up in the settings.

During the development of the plugin, our main goal was to minimize user interaction to ensure that it does not influence the coding process. To begin using the plugin, the users first need to install it as a Python package. Once installed, they activate it as a Jupyter extension. Before starting the plugin, the users are prompted to consent to send data to a remote server and provide the server's address. Once these initial steps are completed, the plugin transparently runs in the background, collecting data across all Jupyter notebooks that are accessible to the given Jupyter Notebook application instance, without interrupting the user. This supports authentic studies that can provide realistic insights into developers' workflows.

3.2. Server

To handle the data generated and sent by the plugin, we developed a remote Python server using Flask [24]. The server receives the data using a `get` request and saves it to a local SQLite database [25]. When the experiment is finished, the server is shut down. The presence of a dedicated server helps run large-scale experiments, without necessarily relying on manually sending and saving files.

3.3. Post-processing Scripts

The data collected from the plugin consists of a collection of raw events from all participating users. Due to the internal implementation of the Jupyter Notebook Web application, one can find inconsistencies in the raw data. For example, Jupyter kernel (Notebook version 6.4.12) sends the *rendered* status for a new Markdown cell before the creation of the corresponding cell. To address possible issues, we developed a set of Python scripts that perform post-processing on the data using a number of heuristics. This package allows for more convenient and efficient analysis of the collected data by transforming it from raw JSON files into two table formats: a log table and a set of notebook snapshots. The log table contains every action for each user of the plugin in chronological order. Snapshot tables serve as a set of snapshots, preserving the entire sequence of notebook cells after each action

recorded in the log table. These transformations enable the use of various Python libraries. For example, they enable the analysis of the entire log of the notebook as a time series of actions, or the analysis of each snapshot of the notebooks to track the evolution of the code.

3.4. Browsing the Notebook Versions

A key benefit of the data collected with our tools is that it allows researchers to reproduce the developers' workflow in the notebook. To do that, we developed a tool for setting up the necessary environment and then reproducing the notebook to browse its particular version. After the user selects a specific collected notebook, a new Jupyter notebook is created within the separate Docker container. Each development log corresponds to a notebook workflow and is translated into actions within this environment, effectively reproducing the developer's actions. This enables the analysis of not only the code but also the state of the environment at each step of the collected data. For example, this could help train a machine learning model that relies on runtime information for code completion.

4. Data Collection

The data collection procedure for our empirical study was designed around two variables that could influence the notebook development process: (1) the *type* of the solved task and (2) the level of participants' *expertise*. We designed two distinct tasks that demanded a wide spectrum of data science skills and, to analyze how the level of expertise impacts the development process, specifically targeted two groups of people: computer science students and industry data science professionals.

4.1. Designing Tasks

We developed two specific tasks for the experiment, designed to emulate the main activities found in Jupyter notebooks: a data analysis task (DA) and a machine learning task (ML). Our principal design goal was to create tasks that replicated the development and usage of an authentic notebook, both in terms of content and duration. We estimated each task to take around four hours, totaling eight hours—the approximate working day for many data scientists. To estimate completion time, the first two authors of this paper tested how long each task would take and adjusted them accordingly.

To encourage the participants to create good quality solutions, we informed them that all solutions would be scored, and we made available a public leaderboard for each task for the duration of the experiment. The full text of the tasks and the scoring criteria can be found in the supplementary materials [12]. Let us now briefly describe them.

4.1.1. Data Analysis Task

For the DA task, participants were provided with a dataset of synthetic logs created by the authors, containing logs of user actions on some imaginary social network where users could post, like, or follow someone. The task consisted of three sub-tasks: (1) data engineering, (2) metrics evaluation, and (3) data visualization.

The first sub-task required participants to parse the input dataset into a designated table format and perform data cleaning. We injected the dataset with structural and syntactic errors in the log strings to test the participants' ability to identify and fix these errors. The second sub-task involved completing a set of statistical tasks, such as calculating the mean number of actions per user. Participants were given a total of five predefined statistical tasks to solve, along with an additional task where they had to devise their own metric of user activity based on the provided data. The third sub-task concerned the visualization of selected metrics. We instructed participants to create several distinct plots based on the provided data, including line plots, bar charts, and heatmaps. Additionally, we asked participants to create a visualization of any measurement they had computed that had not been visualized so far.

Overall, participants were required to complete typical data analyst tasks, involving actions such as data cleaning, exploration, aggregation, and, ultimately, visualization.

4.1.2. Machine Learning Task

For the ML task, we created a Kaggle-like [26] competition. We used the already validated dataset and task instructions from the work of Ramasamy et al. [13]. The dataset consists of 9,678 Jupyter notebook cells, categorized into one of ten possible classes by a group of five experts. The goal of this classification task is to automatically classify the type of data science steps present in Jupyter notebook cells. The dataset also included pre-computed features, such as the number of lines of code in a cell and the number of unique variables in each cell. The study participants were given access to both the

train and test subsets of the dataset from the paper, and their final solutions were evaluated on a separate subset. Following the original procedure from the paper, the evaluation of the classification model was implemented using the weighted F1-score.

The ML task included data science steps complementary to the steps present in the DA task: data exploration, data pre-processing, modeling, evaluation, and result visualization.

4.2. Choosing Participants

To recruit participants, we issued two calls for participation: one targeting students at two universities and one addressing employees at two companies. Interested individuals were asked to complete a short questionnaire about their Python experience and the frequency with which they use Jupyter notebooks. This helped us filter out participants with less than one year of Python experience or no Jupyter notebook experience. The final sample consisted of 20 people: 11 students and 9 industry professionals.

4.3. Executing the Experiment

Since the experiment took place in four different locations (two universities and two companies), we gathered all eligible participants into groups of two to nine people. Each person in the group was asked to solve at least one of the tasks, and participants implemented their notebooks individually. We provided each group with a repository containing task descriptions and the necessary data. Upon presenting the repository, we activated a nine-hour timer, which allowed approximately eight hours for task completion and an additional hour for lunch. After nine hours, we ceased logging and evaluated each solution based on the most recently logged notebook. Following our evaluation, we shared the leaderboard results with each group, as well as the aggregated leaderboard results of all previous groups.

5. Dataset

The resulting dataset of Jupyter Notebooks Executions (**JuNE**) contains more than 100 hours worth of execution logs from 11 students and 9 professionals. Students and professionals differ not only in their formal status but also in their Python experience: (M=56.3, SD=31.9) months for professionals and (M=33.3, SD=11.5) months for students. 80% of participants reported using Jupyter notebooks more than three times a week.

During the experiment, the participants solved a total of 29 tasks, resulting in 29 notebooks. This data included 16 solutions of the DA task and 13 solutions of the ML task. Overall, the data includes 14,641 user events, including 9,207 cell executions, 1,930 cell creations, and 730 cell deletions.

