Exploring the Jupyter Ecosystem: An Empirical Study of Bugs and Vulnerabilities

Wenyuan Jiang
ETH Zürich
Zürich, Switzerland
wenyjiang@student.ethz.ch

Diany Pressato
Concordia University
Montreal, Canada
diany.pressato@mail.concordia.ca

Harsh Darji
University of Alberta
Camrose, Canada
hdarji@ualberta.ca

Thibaud Lutellier
University of Alberta
Camrose, Canada
lutellie@ualberta.ca

***Abstract*—Background. Jupyter notebooks are one of the main tools used by data scientists. Notebooks include features (configuration scripts, markdown, images, etc.) that make them challenging to analyze compared to traditional software. As a result, existing software engineering models, tools, and studies do not capture the uniqueness of Notebook's behavior.**

Aims. This paper aims to provide a large-scale empirical study of bugs and vulnerabilities in the Notebook ecosystem.

Method. We collected and analyzed a large dataset of Notebooks from two major platforms. Our methodology involved quantitative analyses of notebook characteristics (such as complexity metrics, contributor activity, and documentation) to identify factors correlated with bugs. Additionally, we conducted a qualitative study using grounded theory to categorize notebook bugs, resulting in a comprehensive bug taxonomy. Finally, we analyzed security-related commits and vulnerability reports to assess risks associated with Notebook deployment frameworks.

Results. Our findings highlight that configuration issues are among the most common bugs in notebook documents, followed by incorrect API usage. Finally, we explore common vulnerabilities associated with popular deployment frameworks to better understand risks associated with Notebook development.

Conclusions. This work highlights that notebooks are less well-supported than traditional software, resulting in more complex code, misconfiguration, and poor maintenance.

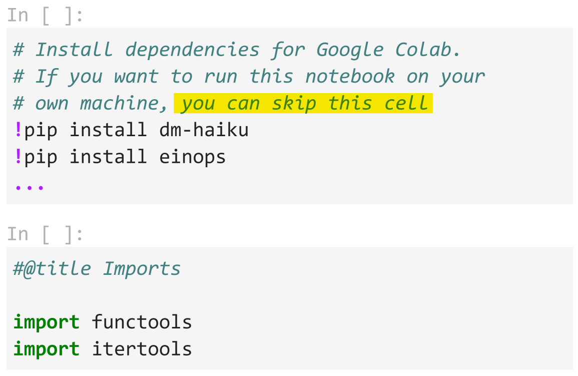
I. Introduction

Data science plays an increasingly pivotal role in driving global economic advancements. Positioned at the convergence of diverse scientific domains, data scientists, while adept in their respective scientific disciplines, often lack extensive familiarity with software development and reliability practices. This knowledge gap may inadvertently lead to the introduction of software bugs and vulnerabilities within data science applications. Such implications underscore growing concerns regarding the reliability, security, and overall dependability of data science software, emphasizing the need for a deeper understanding of software engineering principles within the data science community.

For example, 2022 saw the first ransomware attack on Jupyter Notebook [34], one of the most popular web applications for data science programming. The attack encrypted the users' notebooks and demanded a ransom payment for decryption. The success of this attack was attributed to misconfigured software security settings, emphasizing the critical importance of addressing and rectifying notebook bugs to safeguard against security breaches.

This work is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC-DG), grant No. 2024-00347.

Jupyter Notebook has become an industry standard for writing code related to data exploration, analysis, and machine learning [21]. These documents intertwine source code (e.g., Python, Matlab, or C) with descriptive markdown text that can include equations or media content. Furthermore, these documents showcase the dynamic outputs of the executed code, including variable values, tables, and graphs, providing a comprehensive and interactive platform for coding, documentation, and data visualization.



```
In [ ]:
# Install dependencies for Google Colab.
# If you want to run this notebook on your
# own machine, you can skip this cell
!pip install dm-haiku
!pip install einops
...

In [ ]:
#@title Imports

import functools
import itertools
```

Fig. 1. A Notebook where the first cell may be skipped.

Unlike traditional programming environments where code must be executed sequentially, Jupyter Notebooks allow users to run cells individually, enabling a more interactive and exploratory coding experience. Data scientists take full advantage of this feature as shown in Figure 1. In this real-world notebook from the Google DeepMind repository [11], developers create specific configuration cells that should only be run if working on Colab. If users work on their local machines, they should skip the execution of this cell.

The non-sequential execution capability of Notebooks, while proposing an interactive coding environment, presents a challenge for software engineering tools. These unique features, combined with the accessibility of notebooks to non-experts, increase the probability of bugs within notebooks.

In addition, even when proficient notebook users employ debugging tools, these solutions may fall short in addressing the distinctive characteristics inherent to notebooks. As a consequence, their efficacy in assisting users in enhancing notebook reliability may be compromised.

Despite recent efforts focused on reproducibility [38], [63], [64], [73] and the establishment of best practices [35], [40], [47], the notebook ecosystem is not yet well understood, and an in-depth study of notebooks and their associated bugs is needed. In particular, given recent security incidents involving notebooks [34], understanding the prevalence and nature of security vulnerabilities has become especially critical.

We address this gap through quantitative and qualitative analyses of notebook characteristics and changes. First, we gather a dataset of Notebook files and commits extracted from active GitHub repositories. We then quantitatively analyze the correlation between notebook characteristics and the frequency of bugs (Section IV), develop a taxonomy of notebook bugs (Section V), and analyze security-related commits and vulnerabilities to identify prevalent security issues (Section III-D).

This paper makes the following contributions:

Contribution 1: An empirical study of Jupyter Notebook characteristics and their correlation to bugs. We provide insights into factors most strongly associated with notebook bugs, guiding future tool development and best practices.

Contribution 2: A comprehensive taxonomy of Jupyter Notebook bugs derived from empirical analysis. This taxonomy helps researchers and practitioners better understand prevalent issues and their root causes.

Contribution 3: An analysis of security vulnerabilities within Jupyter Notebook deployment frameworks. We highlight critical security risks, emphasizing the need for robust security practices in notebook infrastructure.

II. Motivation & Background

A. Motivation

Despite the growing adoption of Jupyter Notebooks across data science, machine learning, and scientific computing, the software engineering community still lacks a comprehensive understanding of the quality challenges they introduce. Unlike traditional software, notebooks support nonlinear execution, mixing of code and documentation, dynamic outputs, and minimal testing infrastructure. These unique characteristics can lead to hard-to-detect bugs and poorly maintained projects. Moreover, the ease of use and accessibility of notebooks attracts a wide range of users, many of whom may lack formal software development training, increasing the likelihood of defects and misconfigurations. These challenges call for a systematic investigation of bugs within the notebook ecosystem.

To address this gap, we formulate three research questions. RQ1 explores which notebook characteristics correlate with higher bug frequency, aiming to identify risk factors and inform quality-assurance practices. RQ2 seeks to create a taxonomy of notebook-specific bugs through qualitative analysis, helping researchers and tool builders understand the types of errors that commonly arise in this environment. Finally, RQ3 investigates the security vulnerabilities present in notebook deployment frameworks, an increasingly urgent concern given recent high-profile attacks and the widespread use of notebooks in production pipelines. Together, these questions provide a view of notebook reliability, from structural contributors to systemic risks.

B. Background

A **Jupyter Notebook** is a document that enables users to write and execute code interactively, one cell at a time. In addition to code, users can include markdown text, equations, media, and hyperlinks—organizing content into either code cells or markdown cells. A typical Jupyter Notebook consists of three main components:

Jupyter Notebooks support multi-language code execution, allowing users to run **code cells** in any order. A code cell includes executable code and configuration scripts.

Markdown cells allow users to add text or media, enhancing code cells with explanations and visual context.

Jupyter notebooks are executable documents that include output. Each code cell has its **output cell** that may contain text, traceback contents, graphs, images, videos, audio clips, and any output generated from a code cell.

III. Methodology

To guide our investigation, we formulate the following three research questions: **RQ1:** Which characteristics of Jupyter Notebooks correlate with a higher frequency of bugs?, **RQ2:** What are the most common types of bugs in Jupyter Notebook documents? **RQ3:** What are the most common security vulnerabilities in Notebook deployment frameworks?

Our approach consists of four main steps, as illustrated in Fig. 2. First, we mine active Jupyter Notebook repositories to extract notebook characteristics and code changes (Section III-A). We then conduct an empirical study to identify code and project features that correlate with the presence of bugs, addressing RQ1 (Section III-B). To answer RQ2, we perform a qualitative analysis to develop a taxonomy of common bugs in Jupyter Notebooks (Section III-C). Finally, we investigate security issues in notebook deployment frameworks such as JupyterLab and JupyterHub, addressing RQ3 (Section III-D). Altogether, these steps provide a comprehensive view of bugs and vulnerabilities across the Jupyter Notebook ecosystem.