For each of the 29 notebooks, the dataset contains all the notebook's actions in chronological order, together with cell contents and all the meta-information.

6. Empirical Analysis Methodology

Since our dataset offers a unique opportunity to examine the development process within notebooks, we decided to focus on this aspect by analyzing code changes. We seek new insights into how people use notebooks in real-world settings and to inform tool developers on how to support the development process. Our exploration of the code changes is structured around the following research questions:

- **RQ1: How do code changes take place during cell development in Jupyter Notebooks?** Mining the execution traces of notebooks for changes provides a solid analysis of code evolution. More importantly, once collected, this data can offer far deeper insight into the Jupyter development process and notebooks as an environment than any stable Git snapshots.
- **RQ2: What are the purposes of these code changes?** To deepen our understanding of the discovered changes, we aim to develop a taxonomy based on our unique data. This will help refine the interpretation of the acquired data and provide a clearer picture of how notebooks evolve, as well as highlight the main reasons users interact with code in notebooks.
- **RQ3: How are the data science workflow steps represented during notebook development?** Lastly, we want to examine how the changes align with the data science development process and how the dynamic nature of our data reflects it.

The overview of the entire analysis can be found in Figure 2. Before conducting the analysis, we preprocessed the data to focus on the code changes. First, we filtered the dataset to retain only execution events, representing

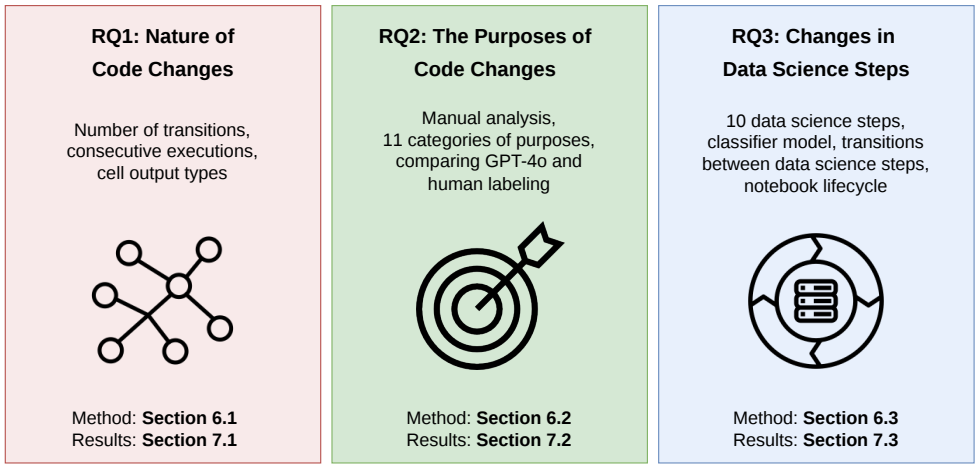

Figure 2: The overview of our empirical analysis.

granular but logically finished steps. Then, we transformed the dataset to represent notebook development as a series of transitions from one cell to another. From the development log, we reconstructed the data as follows: for each execution, we recorded information about the state of the executed cell, including the time of the previous execution, the source code, and the saved outputs. We also stored the state of the next cell to be executed. From this data, we identified two types of transitions: *inter-transitions*, where the developer moves from one cell to another, and *self-transitions*, where the developer re-executes the same cell.

To clarify the notion of transition, we consider the evolution graphs of the resulting notebooks. Figure 3 shows an example of an evolution graph, where the nodes represent cells and edges represent transitions to the next executed cells. We can observe both transition types: self-transitions are highlighted in blue boxes, while inter-transitions are highlighted in red boxes. Self-transitions represent the continuous development/evolution of a cell and are the primary focus of our empirical analysis, as we are interested in code evolution. In contrast, inter-transitions occur between different cells: they may represent the continuous development of a single idea, but they could also stand for two unrelated pieces of code. As it is not possible to automatically understand the reason why two cells are separated, we adopt a conservative approach and mostly focus on the self-transitions.

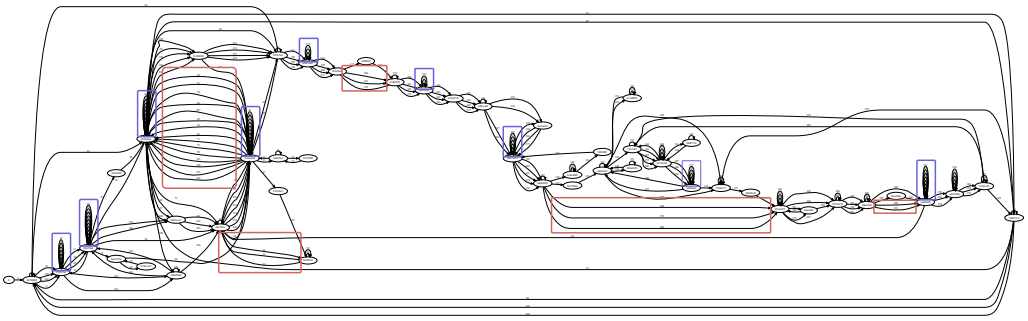

Figure 3: Example of structure of the evolution graph. Red boxes highlight examples of inter-transition and blue boxes highlight examples of self-transitions (loops).

6.1. RQ1: Nature of Code Changes

To describe how source code evolves during notebook development, we calculate and present several metrics. First, we provide the overall number of inter-transitions and self-transitions. The distribution of the number of subsequent self-transitions (*i.e.*, the lengths of re-execution chains) helps us determine how many times, on average, a cell is re-executed in a row, providing insight into the iterative nature of working in a notebook cell. We also examine the distribution of edit distance between two consecutive executions, measured by the normalized number of changed symbols. This metric allows us to understand the average size of a change, revealing whether users tend to rewrite the cell entirely or make minor tweaks between executions. Lastly, we calculate the distribution of cell output types (*e.g.*, text, rich data, errors, etc.) before the self-transition. This analysis can offer insights into the reasons prompting the re-execution of a cell.

6.2. RQ2: The Purposes of Code Changes

To identify the purpose of the code changes, and thus the users' intentions when changing the code, we *qualitatively* analyzed the code before and after self-transitions. The reason why in this part we focus exclusively on self-transitions is that inter-transitions consist of a developer executing two different cells (with two different pieces of code). Therefore, inter-transitions do not provide relevant information to analyze the evolution of the code.

Due to the large scale of our dataset (3,573 self-transitions), we used a combination of manual open coding and automatic labeling using GPT-4o.

We manually annotated a sample of 400 self-transitions to define a set of labels that we could provide to GPT-4o in the prompts. Specifically, two authors of the paper manually reviewed 200 transitions pertaining to the data analysis task and 200 pertaining to the machine learning task. These transitions were randomly sampled following a stratified sampling based on the users, since some users generated more cells than others. Both annotators labeled each of these transitions following an open coding methodology, analyzing the source code of the cell before re-execution and the source code of the cell after re-execution. Subsequently, the two annotators discussed the resulting set of distinct labels (*i.e.*, their labeling schemes) and identified mappings between them. While one of the annotators had a more fine-grained set than the other (*e.g.*, one annotator defined the label "modification", while the other annotator had differentiated between "correct to match what they aim for", "edit to a new variable version", and "test different input data"), we were able to map the labels with 1:1 and 1:N mappings and agree on the necessary specificity. As a result, we identified 11 labels shown in Table 1.

As a second step, we programmatically labeled the self-transitions using GPT-4o. We provided GPT-4o with a system prompt that instructed the LLM to classify code transitions based on the purpose of the changes. Additionally, the system prompt provided our resulting labeling scheme and instructed that if GPT-4o found a case with a label that was not present in our scheme, GPT-4o should use its own label consistently across transitions. The user prompt focused on asking GPT-4o to label one code transition at a time. Both prompts together had a length smaller than the maximum context (*i.e.*, 128k tokens), and, since they mostly duplicate the description of the classes provided in Table 1, they can be found in supplementary materials [12]. Two new labels were generated by GPT-4o ("unknown" and "remove debugging code"), but they were only present in one case each.

Finally, as a third step, we revised the labels from the open coding together with GPT-4o's labels, to define the ground truth labels for the sample of 400 transitions. Comparing the three annotations simultaneously was useful, as in some cases, GPT-4o had provided the correct answer, while the human annotators had not—probably due to human factors, such as tiredness and lack of consistency.

6.3. RQ3: Changes in Data Science Steps

To understand the evolution of the notebooks in the context of the data science workflow, we annotated our data with labels of various data science

Name	Explanation
`no change`	The two pieces of code in the transition are identical. There is no change.
`explore variable`	Code is added or edited to (further) explore the content or shape of a variable without changing the state of the code. In other words, there is no variable assignment involved in or affected by the code change. Operations may be applied to the content of a variable, but then the result is only displayed or printed, not assigned to a variable.
`fix`	Code is edited or extended to fix a syntactic error.
`debug`	Code is changed and/or added to 1) identify the source of a problem that leads to functional incorrectness or 2) to find a solution to the problem, or both 1) and 2).
`edit code`	Code is changed to adjust certain things and possibly obtain better results. This kind of change changes the state of code. In other words, there is a variable assignment involved in or affected by the code change. Please note that editing code to explore a variable in other ways should not be classified as 'edit code', but instead it should be classified as a case of 'explore variable'.
`clean code`	Code was deleted.
`visualize data`	A data visualization (e.g., a plot) is created or edited.
`extend code`	Code that is extended with new partial or complete pieces of code to drive the development further. This kind of change changes the state of code. Please note that adding code to explore a variable further should not be classified as 'extend code' but instead it should be classified as a case of 'explore variable'. Analogously, adding code to create or refine a visualization should not be classified as 'extend code' but instead it should be classified as a case of 'visualize data'.
`improve readability`	Code was changed to improve the formatting of the code and make it easier for developers to read and understand it.
`comment`	Comment code.
`uncomment`	Uncomment code that was previously commented.

Table 1: Resulting classification for the purposes of the cell change after expert annotations. The selected labels were obtained by manually annotating 200 random transitions from each task (400 total), followed by a discussion until full agreement on the labels was reached.

Data science step	Definition (taken from [13])	% of events
`data_preprocessing`	The process of preparing a dataset(s) for the subsequent analysis. It includes tasks such as cleaning, instance selection, normalization, data transformation, and feature selection.	37.1%
`data_exploration`	The process of inspecting the content and shape of a dataset to understand the nature and characteristics of the data. Note that it may involve the usage of visualization techniques but differs in its purpose.	29.8%
`comment_only`	Lines of comments including commented code.	11.2%
`modelling`	The process of applying statistical models and learning-based algorithms to learn from sample data.	7.3%
`helper_functions`	Code that is not directly related to the data science activity at hand but provides useful scripting functions (*e.g.*, importing or configuring libraries).	6.3%
`load_data`	The process of loading a dataset of any type (*e.g.*, .csv, .pkl) into a Jupyter notebook environment.	5.5%
`evaluation`	The process of assessing a model using one or more evaluation metrics such as goodness of fit and accuracy.	1.7%
`prediction`	The process of applying a model trained on a set of data to other or newly arriving pieces of data to forecast new values.	1.0%
`result_visualization`	The process of obtaining a graphical representation (e.g., tables, plots, graphs) of measurements	0.1%
`save_results`	The process of serializing and storing the data.	0.1%

Table 2: Data science steps and their distribution (in the descending order) in the dataset.

steps. To generate the labels, we use the characterization of data science steps in computational notebooks provided by Ramasamy et al. [13]. A complete list of data science steps, their definitions, and their amount in our data can be found in Table 2.

To identify the data science step for each logged cell in our data, we developed a single-label classifier based on the DASWOW dataset [13]. Originally, the DASWOW model was designed to analyze standalone notebooks in a multi-label setup, which made it more challenging to use in our case, as it complicates the analysis of transitions between distinct data science steps. To address this limitation, we made several key architectural modifications. First, we converted the multi-label classification problem into a single-label classification task by training only on the *primary_label* from DASWOW, ensuring each cell receives exactly one data science step label. Second, we replaced the original feature extraction approach with pre-trained CodeBERT embeddings [27], which provide robust semantic representations of code. Third, we employed CatBoost [28] as the classification algorithm instead of the original architecture, which provided better performance and more reliable predictions without requiring a certainty threshold that would leave some cells unlabeled.

We trained the CatBoost classifier with 400 iterations, a learning rate of 0.2, and weighted F1 score as the evaluation metric, similar to the original paper. The model was validated using 5-fold stratified cross-validation on the DASWOW training set, achieving a cross-validated F1 score of 0.71 ± 0.02. On the DASWOW test set, we achieved an F1 score of 0.721, even slightly higher than the original model's F1 score of 0.716. Importantly, unlike the original model, which employed a certainty threshold and could not label all examples, our model can classify every cell with a single label, making it more suitable for transition analysis.

To understand whether the data science steps change at the cell level during development, we studied the probabilities of transitions between various pairs of data science steps. To investigate the distribution of data science steps over time, we divided the log data for a given notebook into ten quantiles sorted by time, and then counted the steps for each bin. We hypothesize that certain steps may be more prevalent at the start or the end of the development process in Jupyter notebooks, *e.g.*, that in general data exploration precedes modelling.