A. Data Extraction

For our empirical study, we chose two orthogonal sources of data. Our first source comes from open-source GitHub repositories, while our second source comes from the Kaggle platform. We selected GitHub because it hosts a vast and diverse array of notebook-based projects spanning numerous application domains, reflecting real-world development practices. Kaggle was chosen due to its competitive environment, which encourages high-quality documentation and robust coding practices, thus complementing our GitHub dataset by providing a different perspective on notebook usage.

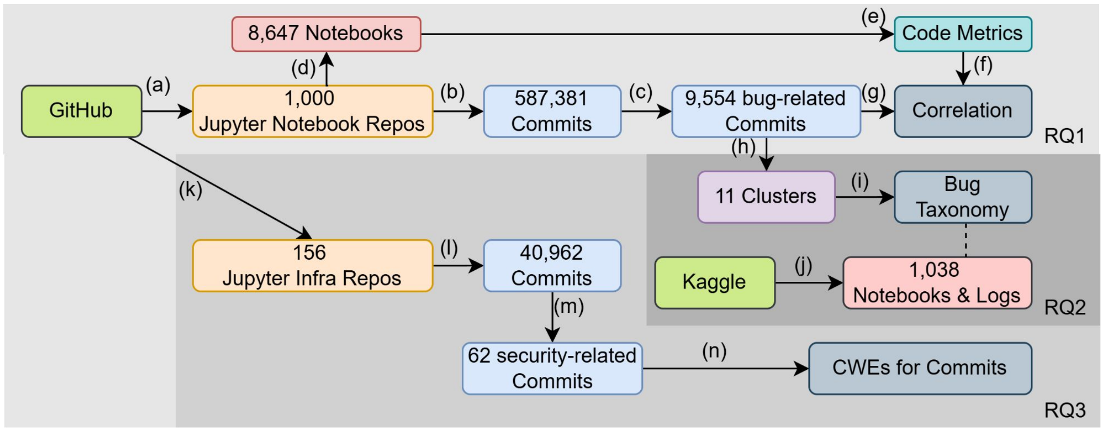


Fig. 2. Overview of the data collection and processing pipelines for each research question.

GitHub Project Selection: Using the GitHub API, we download the top 1,000 public repositories labeled with the "Jupyter Notebook" language tag, sorted by star ratings (Step (a) in Figure 2). Then, to increase the quality of our dataset, we removed inactive projects. We consider a project active if it has received a GitHub event (other than a watch event) within the last year (as of February 2024). Furthermore, since anyone can post personal projects on GitHub, and Jupyter Notebooks are used for a wide variety of less relevant projects, including books or blogs, two of the co-authors manually checked the README and the project descriptions of the repositories to discard non-development-related projects. This selection of relevant projects resulted in an inter-rater agreement of 88% and disagreements were settled by a third co-author. After this filtering process, we obtain 376 active Notebook repositories.

GitHub Notebook Characteristics and Change Extraction: To answer RQ1, we extracted the characteristics of Notebook files and projects (Steps (d) and (e)), including complexity metrics, natural language metrics, and contributor metrics.

Extracting changes in Jupyter notebooks is challenging due to the complexity of the notebook format. Unlike traditional code files, Jupyter notebooks store content in a structured manner that includes not only code but also rich text, images, and interactive elements. Moreover, Jupyter notebooks store metadata and cell outputs within the same file, complicating the differencing process.

Consequently, conventional differential algorithms (e.g., git diff) fail to isolate and represent changes in Jupyter notebooks. These algorithms tend to capture alterations in both structure and output, leading to substantial discrepancies. For instance, in cases where images are generated as output, the git diff algorithm may register thousands of lines as changed, reflecting the regeneration of a new image, even if only a few lines of code were modified.

To retrieve the code changes for each file in one commit, we extract the file before and after the corresponding commit. We then concatenate all Python source code cells in each notebook and parse them into a Python abstract syntax tree (AST). The ASTs are then serialized and compared. The results highlight changes in the code AST, which is more useful than standard line-based diff algorithms. Overall, we extracted 587,381 unique changes in Jupyter Notebook files.

Kaggle Notebook Selection: The Kaggle notebooks' selection was based on Kaggle's ranking system. We employed a systematic algorithm to extract notebooks and their latest log files from the first 20 pages (which roughly mapped to the 1,000 most popular files) of Kaggle's notebook repository. Overall, we gathered 1,038 Kaggle's competition notebooks and execution logs (Step (j)). These notebooks, submitted for competitive challenges, span a diverse range of applications, providing further insights into varied Jupyter notebook usage practices. Our focus on Kaggle competition notebooks was driven by the perceived higher quality in terms of code documentation and log file maintenance. This conclusion was drawn from an analysis of the substantial incentives offered in Kaggle competitions, including significant cash rewards and opportunities for developers to showcase their skills in machine learning and data science.

We further parse the logs, looking for the keywords 'Traceback' and 'Error' to identify bugs and map them back to code cells in the original Kaggle notebook. Finally, we extract the execution time of each file from the logs.

B. RQ1 Settings

Motivation: Understanding which characteristics of Jupyter Notebooks correlate with a higher frequency of bugs in Notebook files is essential for improving the reliability, reproducibility, and maintainability of data science workflows. Jupyter Notebooks are widely used in exploratory data analysis, machine learning model development, and research, but their interactive and flexible nature introduces challenges that increase the likelihood of bugs. These bugs may arise from factors such as code complexity, lack of documentation, lower developer experience, dependence on external libraries and data, or uniqueness of the domain of applications.

Identifying the specific characteristics that contribute to these issues allows for the development of better practices, tools, and guidelines to mitigate bug introduction, improving both individual productivity and collaboration in data science teams. By exploring these correlations, we can uncover patterns that are not only specific to Notebooks but may also apply to other interactive computing environments. We describe below how we identify bug-related commits and how we measure correlations between bugs and notebook features.

Identifying Bug-Related Commits: From the dataset of changes extracted from GitHub Notebook repositories (Section III-A), we identify bug-relate commits following previous work [28], [66] by considering a commit as bug-related if its related message contains any of the keywords “fix”, “bug”, “patch” but does not contain the keywords “rename,” “merge,” “clean-up,” or “refactor.” Overall, we extracted 9,554 bug-related commits (step (c)).

Correlation With General Project Characteristics: From our dataset of 8,647 notebooks and 587,381 unique changes, we extracted metadata and characteristics that may be correlated to bugs. Specifically, we investigate different sets of characteristics covering code complexity, natural language, and contributor metadata.

To assess code complexity, we examined five metrics including the number of functions, lines of code, libraries imported and code blocks in the Notebook, and the cyclomatic complexity of the file. These features were selected because they are recognized as indicators of code complexity, which is often linked to the likelihood of bugs in traditional software.

We also measure three natural language metrics, including code-to-markdown line ratio, length of commit messages, and the entropy of these messages. Code-to-markdown ratio is important as markdown serves as the primary medium for developers to incorporate natural language into Notebooks, impacting readability and maintenance. The length and entropy of commit messages, while not direct measures of clarity, provide indirect insights into how contributors document changes.

Finally, we look at the number of contributors working on the same file, the number of commits on a given file, and the age of their GitHub accounts at the time of contribution. The number of contributors working on the same file reflects the level of collaboration, which can introduce coordination challenges and potential inconsistencies. The age of the GitHub accounts of the contributors at the time of the contribution is a proxy for their experience [3], [13]. These metrics were chosen to capture collaboration and developer experience, factors frequently highlighted as influencing software quality. We measure the correlation between these metrics and the number of bugs in a given file, we use the Pearson Correlation.

C. RQ2 Settings:

Motivation: RQ2 aims at creating a taxonomy of bugs in Notebooks. While general software defect taxonomies exist, they often fail to account for the unique features of notebooks. By systematically categorizing common bugs in this environment, we aim to uncover patterns that are unique to notebook-based workflows. This is essential for building debugging tools, informing best practices, and supporting educational initiatives tailored to the notebook ecosystem.

We follow the grounded theory methodology to do a qualitative analysis of the bugs in Jupyter Notebooks. This process led us to the building of a Jupyter Notebook bug taxonomy. We describe below our main process.

Data Collection Stage: Our main data comes from the two sources as described in Section III-A. For this specific study, we only focus on the 587,381 GitHub notebook changes and on logs from Kaggle notebooks that contain any traceback.

Open Coding Stage: The open coding stage is a foundational step in our qualitative analysis, aimed at identifying distinct types of bug fixes in Jupyter Notebooks. To manage the large volume of commits and focus our manual analysis, we applied an automated clustering strategy to segment the dataset into meaningful groups of similar bug-fixing commits. This structured approach helped ensure diversity in the types of bugs we reviewed while maintaining feasibility.

We began by applying a mixed heuristic sampling process based on commit messages and AST-level code changes. Because Jupyter Notebooks are stored in JSON format, traditional line-based diff tools produce noisy results that include metadata and output differences irrelevant to actual code changes. For example, Listing 1 displays a change where a fix requires a single line change (swap between `input` and `input-1` parameters, lines 10 and 11). In the Notebook format, the diff is more complex to parse since the developer also changed the input file and this change resulted in the change of a very large tensor (in the data field) since such data is stored in the notebook. Additionally, metadata fields are also updated with the `execution_count` of every cell in the Notebook being updated. This makes correctly isolating bug-fixing changes very challenging in the Notebook environment. To address this, we extracted the Python source code cells info from the diff and used AST parsing to represent changes.

We then parsed these source code cells into abstract syntax trees (ASTs) and applied a structural diffing algorithm to identify semantic changes. This AST-based comparison reveals more meaningful transformations than line-based diffs—for example, distinguishing the insertion of control structures like an if condition, which might appear trivial in a line diff but indicates deeper logical modifications.

Listing 1. Example of a 1-line fix resulting in a change complex to parse.
```
"source": [
1 - "(Path(‘train/002844.jpg’), [‘train’])"
2 + "(Path(‘train/008663.jpg’),\n",
3 +      [‘car’, ’person’])
4 [...]
5 "data": { "text/plain": [
6 -    "tensor([-1.0028, [...], -3.6006],\n"
7 +    "tensor([ 2.0258, [...], 1.6073],\n"
8 [...]
9 "source": [
10 - " return  where(1-inputs, inputs).mean()"
11 + " return  where(inputs, 1-inputs).mean()"
12 [...]
```

To characterize each bug fix, we select eight metrics describing properties such as the size and type of changes, as well as boolean flags indicating whether the modification occurs within a loop, condition, constant, and so on. Principal Component Analysis (PCA) reduces these metrics to two principal components, which are then clustered with DBSCAN. This procedure initially yields 8 clusters that achieve a silhouette score [45] of 0.91, indicating high-quality clustering (where negative values suggest misclassified instances, 0 implies overlapping clusters, and 1 is the ideal value). To cover all relevant scenarios, 3 additional clusters are introduced to account for non-code edits (e.g., markdown cell updates), unparsable file changes, and commits that lack bug-fixing keywords. In total, we produced 11 clusters for further qualitative analysis.

Axial Coding: Three co-authors then each randomly sample 30 or 50 bugs from different categories (based on availability), manually analyze the changes, commit messages and associated issues or logs and come up with categories and descriptions for each bug.

Selective Coding and Saturation: After this initial sampling process, the three co-authors had an open discussion to consolidate their categorization and bug descriptions. Once the discussion is done, 30 or 50 additional bugs per person are sampled again. After three iterations and 230 bugs manually investigated, none of the authors found additional insights or categories, ending the search.

D. RQ3 Settings:

Motivation: Jupyter Notebook infrastructures such as JupyterHub, Jupyter Server, and JupyterLab have grown in popularity as core components of data science and interactive computing workflows. However, their widespread adoption also increases the threat surface for potential attacks. We observed when collecting data that the yearly count of Common Vulnerabilities and Exposures (CVEs) reported for software within the Jupyter Notebook ecosystem from 2015 to 2024 significantly increased. For example, the number of security reports in the NVD referencing Jupyter notebooks doubled between 2023 and 2024. This trend emphasizes the necessity of examining how security vulnerabilities manifest in real-world notebook deployment frameworks.

Security-Related Commit Selection: To study security issues in notebook deployment frameworks, we investigate all open-source repositories (156) from the three most popular GitHub organizations for notebook infrastructure: jupyter-hub, jupyter-server, and jupyterlab. We cloned the repositories and extracted all the commits in the main branch, resulting in a dataset of 40,962 commits. To obtain commits related to security issues, we use a two-stage filtering pipeline described below.

First, following previous work [72], we use regular-expression-based (regex) method (available in our replication package) on commit messages to filter out commits that are not relevant to security issues. To avoid including commits generated by automation tools, we also excluded commits that have a commit message longer than 1,000 characters. After this first stage, we obtained 400 security-related commits.

Regex filtering often leads to a relatively high false positive rate due to its limitation in understanding the semantics of the commit message. Inspired by the widespread use of large language models (LLMs) in software engineering, our second stage includes an LLM-based method for refining the results produced by the Regex filtering (full prompt in our replication package). We leveraged a state-of-the-art LLM, DeepSeek-V3 [9], for filtering due to its performance on software engineering tasks. The LLM is used to match a given commit message to an entry of the “2024 CWE Top 25 Most Dangerous Software Weaknesses” [33]. We used one-shot prompting on the LLM. The prompt included the description of the task, the output format, an example commit message and its paired expected output (a CWE ID), as well as the commit message to evaluate. Out of the previous 400 commits, 323 were matched with a CWE entry.

Hallucination is a problem of LLMs. To mitigate this issue, we manually checked the 323 commits for errors in the CWE mapping. We found that commits from one of the repositories (‘zero-to-jupyterhub-k8s’) only contained irrelevant commits from automatic vulnerability scans. After eliminating the automated commit scans, we ended up with 66 security-related commits. We further manually annotated to validate the LLM’s output according to the CWE website’s guidelines, reducing our total number of security-related commits to 62. After this annotation, we grouped the CWEs according to their parent pillar CWE in the CWE-1000 view to consolidate our results.

IV. RQ1 Results

To address RQ1—identifying characteristics of Jupyter Notebooks that correlate with a higher frequency of bugs—we analyzed the correlation between various notebook attributes and the number of bug-related commits.

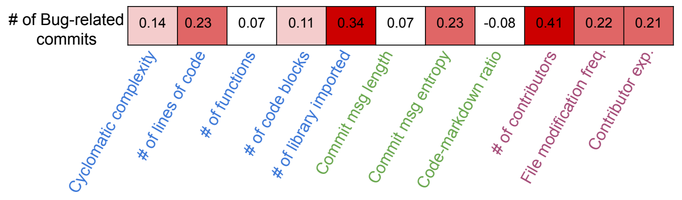


Fig. 3. Pearson Correlation coefficients between the number of bug-related commits and Notebook metrics. White, light red, medium red, and dark red indicate negligible, weak, medium, and strong correlations. Blue, green, and purple coding represent code-related, natural language, and metadata metrics.

Figure 3 presents the Pearson correlation values that illustrate the relationships between the different notebook characteristics and bug-related commits. Blue characteristics (e.g., Cyclomatic complexity) are code-related metrics, green ones are natural language metrics while the purple ones are metadata metrics related to file contributors. All the metrics describe file-level characteristics. For example, the “Commit msg length” measured the average commit message length of all commits modifying a specific notebook file. Pearson coefficients close to 1 (or -1) indicate a strong correlation between two features, while a coefficient close to 0 indicates a lack of correlation between features. For readability, we also colored the Pearson coefficients, with stronger red colors representing stronger correlations. For example, the first column of the Figure indicates that there is a weak positive (0.14) correlation between bug-related commits modifying a file and its cyclomatic complexity. Several key findings emerge from this analysis. We divide our analysis between code, natural language, and contributors metrics.

Code Metrics: Our results on code metrics are diverse. Files importing many libraries and large files are more likely to be modified by bug-related commits (Pearson coefficients of 0.34 and 0.23), reinforcing the idea that longer, complex files are more bug-prone. We found no correlation between function count and bug-related commits, indicating that the use (or non-use) of functions in Notebooks doesn't significantly impact the number of bugs in Jupyter Notebooks. The number of code blocks and cyclomatic complexity of the file play a slightly greater role (Pearson coefficient of 0.11 and 0.14) but still represent only a weak correlation.

Natural Language Metrics: Two of the natural language metrics (commit message length and code-to-markdown ratio) are not significantly correlated with bug-related commits. This is surprising as commit metadata has previously been shown to perform well for bug prediction [71] despite its simplicity. Code-to-markdown ratio not being significant tends to indicate that markdown plays a different role than code comments (i.e., it is not used to describe the code) as code comments in traditional software have been shown to be related to software quality and bug proneness [20]. This suggests that markdown cells, despite being another way to mix natural language and source code, do not replace the need for comments.