In this research question, in addition to analyzing self-transitions, we include an analysis of inter-transitions. Examining the differences in the pro-

cess between working within a single cell and working across multiple cells can be insightful. We hypothesize that there is minimal switching between data science steps during self-transitions, while inter-transitions will demonstrate more frequent transitions between different steps, *i.e.*, people are less likely to switch their step of the workflow while being in the same cell. The reason for this is that if cells can be interpreted as proto-functions [19], then they should be more logically consistent internally than between different cells. In contrast to self-transitions, where we normalize the probabilities for each class, for inter-transitions, we ensure that the sum of all elements in the matrix is equal to 1. By doing so, we can interpret the probability associated with each transition (for each cell in the matrix) as the likelihood of that specific transition occurring during the development process. This provides insights into the significance of particular transitions within the total number of transitions during development.

7. Findings

7.1. ***RQ1:*** *Nature of Code Changes*

We found that self-transitions account for 39% of all transitions, while inter-transitions account for the remaining 61%. As can be seen in the presence of multiple loops in the example evolution graph (see Figure 3), self-transitions frequently occur in series. Figure 4a displays the distribution of the number of subsequent self-transitions, revealing that the mean number of re-executions for cells that have been re-executed at least once is 5. Additionally, 5% of the cells with at least one re-execution account for 25.6% of all re-executions. This shows that a relatively small percentage of code in notebooks demands a significant amount of effort from users, which might indicate the need for special tooling.

To explore the changes further, we plotted the correlation in Figure 4b between the number of re-executions and the mean normalised edit distance in symbols per re-execution. From this plot, we observe that the number of changes decreases over time ($r=0.2$, $p \leq 0.01$), thus indicating that a large number of re-executions typically reflects iterative tweaking of the code, such as debugging or exploration, rather than a development process. The mean change size between two consecutive runs is approximately 13% of the cell size. Based on this small change size, it seems reasonable to think that this corresponds to a fix or the addition of a single statement.

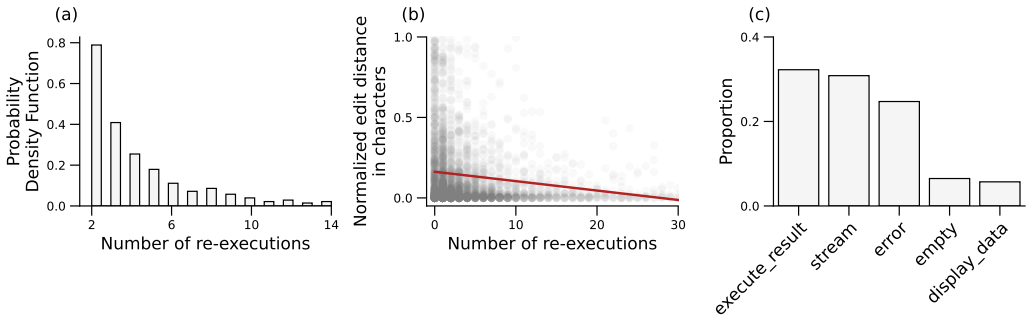

Figure 4: (a) Distribution of the number of consecutive re-executions. Note the exponential nature of the distribution — cells with only one re-execution account for 25.6% of all re-executions. (b) Correlation between the normalized edit distance between re-executions and the number of consecutive re-executions. Cells with a higher number of re-executions tend to have smaller edit distances, which may indicate a debugging process. (c) Distribution of output types prior to re-execution. Error outputs represent only one third of the outputs in cells before re-execution, suggesting that re-execution is more often related to semantic bugs rather than syntactic ones.

Lastly, we analyzed the distribution of types of output before self-transitions. Figure 4c shows: (1) *execute_result* – rich output of the cell, *e.g.*, dataframes; (2) *stream* – text output from print and other stream sources; (3) *error* – error messages from interpreter; (4) *empty* – no output for the cell; (5) *display_data* – visualization from matplotlib and other graphics.

The most common output prior to cell re-execution is rich output, such as dataframes. This is likely because dataframes were central to both tasks, and participants were primarily focused on transforming and validating these objects as part of their workflow. The second most common output was stream outputs, likely driven by exploration, although a significant portion of these texts could be related to debugging. Supporting this hypothesis is that more than 20% of outputs prior to re-execution are error messages. Lastly, the least common outputs were visualizations and empty outputs. Empty outputs likely represent the development process, where people are making blind code modifications, while visualizations are relatively rare, even in notebooks.

***Summary of RQ1**: In case of self-transitions, we find that most changes introduced in iterative work over singular cell are relatively small yet repeated multiple times in a cyclical manner. This may indicate that the interactive nature of notebooks influences the development process towards smaller experimental changes for debugging and exploration.*

7.2. ***RQ2:*** *The Purposes of Code Changes*

We analyzed the set of labels provided by GPT-4o for the 3,573 self-transitions. When we revised the sample of 400 transitions to come up with a final "ground truth" label, we established that GPT-4o showed a 0.55 overlap with the ground truth over these 400 transitions, in our 11-label task. We compute this overlap by calculating the number of transitions for which GPT-4o and the ground truth share at least one label in common. However, there are some non-overlapping answers for which the GPT-4o label could also be considered as appropriate. For instance, GPT-4o identified 327 "no change"-s, while technically, a string comparison revealed that only 306 cases were no-code changes. 7 additional transitions contained changes that could potentially be semantically considered a "no content change" (*e.g.*, removing or adding a newline character). The remaining 14 cases contained changes that can only be considered code changes.

Figure 5 shows the proportion of occurrence of the purpose labels. 8.8% of the labels refer to "no change". These transitions represent the cells that developers re-executed without editing the content. This could be a sign that the user needed to re-execute a cell due to technical problems, like an unsuccessful rendering of the output. As for the other labels, we can group them into three major categories: *code iteration* ("fix", "debug", "edit code", "clean code", "improve readability", "comment", and "uncomment"), *exploration* ("explore variable" and "visualize data"), and *further development* ("extend code"). Let us now go over them one by one, referring to Figure 5 in all cases.

Code iteration. This category is present in 72.2% of the labels. "Fix" (fixing syntactic errors) alone represents 16.8% of the label count. This means that developers needed to re-execute cells to identify the syntactic mistakes they made while writing their code. Such a result suggests that the support that these developers received in Jupyter might not be sufficient or effective in terms of Python/library code documentation and syntax control. The more

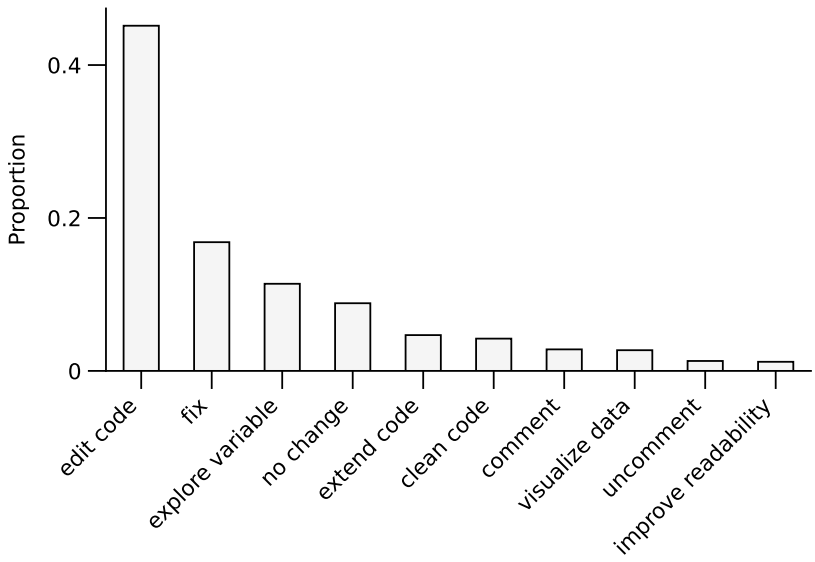

Figure 5: Proportion of labels provided by GPT-4o (as of the 10th of March, 2025) when asked to classify the goal that the developer pursued when implementing the code change(s) in each code transition. Note that some code transitions may include more than one change purpose and labels with frequency less then 1% were omitted from the plot.

lines and the more complexity the cells contain, the more inefficient this way of discovering errors becomes. This issue could be addressed by extending Jupyter with an assistant that helps developers auto-complete code and check syntax correctness on the fly, similarly to other IDEs. The "edit code" label also appeared with high frequency — in 45.1% of labels, pointing to the need that developers seem to have to iterate and gradually implement, as they need the execution feedback. The results also show that developers do "clean code" and change code to "improve readability", which indicates that they care about the final code quality. Given the competitive context of the hackathon, it makes sense to find instances of these labels.

Exploration. This category appears in 14.1% of the labels. The label "explore variable" occurred in 11.4% of the labels, while "visualize data" — in 2.7.%. This indicates that the features for inspecting variable content in Jupyter are not sufficient.

Further development. This category, including the single label "extend code", is present in 4.7% of the cases. The fact that developers need to re-execute

cells after adding new pieces of code shows that they need to gradually verify the performance of their code. This also calls for more IDE features in Jupyter notebooks.

> ***Summary of RQ2:*** *Based on the provided annotations, it is visible that the interactive nature of notebooks is often leveraged for code iteration purposes, including tasks for editing and fixing code. This result suggests a lack of adequate tools to support the code iteration in the notebook environment.*

7.3. ***RQ3:*** *Changes in Data Science Steps*

We first analyze when each data science step occurs, and then we study the transitions between data science steps during the development.

7.3.1. Stages of different data science steps

With the help of the generated data science labels, we investigate when each of the data science steps occurs in a notebook development process. We categorize the log data for a given notebook into ten quantiles sorted by time. In Figure 6, we plot the distribution of labels across the quantiles. Our results show that while some steps occur as expected (*e.g.*, *load_data* events occur more frequently in the first few quantiles, and *modelling* events occur more frequently in the last few quantiles), there are some surprising trends. We find that the *helper_functions* step, which is about importing libraries and defining custom functions, occurs fairly evenly throughout the development process, instead of happening at the start as one might assume. Similarly, *data_preprocessing* and *data_exploration* steps occur throughout the entire development process and not only towards the first half.

This provides additional support for the findings of Chen et al. [11] that the *data_preprocessing* and *data_exploration* steps are the most error-prone, since they constitute the majority of actions in realistic Jupyter Notebook development. Interestingly, their third most frequent error-related step, *load_data*, is rare in our data, suggesting that it is relatively difficult for users.

These results demonstrate that the process of development could be split into time-dependent steps like *load_data* or *modeling* and time-independent steps like *data_preprocessing* and *data_exploration*. This classification can aid in developing tools for notebooks. When constructing tools to support data exploration or Extract Method refactorings (to extract helper functions), it would be beneficial to design these tools for continuous assistance,

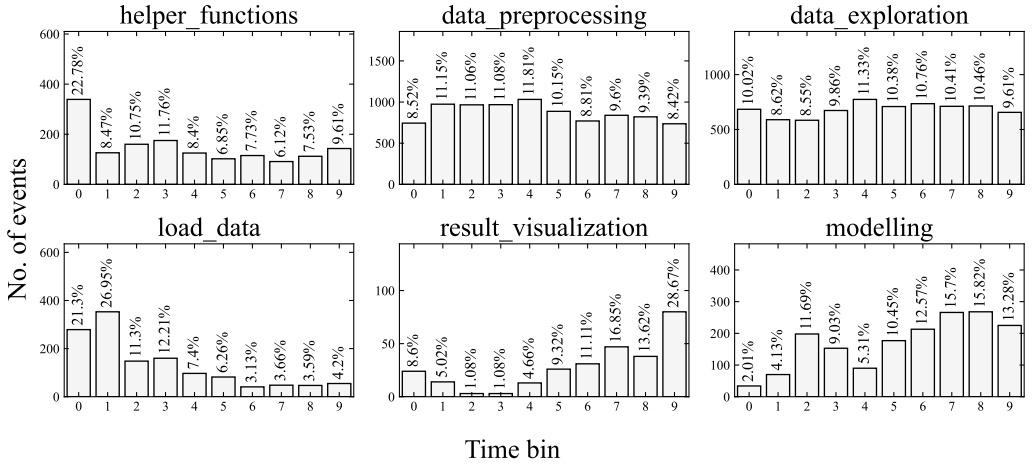

Figure 6: The presence of different data science steps depending on the quantile of the history of a notebook.

operating after each action in the notebooks. On the other hand, tools assisting with modeling or loading data could be invoked only several times when the corresponding action is detected.

7.3.2. Transitions between data science steps

To answer this question, we look at the transitions of data science steps between notebook cells. Based on the algorithm described in Section 6.3, we generate a transition matrix of probabilities for data science steps in a workflow. As stated in the methodology, we focus on both self-transitions and inter-transitions.

Self-transitions. The number of self-transitions in our data accounts for 39% of all executions. This confirms the cyclical nature of development in Jupyter notebooks, as users continuously iterate over the same cell and experiment with the code contained within it. We arranged self-transitions into transition probability matrices. The notable characteristic of the resulting matrices is their diagonal dominance, which indicates that—in the vast majority of instances—the data science label remains unchanged after re-executing the cell (*i.e.*, the step does not change across re-executions). This feature underscores both the stability of our machine learning annotation model and the infrequent need for users to carry out multiple stages in the same cell.