Contributor Metrics: Contributor metrics are the metrics most correlated with bug-related commits. Specifically, the number of contributors metric is strongly correlated, indicating that more contributors tend to modify buggy files, possibly due to increased complexity in coordination and merging changes. This high correlation compared to other metrics may also indicate that Notebooks are not as convenient for teamwork as more traditional source code. Similarly, frequently modified files are more prone to bugs, possibly due to instability from frequent changes. Finally, experienced users are more likely to work on files containing more bug-related commits, possibly because they are trusted with fixing issues.

RQ1 Summary: The main correlation on bugs is related to contributors' behavior and characteristics more than any code complexity or natural language metrics. This suggests that software engineering tools made for sharing Notebooks and working as a team (e.g., version control systems) are not well-suited for Jupyter Notebooks.

V. RQ2 Results: Bug Taxonomy

In this section, we follow grounded theory to generate a taxonomy of bugs' root causes in Jupyter Notebooks in order to understand the most common types of bugs present in the Notebook ecosystem. The resulting root cause categories of our study are described below.

Incorrect configuration: Notebooks share the same enormous and ever-growing ecosystem as Python does, and therefore share the tedious configuration process for many libraries. There are two major configuration problems that cause bugs. The first one is path-related. When users need to specify the location of resources, a path or url needs to be given. Without a structured configuration management tool, the common practice for a notebook is to manually fill in some path/url string, which may lead to problems when there is a change to the running environment. Code Snippet 2 is an example where a misconfigured path is updated.

Listing 2. Example of an Incorrect configuration bug
```
- path_data = '../../../data/'
+ path_data = '../../../assets/data/'
```

The other type of configuration bugs comes from versions of both the Python runtime and libraries. Unlike the package manager, npm, in the Javascript ecosystem, the default Python package manager, pip, does not force to specify a version when installing a library. This leads to incompatibility problems when some of the libraries a notebook depends on have undergone major updates. For example, one bug in a Colab notebook is caused by not specifying the Tensorflow version [57]. The fix depends on the 'magic command' feature provided by both the Jupyter Notebook ecosystem and the Colab environment, as is shown in Code Snippet 3.

Listing 3. Example of an Incorrect configuration bug
```
+ %tensorflow_version 1.x
```

Magic functions, also referred to as 'IPython magics', are specialized commands within Jupyter notebooks designed to offer convenient shortcuts and additional functionalities. Prefixed with a % symbol, these commands enable users to execute various operations beyond standard Python syntax. Throughout our research, we noticed frequent misuse of magic functions. This was often rooted in typographical errors, minor syntax mistakes, or the selection of an inappropriate magic function for a particular task. While magic functions provide a valuable means to enhance Jupyter notebook functionality, instances of misapplication can result in unintended errors, potentially compromising the overall integrity of the notebook.

Data shape/structure: One of the major use cases for Jupyter Notebooks is the data science field, where users need to manipulate different shapes of data, ranging from unstructured text, semi-structured JSON files to different shapes of matrices and tensors. As the core of notebooks, python provides various syntactic sugar for convenient manipulation of one or multiple data items in a very concise statement, which may take traditional languages like Java several or tens of statements to do. Third-party libraries like numpy and pandas made this coding style even more popular, which also complicates the semantics of these syntactic sugars. Bugs caused by incorrect handling of data structures are often seen. This is made worse by the lack of static type checking in Python interpreters. For example, a bug in a tutorial was caused by the misuse of certain shapes of tensors [68].

The fix of this bug, as is shown in Code Snippet 4, features the flexibility of Python syntax for tensors.

Listing 4. Example of a Data shape/structure bug
```
- mae = data.inverse_transform(
    ↪mae.reshape(1,-1))[0][0]
+ mae = data.inverse_transform([[mae]])[0][0]
```

API misuse: Notebooks span very diverse topics, with many domains relying on specific APIs such as TensorFlow, Pandas, website's REST APIs, etc. Many bugs in Jupyter Notebooks are related to incorrect usage of such API, resulting in incorrect function calls, missing or incorrect input parameters, or misunderstanding in return types.

Listing 5. Example of an Incorrect API bug
```
- return vsm_leaves_phi(text, yelp_lookup,
    ↪np_func)
+ return vsm_phi(text, yelp_lookup, np_func)
```

Additional instances of API misuse arise from customized methods created by Jupyter Notebook users, stemming from the absence of automated refactoring tools available to Notebook developers. Code Snippet 5 depicts an example of such a bug [5]. The developers altered the method's name from `vsm_leaves_phi` to `vsm_phi`. However, due to the lack of automatic refactoring tools, the author did not update all calls to this function. In Jupyter Notebooks, the presence of the original function name, `vsm_leaves_phi`, persists in memory until the user reruns the cell defining the function and no crash is observed. The issue will only manifest itself if a user reruns the modified cell.

Incorrect syntax: In this category, we observe struggles linked to Python's syntax for object-oriented programming, such as incorrect usage of the keyword self. This type of bug happened in the popular fast.ai framework [8]. This may be connected to the fact that Jupyter Notebook developers rarely use any classes or functions, and may not be comfortable with oriented object programming features.

Listing 6. Example of an Incorrect syntax bug
```
- self.opt.set_hyper(opt.hypers[0]['lr']*self.
    ↪mult_lr)
+ self.opt.set_hyper(self.opt.hypers[0]['lr']*
    ↪self.mult_lr)
```

Wrong logic: Code Snippet 7 shows an example of Wrong Logic in the HuggingFace notebook repository [27]. In the original implementation, the author missed cases where the answer span partially overlapped with the context.

Listing 7. Example of a Wrong logic bug
```
- if offset[context_start][0] > end_char \
-     or offset[context_end][1] < start_char:
+ if offset[context_start][0] > start_char \
+     or offset[context_end][1] < end_char:
```

Another type of "Wrong Logic" bug occurs when developers pick an incorrect algorithm. For example, we found a bug in the Google DeepMind repository [10] where the weights for computing the optical flow were incorrectly set up.

Non-determinism: Probabilistic features (e.g., the random standard library) are widely used in modern programs to implement simulation and machine learning algorithms. One common practice for ensuring reproducibility is to use pseudo-random functions and explicitly specify the random seed. However, due to the complexity of the Notebook ecosystem, even when users specify the random seed, there are cases when unexpected random and non-deterministic bugs happen.

Code Snippet 8 shows an example of a randomness bug from fastai2 [49]. When initializing the random number generator, the author of the notebook didn't set the seed according to an already-seeded random source, therefore causing the bug that even when users explicitly set a random seed, the results are not reproducible. To fix the bug, the contributors passed a random number generated from the user-defined seed as the seed for the new random number generator, making the new random number generator deterministic.

Listing 8. Example of a Random-related bug
```
- self.rng = random.Random()
+ self.rng = random.Random(
    ↪random.randint(0,2**32-1))
```

Exception/Error/Log/Debugging: The cell-based execution model of Jupyter Notebooks makes it difficult for users to reason about the current execution context when an exception happens. Also, the cell-based output mechanism adds complexity to the classical standard output mechanism assumed by Python, leading to traditional logging and debugging utilities that are difficult to use correctly.

Errors in Test Code and Assertions: The flexibility of cell execution and the convenience of result visualization in Jupyter Notebooks make test frameworks neither necessary nor compatible. One practice of testing a piece of code in notebooks is by creating another cell or directly adding assertion statements to the target code. This simple testing method could be used incorrectly by users and cause bugs.

Resource management: These bugs encompass issues related to the allocation and utilization of computing resources, such as the number of GPUs allocated, input sizes, and batch sizes. Improper handling of these aspects can lead to suboptimal execution, performance bottlenecks, or even system failures. For instance, we found a bug in the fastai2 [48] repository where developers allocated an excessively large input size, overwhelming available memory, causing the notebook to crash. Identifying and addressing these issues enhances the reproducibility of analyses and contributes to the overall robustness of Jupyter Notebooks.

Incomplete code: There are several instances of incomplete code in Jupyter Notebooks. Unlike traditional programming languages, incomplete code in notebooks often arises from educational or tutorial contexts. However, we consider incomplete code to be a bug because it leads to runtime errors or prevents users from properly executing or understanding subsequent notebook cells. Thus, incomplete code negatively impacts notebook functionality and usability, warranting its inclusion in our bug taxonomy and qualitative analysis.