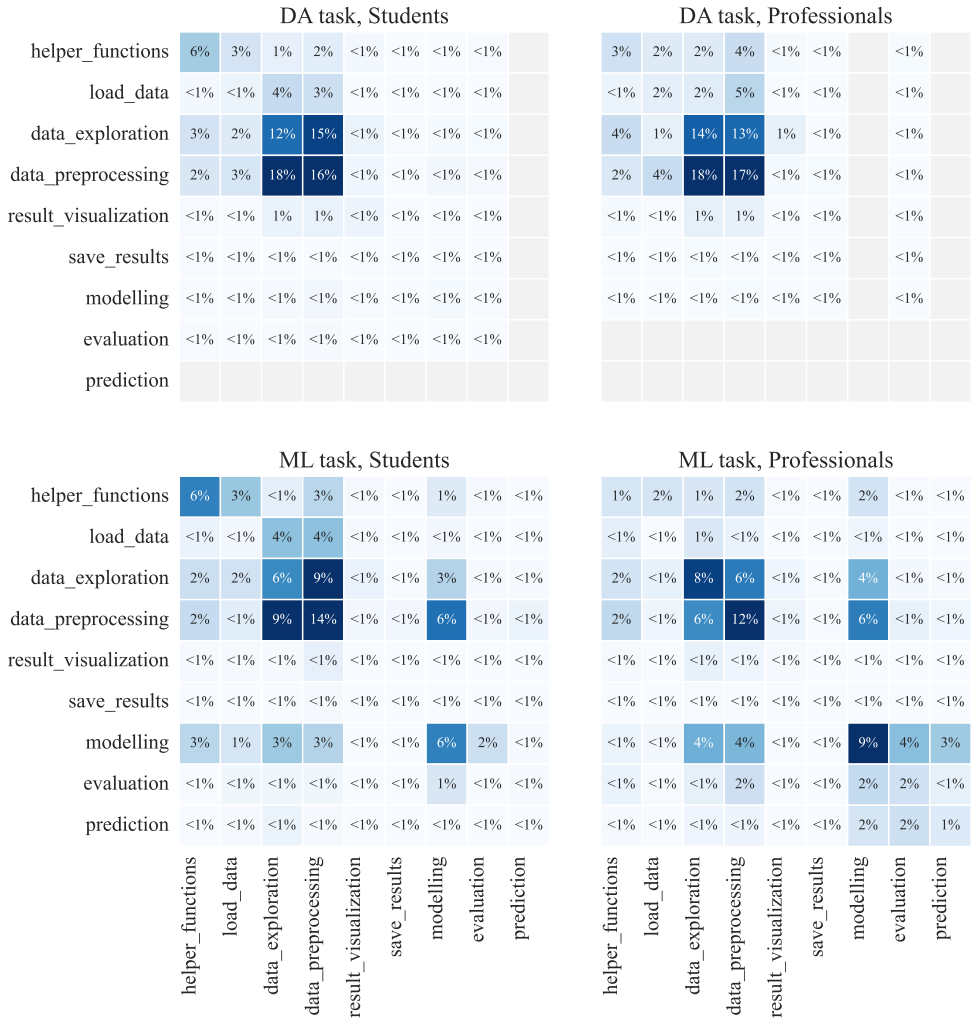

Figure 7: The transitional matrices for data science steps — only inter-transitions. Note that the matrices are normalized so that all the probabilities sum up to 1.

Inter-transitions. We continue our analysis by examining transitions between different cells. Figure 7 shows the transition matrices for different cells, normalized to sum up to 1. From the matrices, it is apparent that professionals transitioned more from data exploration to helper functions in the DA task. This could be an indication that professionals try to extract methods for data exploration into helper functions. For both tasks, a substantial part of transitions are tied between *data_exploration* and *data_preprocessing* labels (including transitions to cells with the same label) — 61% for the DA task and 35% for the ML task. This suggests that users engage in data exploration and data cleaning at different stages of the development process. We believe the transition analysis indicates that the main work in the notebook, regardless of the task, revolves around data manipulation.

> ***Summary of RQ3****: The analysis of the transition matrices between various data science steps indicates that, regardless of the task or level of expertise, core data science steps in the notebooks predominantly involve data exploration and pre-processing. Some steps are time-dependent, while others are time-independent. Data exploration and processing occur uniformly throughout the entire working period within the notebook. Conversely, other steps tend to occur more frequently towards the start or the end of the notebook development process.*

8. Discussion

8.1. Threats to Validity

The main potential challenge to the validity of our conclusions is the reliability of the annotation models. We employed our own model for the data science step annotation, and while it performs slightly better than the one proposed in the original paper, the F1 score is still not very high. The annotations for the change types were obtained using GPT-4o. Although the model is incredibly powerful, we cannot fully trust the results produced by this annotation. However, the conclusions drawn from this analysis are based on strong statistical effects, so we believe that the quality of the model does not significantly impact the results.

It is also important to note the sample size in our study. Although our data collection led to a considerable number of actions within notebooks, it was based on a relatively small sample size of 20 developers, due to the challenge of recruiting participants for user studies, especially those that

strive to be realistic and thus take many hours. While we made our best effort to balance the sample in terms of experience, one must be cautious when generalizing our results — they could be skewed.

Finally, this project involves several software components: a JavaScript plugin, a Python server, machine learning annotation, and complex data analysis. Because the solution is composed of multiple components written in different languages, the application may not be stable on newer versions of Jupyter Notebooks. We have released all of our code, datasets, and materials to the community so they can be used for future research, and we also provide pre-calculated results and intermediate steps for the supplied notebooks so the results can be reproduced and verified step by step [12].

Despite all these important limitations that grow from the novelty of our work, we believe that they do not invalidate the general high-level observations of our study.

8.2. Implications

From the results of the study, we can summarize the following implications and action points for notebook tool developers and researchers.

Jupyter Notebook developers need more code iteration tools. A significant amount of cyclical work in notebooks is related to code iteration, as users tend to iteratively edit and fix their code, leveraging the interactive nature of the notebooks and data-driven tasks. To assist with this and bring notebooks closer to their original goal of being a tool for exploration, we suggest adding more debugging tools. Features like IDE-style inspections, variable tracking, or even dedicated debugging cells that are excluded from the linear structure of the notebook could greatly enhance the user experience. There are already several Jupyter Notebook tools that support extended debugging experience [29, 30, 31]. However, like many Jupyter plugins, they suffer from a lack of popularity and therefore limited adoption. We hope that our study will help highlight the importance of such features, and that some of them may eventually be supported by the Jupyter community or by major IDE providers such as Microsoft or JetBrains.

The data science workflow is highly entangled. Our analysis shows that much of the workflow involves frequent transitions between exploration, preprocessing, and visualization. We argue that the current classification model does not accurately capture this process, as many steps occur only a few times or not at all, while the remaining steps are closely intertwined. Additionally, a significant amount of work happens within the development

of individual cells, which is not accounted for in the existing classification. To better understand the data science process, a more refined taxonomy is needed. The next iteration of the taxonomy and classification model should also account for developer-related variables, such as experience or background. Since we collected data from both professionals and students and labeled it accordingly, our dataset may provide a foundation for this line of work.

Snapshots of notebooks are not representative of the actual development process. Our work demonstrates that a significant portion of the development occurs within individual cells. This development cannot be fully analyzed using VCS snapshots of the notebook, which could miss many important insights about the process. We show that users interact with the code in various ways during cell development, and a deeper analysis of these interactions could lead to much better support for notebook development. Following the work of Chen et al. [11], we are extending the corpus of fine-grained Jupyter log data. The combined datasets may not only improve our understanding of the Jupyter Notebook development process, but also create new opportunities in the field of machine learning. More available data could enable more precise models of data science workflows and support models for next-edit suggestion. We hope that our tooling is used to collect even more datasets in the future.

9. Possible Future Research

In this section, we showcase several directions where one can go to investigate our data further. We also highlight some relevant results that did not fit our main research direction but that can instead be used as a foundation for future deeper exploration.

9.1. Temporal Analysis of the Workflow

One of the strengths of the dataset we collected is that it contains timestamps of each action. It could be interesting to take a look at how people spent their time during the development process. For example, we can calculate the amount of time spent on the execution state of cells, which accounts for an average of 8.96% of the total time spent in the notebook. Table 3 presents a summary of the time users spend in the execution state for each task, along with its percentage in relation to the overall task solving.

Task	Expertise	Execution time (s)		% of total time (%)	
		mean	std	mean	std
ML	Student	7.63	50.74	11.90	9.50
	Professional	5.47	46.53	14.21	13.35
	All	6.33	48.26	13.05	11.12
DA	Student	2.95	15.81	7.60	8.48
	Professional	1.22	9.20	2.94	2.58
	All	2.58	14.67	6.38	7.61
All	Student	4.05	28.26	8.89	8.78
	Professional	3.90	37.39	9.08	11.24
	All	3.99	31.84	8.96	9.54

Table 3: Descriptive statistics for the execution time and the percentage of execution time to total time.

Looking at the data, we can observe a difference in the average time taken for cell execution between the DA task (M=2.58, SD=14.68) and the ML task (M=6.33, SD=48.26), as well as between students (M=4.05, SD=28.26) and professionals (M=3.90, SD=37.39). This is an interesting finding that shows a difference in the way students and experts engage in various data science tasks. However, to understand the reasons that lead to such a difference would require a much deeper analysis. One could analyze the dataset further to identify the actions that take most of the time to execute, how the time between actions is distributed, or even predict the execution time from the cell source. All the details for the temporal analysis and several other features for time calculation can be found in supplementary materials [12].

9.2. Analysis of Code Dynamics

Another interesting topic to investigate further is how the code evolves during the development of the notebook. One could extend our analysis by looking at, for instance, how code complexity varies over time or identify the most frequent errors during the development of these notebooks. We calculated how the mean number of Python objects (nodes of the abstract syntax tree with a name and a value) per cell changed during the development of the notebook (see Figure 8). We can clearly see the difference in dynamics between the two tasks. We believe that there should be other interesting

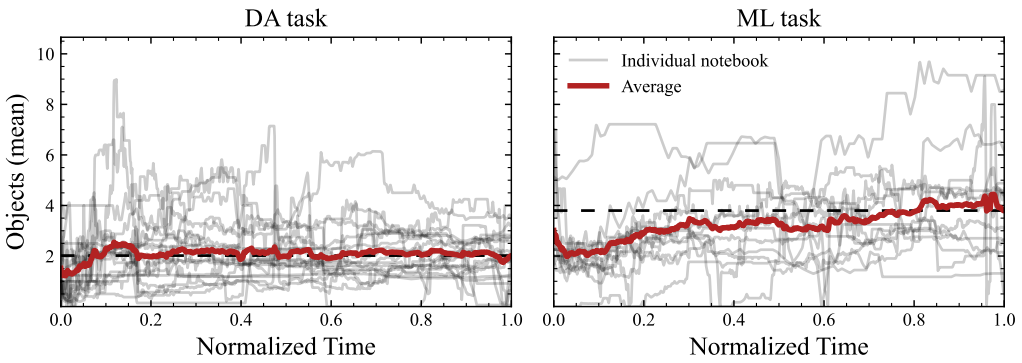

Figure 8: The evolution of the *Mean number of objects* metric in time for individual notebooks (grey lines) and their average curve (red line).

effects that require deeper investigation. One can pursue studying what errors frequently occur in the notebooks, what objects are most frequently changed, and how exactly people change code when they re-run multiple cells in a row. All of the details for code analysis and several other code-based features can be found in supplementary materials [12].

10. Conclusion

This study provides a unique perspective on the real-time operation of Jupyter notebooks. We demonstrate that analyzing fine-grained logs in notebooks can provide valuable insights for tool developers and researchers. By focusing on code changes, we reveal that a significant portion of user effort is directed towards debugging rather than exploration. This calls for work on improved debugging tools in Jupyter notebooks.

We further analyzed the development in terms of the framework modelling data science steps proposed by Ramasamy et al. [13], finding that the dynamics of real-time data science workflow in our study align with the proposed steps, but without necessarily following a linear structure.

Consequently, we propose that the development of tools and notebook features should prioritize supporting the process over merely focusing on the final form of the notebook. Embracing and facilitating the inherent nonlinearity of the development process, rather than attempting to counteract it, should be a key objective in future tools for Jupyter notebooks.

Data Availability

Our materials, code, the dataset, and the extended results can be found in our supplementary materials [12]. There, one can also find all the necessary tools to conduct further studies on the presented datasets, including those proposed in Section 9, as well as tools for collecting dynamic Jupyter notebook data and performing similar analyses. Additionally, the repository contains the code and intermediate results required to fully replicate the findings of the presented study.

Acknowledgments

This work was partially supported by the Swiss National Science Foundation through projects 'D3' (contract no. CRSII5_205975) and 'CrowdAlytics' (contract no. 200020_184994). A. Bacchelli gratefully acknowledges the support of the Swiss National Science Foundation through the SNSF Project 200021M_205146. We would also like to thank the hackathon participants for their valuable contribution.