Undeclared variables & Typos: Many bugs we found would be considered minor, yet ended up in Jupyter Notebooks of popular organizations such as fast.ai, or in the ”Data Science on AWS” textbook. These bugs highlight the lack of static analysis and linting in the default Notebook web interface.

Documentation mistakes: Mistakes in markdown may be considered bugs since they will affect the understanding of the complete document. For example, a markdown cell may describe a specific mathematical formula that is then implemented in Python. If that formula is incorrectly described, this is a bug as the document becomes incorrect. This is analogous to mistakes in comments being bugs [53], [55], [56], [69].

Kaggle Notebooks and Logs Analysis: In inspecting 1,038 Kaggle notebooks, 89 instances (9%) exhibiting traceback contents were identified, signifying errors within the logs. These errors were categorized into four primary types: Compiler Error (59 notebooks), Logic Error (5 notebooks), Configuration Error (20 notebooks), and API Misuse (5 notebooks).

Incorrect configuration errors underscore the importance of configuration management, including dependencies, environment settings, file paths, and configuration errors. **Kaggle Compiler errors** are often attributed to syntax issues or runtime anomalies within the codebase. **API errors** arise from issues with interfacing external services or libraries. Finally, **Logic errors** are caused by flawed implementations.

These findings from Kaggle align closely with our GitHub-based taxonomy of bugs. In both datasets, configuration issues and API misuse were among the most frequently observed root causes. While the Kaggle dataset contained additional compiler errors, both platforms demonstrated the presence of logic errors, reinforcing the generality of these bug categories. This overlap validates the robustness of our taxonomy across diverse notebook usage contexts and highlights consistent pain points for notebook users regardless of the platform.

RQ2 Summary: The most common bug root causes in Jupyter Notebooks fall into 12 main categories, with incorrect configuration, data shape mistakes, API misuse, incomplete code, wrong logic, and documentation errors being the most common root causes. Kaggle-based observations confirm our observations on GitHub with configuration issues, API misuse, and wrong logic being the main root causes of bugs.

VI. RQ3 Results: Notebook Security

RQ3 investigates security issues in Jupyter Notebooks. While we didn’t find any security issues in Notebook documents, deployment frameworks contain security vulnerabilities that might make running Notebook unsafe. For example, CVE-2024-43805 refers to a security vulnerability in the Jupyter-lab deployment environment where opening a maliciously crafted notebook allows an attacker to perform a cross-site scripting attack. Below, we present our findings on security-related commits in Jupyter Notebook infrastructures, drawing on the methodology described in Section III-D.

TABLE I
Number of commits related to each CWE pillar entry.

# of Commits	Common Weakness Enumeration ID (CWE ID)
19	CWE-693: Protection Mechanism Failure
17	CWE-284: Improper Access Control
12	CWE-664: Improper Control of a Resource Through its Lifetime
8	CWE-710: Improper Adherence to Coding Standards
4	CWE-707: Improper Neutralization
1	CWE-691: Insufficient Control Flow Management
1	CWE-703: Improper Check or Handling of Exceptional Conditions

Table I summarizes the number of commits under each CWE pillar in the CWE-1000 view, with each CWE pillar item indicated by its CWE ID and name. For example, the first line of the table indicates that we found 19 commits in Jupyter Notebook deployment frameworks that fix vulnerabilities related to CWE-693, Protection Mechanism Failure.

The frequency of CWE types in the selected commits follows a long-tail distribution. The most frequently observed CWE was Protection Mechanism Failure (CWE-693), followed by Improper Access Control (CWE-284) and Improper Control of a Resource Through Its Lifetime (CWE-664). These CWEs often arise in web-based systems and align with the Jupyter Notebook architecture, which relies on HTTP services and browsers. Less frequent CWEs include Command Injection (CWE-77) and Use of Known Vulnerable Component (CWE-1395), both of which are also typical in web services.

Listing 9. Example of CSRF bug in the jupyterhub project
```
jupyterhub/handlers/base.py
+ clear_xsrf_cookie_kwargs = {
+     key: value for key, value in
+     self.settings.get('xsrf_cookie_kwargs',
    {})
+     if key in {"path", "domain"}}
+     }
self.clear_cookie('_xsrf',
-     **self.settings.get('xsrf_cookie_kwargs',
    {}),
+     **clear_xsrf_cookie_kwargs, )
```

Within Protection Mechanism Failure, the most recurrent vulnerability concerns Cross-Site Request Forgery, accounting for 14 of the 19 relevant commits. Listing 9 shows an example of such a bug in the JupyterHub framework.

Listing 10. An Access Control bug in the nbgitpuller project
```
nbgitpuller/handlers.py
class LegacyInteractRedirectHandler(
    ↪IPythonHandler):
+    @web.authenticated
     def get(self):
```

Access control is also a significant bug in notebook infrastructure. Improper Access Control (CWE-284) was often addressed by bolstering authentication for notebook infrastructures that were previously unsecured. For example, a vulnerability in the nbgitpuller was fixed by "making sure that all endpoints are authenticated", which improves the missing access control for notebook infrastructure (See Listing 10).

Other web-based vulnerabilities included Cross-Site Scripting (XSS) due to insufficient input sanitization (CWE-707). Listing 11 shows an example of fixing an XSS vulnerability. Here, the addition of the "autoescape" option will automatically escape special characters, preventing the vulnerability.

Listing 11. Example of an XSS bug in the notebook project

```
notebookapp.py
- jenv_opt = jinja_env_options
+ jenv_opt = {"autoescape": True}
+ jenv_opt.update(jinja_env_options
```

These findings are consistent with recent CVE reports on the Notebook infrastructures. Among the 60 CVE reports we collected from 2015 to 2024, the majority of HIGH or CRITICAL rated vulnerabilities relate to web or access control issues, highlighting similar security concerns observed in our commit analysis. CVE reports also give insights beyond the infrastructure repos in our study, as many CVEs are rooted in the ecosystem of Jupyter Notebooks like servers and plugins.

Listing 12. Security Relax for Single User Server

```
jupyterhub/singleuser.py:
+ @classmethod
+ def validate_security(cls, app,
    ssl_options=None):
+       return
```

Trade-off between usability and security: Most security-related commits involved fixes for discovered vulnerabilities. However, a subset of commits relaxed security settings to simplify configuration in common single-user scenarios. For instance, Listing 12 displays a security change [32] that overwrites the validate_security method to suppress TLS-related security warnings for single-user servers. These changes are not advisable for publicly accessible deployments; however, they address usability challenges often encountered by individual or non-professional users. In the examples above, both TLS and cross-origin configuration are known to be difficult and messy in a non-standard web production scenario. These commits make configuration easier for out-of-the-box use of notebooks with a trade-off between usability and security.

Security implications: Although Jupyter Notebook infrastructures were originally designed for single-user, local usage, our analysis shows that they now face a range of common web-related vulnerabilities (e.g., CSRF, XSS). These issues, which stem from an HTTP-based architecture, can be exploited more readily in scenarios where notebooks are exposed to broader networks or multi-user environments. In practice, many organizations deploy notebook servers on corporate networks, in the cloud, or as part of shared platforms—environments that expand the attack surface beyond an individual's machine.

One key tension is the need to balance usability and security. A subset of security-related commits intentionally relaxes security controls to address the complexity of managing TLS certificates or cross-origin settings, particularly when notebooks are used on non-standard platforms or by non-professional users. While these relaxed settings simplify local installation and reduce support overhead, they pose considerable risks if the same configurations are adopted in production. For instance, disabling strict authentication or cross-origin checks could enable unauthorized access or session hijacking if the server is exposed to a public network.

These findings emphasize that notebook infrastructures are not intended for large-scale or security-critical contexts. Security-conscious deployments should implement robust authentication, enforce TLS for all traffic, apply strict cross-origin policies, and keep dependencies updated. Additional measures such as containerization or sandboxing can further restrict the scope of potential exploits. Nevertheless, since many Jupyter installations remain local, developers are facing a dilemma: adding robust security controls often introduces additional complexity, which can harm the out-of-the-box experience that makes Jupyter Notebook usable.

RQ3 Summary: Jupyter Notebook infrastructures are prone to common web-based vulnerabilities. We also observed a tension between usability and security, where certain commits relaxed default protection settings to improve ease of deployment, especially in single-user scenarios.

VII. Threats to Validity

Conclusion validity: Bugs can be subjective without a clear specification of how a piece of program should behave. Therefore, there is a potential threat regarding the identification and classification of the bugs in our study. To mitigate this issue, the classification was done in a systematic process in multiple iterations and verified by several authors.

External validity: Despite the large size of our datasets, the notebooks used in our evaluation come from a relatively homogeneous source (e.g., GitHub and Kaggle). Thus, the results on notebooks from other sources (e.g., company internal repositories or specialized communities) might be different. However, the data set should be reasonably representative for most real-world Jupyter notebooks given that GitHub and Kaggle are two major platforms for hosting notebook-related resources and they contain a wide range of topics.

Construct validity: If one category of certain bugs escaped our iterative sampling, then our approach would fail to include this bug in our bug taxonomy. We mitigate this issue by resampling the data for our manual analysis until saturation.

VIII. Related Work

Empirical Studies on Jupyter Notebooks: Much work has been done investigating software engineering practices for data science in Jupyter Notebooks.

The closest work is the concurrent study done by De Santana et al. [46] that analyzes Notebooks from GitHub and Stack Overflow posts, and conducts interviews with Jupyter developers in order to develop a taxonomy of problems related to Jupyter Notebooks. While their study is thorough, it focuses on higher-level issues encountered by developers (e.g., the kernel crash, the notebook cannot be converted to another format) while we focus on bugs in the source code (e.g., undeclared variable, error in test code, incorrect API code). Our lower-level taxonomy complements previous work well, and the connection to actual source code makes it more actionable than the previous taxonomy. In addition, while they filter tutorials and projects related to courses or books, this was done automatically, leading to potentially mislabelled repositories while we manually went through over 1,000 notebook repositories to ensure the quality of our dataset. Finally, our inclusion of Kaggle Notebooks, an independent secondary source of Notebooks further complements De Santana et al. [46] contributions.

Previous work [12] presents a qualitative study of cleaning activities in Jupyter Notebooks (adding, removing cells, etc.). [24], [43] analyses 2.7 million Jupyter notebooks hosted on GitHub and found that 70% of code snippets were clones and 50% of Notebooks have no unique code snippets.

Grotov et al. [15] quantitatively compare Python scripts and Jupyter Notebooks, finding that Notebooks have more stylistic issues—suggesting different developer behaviors and the need for Notebook-specific studies. In contrast, Adams et al. [1] report that Notebooks used in machine learning have fewer stylistic issues than Python scripts. Nalin [36] explores variable name/value inconsistencies in Notebooks using AI and dynamic analysis. Pimentel et al. [37], [38] and Wang et al. [63], [64] focus on Notebook reproducibility, proposing tools and techniques to detect and resolve related issues. Unlike these studies, our work provides an analysis of bugs and security vulnerabilities in the Jupyter Notebook ecosystem, combining both quantitative and qualitative perspectives.

Quaranta et al. [40] interviewed Notebook developers and studied best practices in 1,380 Jupyter Notebooks from Kaggle. They found that experts are generally aware of best practices (using version control, testing, etc.) but inconsistently apply them. Settewong et al. [47] investigated how visualization is used in competition notebooks to explain coding solutions and proposed a taxonomy of 9 types of visualizations used by expert notebook users. Chattopadhyay et al. [6] identify 9 pain points of computational notebooks. These pain points are not directly related to bugs in the code and do not overlap with our taxonomy. Finally, Van Binsbergen et al. [60] survey Read-eval-print-loops principles, which are used by computational notebooks.

Jupyter Notebooks tools and datasets: Quaranta et al. proposed a large dataset of Jupyter Notebook, KGTorrent [39] to help researchers. Pynblint [41], NBLyzer [51], and Julynter [38] are existing static analysis tools for Jupyter Notebooks. While being a step in the right direction, they are still lacking most standard linting features. Merino et al. [31] discuss the possibility for software engineers to develop widgets to increase users' access to the internal states of the executed notebook. Other extensions proposed in previous work [7] help determine which cells should be migrated, reducing the notebook's states and increasing performance.

Bug Taxonomy: Many bug taxonomies have been proposed, focusing on different ecosystems such as autonomous vehicle bugs [14], [62], deep learning systems [19], [22], [50], [52], JavaScript [16], [17], HTML [29], video games [26], Python API [18], [25], Data Analytics [2], test code [58], blockchain [61], internet of things [30], infrastructure as code [42], open-source software [4], [54], [59], [70], compiler [44], regular expressions [65] and security bugs [23], [67]. While they follow a similar process as ours, all these taxonomies are in widely different domains, making these works very different from the current paper.

IX. Implications & Conclusion

Practical Implications. Our study offers several practical insights for software developers, data scientists, and AI engineers involved with Jupyter Notebooks. First, given that configuration-related issues were found to be among the most common bugs, practitioners should adopt rigorous configuration management strategies, including clearly specifying library versions and paths, maintaining environment files, and leveraging containerization solutions such as Docker. Integrating notebook-specific linting and static analysis tools can further reduce frequent mistakes, such as API misuse or typos.

From a research perspective, our taxonomy of notebook bugs and the insights derived from analyzing vulnerabilities highlight the need for further studies focused on tailored software engineering methodologies specifically suited for computational notebooks. Future research could explore effective notebook-specific debugging techniques, develop more sophisticated static analysis tools that understand the notebook's cell-based execution model, or investigate notebook-friendly refactoring methods. Lastly, given the rising prevalence of security vulnerabilities within notebook deployment frameworks, researchers should prioritize developing notebook infrastructure that better balances usability with security, perhaps through context-aware adaptive security mechanisms or enhanced user awareness and education initiatives.

Conclusion. We proposed a large-scale empirical study, including a quantitative analysis of the notebook ecosystem, a qualitative analysis of bugs in notebook documents, and a study of vulnerabilities in notebook frameworks. We studied 8,647 Jupyter notebooks from GitHub and 1,038 notebooks from Kaggle. Based on our analysis, we deduce that configuration issues and API misuse were two of the most common errors that notebook users faced and presented a new taxonomy for bugs faced by users working in Jupyter notebooks.

Overall, our work highlights that attractive features of Notebooks such as interactivity come at a cost, increasing configuration issues and raising concerns about the reproducibility, maintainability, and security of notebook projects.

Data Availability: Our replication package and dataset are available on our anonymous GitHub[1].

[1]https://github.com/jwyjohn/Exploring-the-Jupyter-Ecosystem

References

[1] Adams, K., Vilkomir, A., Hills, M.: A comparison of machine learning code quality in python scripts and jupyter notebooks. Journal of Computing Sciences in Colleges **39**(5), 96–108 (2023)

[2] Ahmed, S., Wardat, M., Bagheri, H., Cruz, B.D., Rajan, H.: Characterizing bugs in python and r data analytics programs. arXiv preprint arXiv:2306.08632 (2023)

[3] Bao, L., Xia, X., Lo, D., Murphy, G.C.: A Large Scale Study of Long-time Contributor Prediction for Github Projects. IEEE Transactions on Software Engineering **47**(6), 1277–1298 (2019)

[4] Catolino, G., Palomba, F., Zaidman, A., Ferrucci, F.: Not all bugs are the same: Understanding, characterizing, and classifying bug types. Journal of Systems and Software **152**, 165–181 (2019)

[5] cgpotts: cs224u, commit d582e8d. https://github.com/cgpotts/cs224u/commit/d582e8d057543fb4972c642007db104a15914e07 (2021). Accessed: July 2025

[6] Chattopadhyay, S., Prasad, I., Henley, A.Z., Sarma, A., Barik, T.: 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, pp. 1–12 (2020)

[7] Cunha, R.L., Real, L.C.V., Souza, R., Silva, B., Netto, M.A.: Context-aware execution migration tool for data science jupyter notebooks on hybrid clouds. In: IEEE 17th International Conference on eScience, pp. 30–39. IEEE (2021)

[8] cwza: fastai2, commit 9cecf81. https://github.com/fastai/fastai2/commit/9cecf8192a232176a9532b785de22bcc50a09865 (2020). Accessed: July 2025