References

[1] S. Negara, M. Vakilian, N. Chen, R. E. Johnson, D. Dig, Is it dangerous to use version control histories to study source code evolution?, in: European Conference on Object-Oriented Programming, Springer, 2012, pp. 79–103.

[2] C. Ko, M. Ruschitzka, K. Levitt, Execution monitoring of security-critical programs in distributed systems: A specification-based approach, in: Proceedings. 1997 IEEE symposium on security and privacy (Cat. No. 97CB36097), IEEE, 1997, pp. 175–187.

[3] M. D. Syer, Z. M. Jiang, M. Nagappan, A. E. Hassan, M. Nasser, P. Flora, Leveraging performance counters and execution logs to diagnose memory-related performance issues, in: 2013 IEEE international conference on software maintenance, IEEE, 2013, pp. 110–119.

[4] V. Bibaev, A. Kalina, V. Lomshakov, Y. Golubev, A. Bezzubov, N. Povarov, T. Bryksin, All you need is logs: Improving code completion by learning from anonymous IDE usage logs, in: Proceedings of the 30th

ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2022, pp. 1269–1279.

[5] E. Lyulina, A. Birillo, V. Kovalenko, T. Bryksin, TaskTracker-tool: A toolkit for tracking of code snapshots and activity data during solution of programming tasks, in: Proceedings of the 52nd ACM Technical Symposium on Computer Science Education, 2021, pp. 495–501.

[6] V. Kasatskii, A. Sergeyuk, A. Serova, S. Titov, T. Bryksin, The effect of perceptual load on performance within IDE in people with ADHD symptoms, in: International Conference on Human-Computer Interaction, Springer, 2023, pp. 122–141.

[7] J. F. Pimentel, L. Murta, V. Braganholo, J. Freire, Understanding and improving the quality and reproducibility of Jupyter notebooks, Empirical Software Engineering 26 (4) (2021) 65.

[8] K. Grotov, S. Titov, V. Sotnikov, Y. Golubev, 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, 2022, pp. 353–364.

[9] D. E. Knuth, Literate programming, The computer journal 27 (2) (1984) 97–111.

[10] D. Raghunandan, A. Roy, S. Shi, N. Elmqvist, 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, 2023, pp. 1–12.

[11] W.-H. Chen, J. L. Cheoh, M. Keim, S. Brunswicker, T. Zhang, Towards understanding fine-grained programming mistakes and fixing patterns in data science, Proceedings of the ACM on Software Engineering 2 (FSE) (2025) 1824–1846.

[12] S. Titov, K. Grotov, C. Sarasua, Y. Golubev, D. Ramasamy, A. Bacchelli, A. Bernstein, T. Bryksin, Supplementary materials, `https://doi.org/10.5281/zenodo.16098734`, [Online. Accessed 12-June-2026].

[13] D. Ramasamy, C. Sarasua, A. Bacchelli, A. Bernstein, Workflow analysis of data science code in public GitHub repositories, Empirical Software Engineering 28 (1) (2023) 7.

[14] S. Negara, N. Chen, M. Vakilian, R. E. Johnson, D. Dig, A comparative study of manual and automated refactorings, in: ECOOP 2013–Object-Oriented Programming: 27th European Conference, Montpellier, France, July 1-5, 2013. Proceedings 27, Springer, 2013, pp. 552–576.

[15] Y. S. Yoon, B. A. Myers, A longitudinal study of programmers' backtracking, in: 2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), IEEE, 2014, pp. 101–108.

[16] S. Negara, M. Codoban, D. Dig, R. E. Johnson, Mining fine-grained code changes to detect unknown change patterns, in: Proceedings of the 36th International Conference on Software Engineering, 2014, pp. 803–813.

[17] M. Hilton, N. Nelson, H. McDonald, S. McDonald, R. Metoyer, D. Dig, TDDViz: Using software changes to understand conformance to test driven development, in: Agile Processes, in Software Engineering, and Extreme Programming: 17th International Conference, XP 2016, Edinburgh, UK, May 24-27, 2016, Proceedings 17, Springer International Publishing, 2016, pp. 53–65.

[18] A. van der Wolk, The (im)possibilities of scientific research under the GDPR, Cybersecurity Law Report (2019).

[19] S. Titov, Y. Golubev, 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), IEEE, 2022, pp. 492–496.

[20] J. Wang, L. Li, A. Zeller, Better code, better sharing: On the need of analyzing Jupyter notebooks, in: Proceedings of the ACM/IEEE 42nd international conference on software engineering: new ideas and emerging results, 2020, pp. 53–56.

[21] J. F. Pimentel, L. Murta, V. Braganholo, 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), IEEE, 2019, pp. 507–517.

[22] M. Källén, T. Wrigstad, Jupyter notebooks on GitHub: characteristics and code clones, arXiv preprint arXiv:2007.10146 (2020).

[23] D. Wang, J. D. Weisz, M. Muller, P. Ram, W. Geyer, C. Dugan, Y. Tausczik, H. Samulowitz, A. Gray, Human-AI collaboration in data science: Exploring data scientists' perceptions of automated ai, Proceedings of the ACM on human-computer interaction 3 (CSCW) (2019) 1–24.

[24] M. Grinberg, Flask Web development, "O'Reilly Media, Inc.", 2018.

[25] M. Owens, G. Allen, SQLite, Apress LP New York, 2010.

[26] C. S. Bojer, J. P. Meldgaard, Kaggle forecasting competitions: An overlooked learning opportunity, International Journal of Forecasting 37 (2) (2021) 587–603.

[27] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, M. Zhou, CodeBERT: A pre-trained model for programming and natural languages, in: Findings of the Association for Computational Linguistics: EMNLP 2020, 2020, pp. 1536–1547.

[28] L. Prokhorenkova, G. Gusev, A. Vorobev, A. V. Dorogush, A. Gulin, CatBoost: Unbiased boosting with categorical features, Advances in neural information processing systems 31 (2018).

[29] J. Patra, M. Pradel, Nalin: Learning from runtime behavior to find name-value inconsistencies in Jupyter notebooks, in: Proceedings of the 44th International Conference on Software Engineering, 2022, pp. 1469–1481.

[30] K. Grotov, A. Borzilov, M. Krivobok, T. Bryksin, Y. Zharov, Debug smarter, not harder: AI agents for error resolution in computational notebooks, in: Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: System Demonstrations, 2024, pp. 363–371.

[31] C. Yang, R. A. Brower-Sinning, G. Lewis, C. Kästner, Data leakage in notebooks: Static detection and better processes, in: Proceedings of the 37th IEEE/ACM international conference on automated software engineering, 2022, pp. 1–12.