[9] DeepSeek-AI, Liu, A., Feng, B., Xue, B., Wang, B., Wu, B., Lu, C., Zhao, C., Deng, C., Zhang, C., Ruan, C., Dai, D., Guo, D., Yang, D., Chen, D., Ji, D., Li, E., Lin, F., Dai, F., Luo, F., Hao, G., Chen, G., Li, G., Zhang, H., Bao, H., Xu, H., Wang, H., Zhang, H., Ding, H., Xin, H., Gao, H., Li, H., Qu, H., Cai, J.L., Liang, J., Guo, J., Ni, J., Li, J., Wang, J., Chen, J., Chen, J., Yuan, J., Qiu, J., Li, J., Song, J., Dong, K., Hu, K., Gao, K., Guan, K., Huang, K., Yu, K., Wang, L., Zhang, L., Xu, L., Xia, L., Zhao, L., Wang, L., Zhang, L., Li, M., Wang, M., Zhang, M., Zhang, M., Tang, M., Li, M., Tian, N., Huang, P., Wang, P., Zhang, P., Wang, Q., Zhu, Q., Chen, Q., Du, Q., Chen, R.J., Jin, R.L., Ge, R., Zhang, R., Pan, R., Wang, R., Xu, R., Zhang, R., Chen, R., Li, S.S., Lu, S., Zhou, S., Chen, S., Wu, S., Ye, S., Ye, S., Ma, S., Wang, S., Zhou, S., Yu, S., Zhou, S., Pan, S., Wang, T., Yun, T., Pei, T., Sun, T., Xiao, W.L., Zeng, W., Zhao, W., An, W., Liu, W., Liang, W., Gao, W., Yu, W., Zhang, W., Li, X.Q., Jin, X., Wang, X., Bi, X., Liu, X., Wang, X., Shen, X., Chen, X., Zhang, X., Chen, X., Nie, X., Sun, X., Wang, X., Cheng, X., Liu, X., Xie, X., Liu, X., Yu, X., Song, X., Shan, X., Zhou, X., Yang, X., Li, X., Su, X., Lin, X., Li, Y.K., Wang, Y.Q., Wei, Y.X., Zhu, Y.X., Zhang, Y., Xu, Y., Xu, Y., Huang, Y., Li, Y., Zhao, Y., Sun, Y., Li, Y., Wang, Y., Yu, Y., Zheng, Y., Zhang, Y., Shi, Y., Xiong, Y., He, Y., Tang, Y., Piao, Y., Wang, Y., Tan, Y., Ma, Y., Liu, Y., Guo, Y., Wu, Y., Ou, Y., Zhu, Y., Wang, Y., Gong, Y., Zou, Y., He, Y., Zha, Y., Xiong, Y., Ma, Y., Yan, Y., Luo, Y., You, Y., Liu, Y., Zhou, Y., Wu, Z.F., Ren, Z.Z., Ren, Z., Sha, Z., Fu, Z., Xu, Z., Huang, Z., Zhang, Z., Xie, Z., Zhang, Z., Hao, Z., Gou, Z., Ma, Z., Yan, Z., Shao, Z., Xu, Z., Wu, Z., Zhang, Z., Li, Z., Gu, Z., Zhu, Z., Liu, Z., Li, Z., Xie, Z., Song, Z., Gao, Z., Pan, Z.: Deepseek-v3 technical report (2024). URL https://arxiv.org/abs/2412.19437

[10] Ding, D.: deepmind-research, commit 8cc5c39. https://github.com/google-deepmind/deepmind-research/commit/8cc5c3966282fc677b032e8c67f1b89458e6d47a (2021). Accessed: July 2025

[11] Ding, D., diegolascasas: Deepmind research, optical_flow.ipynb. https://github.com/google-deepmind/deepmind-research/blob/8cc5c3966282fc677b032e8c67f1b89458e6d47a/perceiver/colabs/optical_flow.ipynb (2021). Accessed: July 2025

[12] Dong, H., Zhou, S., Guo, J.L., Kästner, C.: Splitting, renaming, removing: A study of common cleaning activities in jupyter notebooks. In: 36th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW), pp. 114–119. IEEE (2021)

[13] Eluri, V.K., Mazzuchi, T.A., Sarkani, S.: Predicting Long-time Contributors for Github Projects Using Machine Learning. Information and Software Technology **138**, 106616 (2021)

[14] Garcia, J., Feng, Y., Shen, J., Almanee, S., Xia, Y., Chen, A.Q.A.: A comprehensive study of autonomous vehicle bugs. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, pp. 385–396. ACM (2020). DOI 10.1145/3377811.3380397. URL https://dl.acm.org/doi/10.1145/3377811.3380397

[15] Grotov, K., Titov, S., Sotnikov, V., Golubev, Y., Bryksin, T.: A large-scale comparison of python code in jupyter notebooks and scripts. In: Proceedings of the 19th international conference on mining software repositories, pp. 353–364 (2022)

[16] Gyimesi, P., Vancsics, B., Stocco, A., Mazinanian, D., Beszédes, Á., Ferenc, R., Mesbah, A.: Bugsjs: a benchmark and taxonomy of javascript bugs. Software Testing, Verification And Reliability **31**(4), e1751 (2021)

[17] Hanam, Q., Brito, F.S.d.M., Mesbah, A.: Discovering bug patterns in javascript. In: Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering, pp. 144–156 (2016)

[18] Hu, M., Zhang, Y.: An empirical study of the python/c API on evolution and bug patterns. Journal of Software: Evolution and Process **35**(2), e2507 (2023). DOI 10.1002/smr.2507. URL https://onlinelibrary.wiley.com/doi/10.1002/smr.2507

[19] Humbatova, N., Jahangirova, G., Bavota, G., Riccio, V., Stocco, A., Tonella, P.: Taxonomy of real faults in deep learning systems. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, pp. 1110–1121. ACM (2020). DOI 10.1145/3377811.3380395. URL https://dl.acm.org/doi/10.1145/3377811.3380395

[20] Ibrahim, W.M., Bettenburg, N., Adams, B., Hassan, A.E.: On the relationship between comment update practices and software bugs. Journal of Systems and Software **85**(10), 2293–2304 (2012)

[21] JetBrains: The State of Developer Ecosystem 2022 (2022). URL https://www.jetbrains.com/lp/devecosystem-2022/data-science/

[22] Jia, L., Zhong, H., Wang, X., Huang, L., Lu, X.: An empirical study on bugs inside tensorflow. In: International Conference on Database Systems for Advanced Applications, pp. 604–620. Springer (2020)

[23] Jimenez, M., Papadakis, M., Le Traon, Y.: An empirical analysis of vulnerabilities in openssl and the linux kernel. In: 23rd Asia-Pacific Software Engineering Conference, pp. 105–112. IEEE (2016)

[24] Källén, M., Sigvardsson, U., Wrigstad, T.: Jupyter notebooks on github: characteristics and code clones. The Art, Science, and Engineering of Programming **5**(3) (2021)

[25] Kamienski, A.V., Palechor, L., Bezemer, C.P., Hindle, A.: Pysstubs: Characterizing single-statement bugs in popular open-source python projects. In: IEEE/ACM 18th International Conference on Mining Software Repositories, pp. 520–524. IEEE (2021)

[26] Lewis, C., Whitehead, J., Wardrip-Fruin, N.: What went wrong: a taxonomy of video game bugs. In: Proceedings of the fifth international conference on the foundations of digital games, pp. 108–115 (2010)

[27] lewtun: Huggingface notebooks, commit 589f8aa. https://github.com/huggingface/notebooks/commit/589f8aa27f57562bab4f3427183a4be77a888459 (2022). Accessed: July 2025

[28] Lutellier, T., Pham, H.V., Pang, L., Li, Y., Wei, M., Tan, L.: Coconut: combining context-aware neural translation models using ensemble for program repair. In: Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis, pp. 101–114 (2020)

[29] Macklon, F., Viggiato, M., Romanova, N., Buzon, C., Paas, D., Bezemer, C.P.: A taxonomy of testable html5 canvas issues. IEEE Transactions on Software Engineering (2023)

[30] Makhshari, A., Mesbah, A.: Iot bugs and development challenges. In: IEEE/ACM 43rd International Conference on Software Engineering, pp. 460–472. IEEE (2021)

[31] Merino, M.V., van Binsbergen, L.T., Seraj, M.: Making the invisible visible in computational notebooks. In: IEEE Symposium on Visual Languages and Human-Centric Computing, pp. 1–3. IEEE (2022)

[32] minrk: Jupyterhub, commit 43a6cd0. https://github.com/jupyterhub/jupyterhub/commit/43a6cd0bf90c8496ed5c47eaadd2803bbb87e0b6 (2017). Accessed: July 2025

[33] Mitre: 2024 cwe top 25 most dangerous software weaknesses. https://cwe.mitre.org/top25/archive/2024/2024_cwe_top25.html (2024). Accessed: July 2025

[34] Morag, A.: The Threats to Jupyter Notebook. Computer Fraud & Security **2022**(12) (2022)

[35] Nahar, N., Zhou, S., Lewis, G., Kästner, C.: Collaboration Challenges in Building ML-Enabled Systems: Communication, Documentation, Engineering, and Process. Proceedings of the 44th International Conference on Software Engineering pp. 413–425 (2022)

[36] Patra, J., Pradel, M.: Nalin: learning from runtime behavior to find name-value inconsistencies in jupyter notebooks. In: Proceedings of the

44th International Conference on Software Engineering, pp. 1469–1481 (2022)

[37] Pimentel, J.F., Murta, L., Braganholo, V., Freire, J.: A large-scale study about quality and reproducibility of jupyter notebooks. In: IEEE/ACM 16th international conference on mining software repositories, pp. 507–517. IEEE (2019)

[38] Pimentel, J.F., Murta, L., Braganholo, V., Freire, J.: Understanding and Improving the Quality and Reproducibility of Jupyter Notebooks. Empirical Software Engineering **26**(4), 1–55 (2021)

[39] Quaranta, L., Calefato, F., Lanubile, F.: Kgtorrent: A dataset of python jupyter notebooks from kaggle. In: IEEE/ACM 18th International Conference on Mining Software Repositories, pp. 550–554. IEEE (2021)

[40] Quaranta, L., Calefato, F., Lanubile, F.: Eliciting Best Practices for Collaboration with Computational Notebooks. Proceedings of the ACM on Human-Computer Interaction **6**(CSCW1), 1–41 (2022)

[41] Quaranta, L., Calefato, F., Lanubile, F.: Pynblint: a static analyzer for python jupyter notebooks. In: Proceedings of the 1st International Conference on AI Engineering: Software Engineering for AI, CAIN '22. ACM (2022). DOI 10.1145/3522664.3528612. URL http://dx.doi.org/10.1145/3522664.3528612

[42] Rahman, A., Farhana, E., Parnin, C., Williams, L.: Gang of eight: A defect taxonomy for infrastructure as code scripts. In: IEEE/ACM 42nd International Conference on Software Engineering, pp. 752–764. IEEE (2020)

[43] Ritta, N., Settewong, T., Kula, R.G., Ragkhitwetsagul, C., Sunetnanta, T., Matsumoto, K.: Reusing my own code: Preliminary results for competitive coding in jupyter notebooks. In: 29th Asia-Pacific Software Engineering Conference, pp. 457–461. IEEE (2022)

[44] Romano, A., Liu, X., Kwon, Y., Wang, W.: An empirical study of bugs in webassembly compilers. In: 36th IEEE/ACM International Conference on Automated Software Engineering, pp. 42–54. IEEE (2021)

[45] Rousseeuw, P.J.: Silhouettes: a Graphical Aid to the Interpretation and Validation of Cluster Analysis. Journal of computational and applied mathematics **20**, 53–65 (1987)

[46] de Santana, T.L., Neto, P.A.d.M.S., de Almeida, E.S., Ahmed, I.: Bug analysis in jupyter notebook projects: An empirical study. ACM Trans. Softw. Eng. Methodol. (2024). DOI 10.1145/3641539. URL https://doi.org/10.1145/3641539. Just Accepted

[47] Settewong, T., Ritta, N., Kula, R.G., Ragkhitwetsagul, C., Sunetnanta, T., Matsumoto, K.: Why Visualize Data When Coding? Preliminary Categories for Coding in Jupyter Notebooks. In: 29th Asia-Pacific Software Engineering Conference, pp. 462–466. IEEE (2022)

[48] sgugger: fastai2, commit 8215467. https://github.com/fastai/fastai2/commit/8215467935e431415f5c854e876db7af59bf0e37 (2020). Accessed: July 2025

[49] sgugger: fastai2, commit 8ab3e70. https://github.com/fastai/fastai2/commit/8ab3e702bf248a7c2545b64d05d60a9e60325672 (2020). Accessed: July 2025

[50] Shen, Q., Ma, H., Chen, J., Tian, Y., Cheung, S.C., Chen, X.: A comprehensive study of deep learning compiler bugs. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 968–980 (2021)

[51] Subotić, P., Milikić, L., Stojić, M.: A static analysis framework for data science notebooks. In: Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice, pp. 13–22 (2022)

[52] Sun, X., Zhou, T., Wang, R., Duan, Y., Bo, L., Chang, J.: Experience report: investigating bug fixes in machine learning frameworks/libraries. Frontiers of Computer Science **15**(6), 1–16 (2021)

[53] Tan, L.: Code comment analysis for improving software quality. In: The art and science of analyzing software data, pp. 493–517. Elsevier (2015)

[54] Tan, L., Liu, C., Li, Z., Wang, X., Zhou, Y., Zhai, C.: Bug characteristics in open source software. Empirical software engineering **19**(6), 1665–1705 (2014)

[55] Tan, L., Yuan, D., Krishna, G., Zhou, Y.: /* icomment: Bugs or bad comments?*. In: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles, pp. 145–158 (2007)

[56] Tan, S.H., Marinov, D., Tan, L., Leavens, G.T.: @ tcomment: Testing javadoc comments to detect comment-code inconsistencies. In: IEEE Fifth International Conference on Software Testing, Verification and Validation, pp. 260–269. IEEE (2012)

[57] tugstugi: dl-colab-notebooks, commit 67caaeb. https://github.com/tugstugi/dl-colab-notebooks/commit/67caaeb668aa6b596b4ad3df2acf15228b5ba0be (2020). Accessed: July 2025

[58] Vahabzadeh, A., Fard, A.M., Mesbah, A.: An empirical study of bugs in test code. In: IEEE international conference on software maintenance and evolution, pp. 101–110. IEEE (2015)

[59] Valdivia-Garcia, H., Shihab, E., Nagappan, M.: Characterizing and predicting blocking bugs in open source projects. Journal of Systems and Software **143**, 44–58 (2018)

[60] Van Binsbergen, L.T., Verano Merino, M., Jeanjean, P., Van Der Storm, T., Combemale, B., Barais, O.: A principled approach to repl interpreters. In: Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, pp. 84–100 (2020)

[61] Wan, Z., Lo, D., Xia, X., Cai, L.: Bug characteristics in blockchain systems: a large-scale empirical study. In: IEEE/ACM 14th International Conference on Mining Software Repositories, pp. 413–424. IEEE (2017)

[62] Wang, D., Li, S., Xiao, G., Liu, Y., Sui, Y.: An exploratory study of autopilot software bugs in unmanned aerial vehicles. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 20–31 (2021)

[63] Wang, J., Kuo, T.y., Li, L., Zeller, A.: Assessing and Restoring Reproducibility of Jupyter Notebooks. In: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, pp. 138–149 (2020)

[64] Wang, J., Li, L., Zeller, A.: 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, pp. 53–56 (2020)

[65] Wang, P., Brown, C., Jennings, J.A., Stolee, K.T.: Demystifying regular expression bugs. Empirical Software Engineering **27**(1), 1–35 (2022)

[66] Wang, S., Liu, T., Tan, L.: Automatically learning semantic features for defect prediction. In: Proceedings of the 38th international conference on software engineering, pp. 297–308 (2016)

[67] Wei, Y., Sun, X., Bo, L., Cao, S., Xia, X., Li, B.: A comprehensive study on security bug characteristics. Journal of Software: Evolution and Process **33**(10), e2376 (2021)

[68] x4nth055: pythoncode-tutorials, commit 351ee95. https://github.com/x4nth055/pythoncode-tutorials/commit/351ee9548a4a2981f39f557127f99fd0ac11f506 (2020). Accessed: July 2025

[69] Zhai, J., Xu, X., Shi, Y., Tao, G., Pan, M., Ma, S., Xu, L., Zhang, W., Tan, L., Zhang, X.: Cpc: Automatically classifying and propagating natural language comments via program analysis. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, pp. 1359–1371 (2020)

[70] Zhao, Y., Leung, H., Yang, Y., Zhou, Y., Xu, B.: Towards an understanding of change types in bug fixing code. Information and software technology **86**, 37–53 (2017)

[71] Zhou, X., Han, D., Lo, D.: Simple or complex? together for a more accurate just-in-time defect predictor. In: Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension, pp. 229–240 (2022)

[72] Zhou, Y., Sharma, A.: Automated identification of security issues from commit messages and bug reports. In: Proceedings of the 2017 11th joint meeting on foundations of software engineering, pp. 914–919 (2017)

[73] Zhu, C., Saha, R.K., Prasad, M.R., Khurshid, S.: Restoring the Executability of Jupyter Notebooks by Automatic Upgrade of Deprecated APIs. In: Proceedings of the IEEE/ACM 36th International Conference on Automated Software Engineering, pp. 240–252. IEEE (2021)