Characterising Bugs in Jupyter Platform

Yutian Tang*
University of Glasgow
United Kingdom

Hongchen Cao
ShanghaiTech University
China

Yuxi Chen
University of Glasgow
United Kingdom

David Lo
Singapore Management University
Singapore

ABSTRACT

As a representative literate programming platform, Jupyter is widely adopted by developers, data analysts, and researchers for replication, data sharing, documentation, interactive data visualization, and more. Understanding the bugs in the Jupyter platform is essential for ensuring its correctness, security, and robustness. Previous studies focused on code reuse, restoration, and repair execution environment for Jupyter notebooks. However, the bugs in Jupyter notebooks' hosting platform Jupyter are not investigated. In this paper, we investigate 387 bugs in the Jupyter platform. These Jupyter bugs are classified into 11 root causes and 11 bug symptoms. We identify 14 major findings for developers. More importantly, our study opens new directions in building tools for detecting and fixing bugs in the Jupyter platform.

CCS CONCEPTS

• **Software and its engineering**; • **General and reference** → **Empirical studies**;

KEYWORDS

Bugs, Jupyter, Empirical software engineering

ACM Reference Format:
Yutian Tang, Hongchen Cao, Yuxi Chen, and David Lo. 2025. Characterising Bugs in Jupyter Platform. In *Proceedings of The 29th International Conference on Evaluation and Assessment in Software Engineering (EASE 2025).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Literate programming is the style of programming by interleaving executable code snippets, text descriptions, and computation results [68]. The computation results are generated by executing the code snippets. The text description explains the source code and computation results. Such a diagram contributes to code understanding and makes the computation results explainable.

The **Jupyter** platform is the most widely used platform for interactive literate programming [4, 85]. Upon the Jupyter platform, developers and data analyzers can develop **Jupyter notebooks**. They are mainly used for *replication*, *sharing*, *documentation*, *interactive data visualization* for data analysis [5, 10, 14]. The Jupyter platform offers the execution environment for running Jupyter notebooks, which are written by notebook authors or developers. As shown by the statistics, in September 2018, there are more than 2.5 million Jupyter notebook repositories, which are 10 times more than that in 2015 [82]. The Jupyter platform originated from IPython [75]. By now, more languages are supported by the Jupyter platform, such as R, JavaScript, and C.

Motivation. As all Jupyter notebooks are executed with the Jupyter platform, ensuring the correctness of the Jupyter platform is crucial for using and developing Jupyter notebooks. Unfortunately, the nature of bugs in the Jupyter platform is currently not well understood. It is not clear what are the root causes of the bugs in the Jupyter platform, the consequences of Jupyter bugs, and their impacts. Such information can assist Jupyter platform developers and researchers in (1) understanding the root causes of the bugs in the Jupyter platform (Finding 1-4 in Sec.4.1); (2) understanding the common symptoms caused by bugs in the Jupyter platform (Finding 5-7 in Sec. 4.2); (3) quick localization and fixing of Jupyter bugs (Finding 11-14 in Sec.4.4 and 4.5); and (4) the development of Jupyter bug detection and testing tools (Sec. 5.1). Thus, it is time to investigate the bugs in the Jupyter platform.

Related Work. The existing studies explored the code replication and code reuse on Jupyter notebooks [69, 76, 83, 92], how to restore and repair Jupyter notebooks for reproduction [94, 99], code quality on Jupyter notebooks [93]. Some other non-SE domain researches care about interaction and data visualization in Jupyter notebook [66, 67, 70, 71, 74, 79, 81, 96]. In summary, none of the existing studies investigated the bugs on the Jupyter platform.

Our Study. To fill this gap, in this paper, we present the *first* systematic study of the bugs on the Jupyter platform. To understand the nature of bugs in the Jupyter platform, in this paper, we aim at answering the following research questions (RQ).

- RQ1: What are common root causes and how often do they occur?
- RQ2: What are common symptoms and how often do they occur?
- RQ3: What are the connections between root causes and symptoms of Jupyter platform bugs?
- RQ4: What are the challenges in detecting Jupyter platform bugs?
- RQ5: What are the challenges in fixing Jupyter platform bugs?

2 BACKGROUND

Knuth [68] introduced literate programming by interleaving the code snippets and natural language to support developers in understanding the underlying thoughts behind the program segments. As a practice of literate programming, the *interactive literate programming environment*, Jupyter platform, is widely adopted by developers, data analyzers, and researchers.

Jupyter notebooks, the interactive literate programming documents, are designed and developed by developers to replicate, share, and visualize data. To avoid ambiguity, in this paper, we use the

*Corresponding author. Email: yutian.tang@glasgow.ac.uk

term *Jupyter platform* to refer to the interactive literate programming environment. The terms *Jupyter notebooks* refer to the documents developed by developers and executed upon the programming environment Jupyter platform.

2.1 Jupyter Notebook

A notebook is a sequence of *cells* [58]. The type of a cell can be a *code cell* or a *markdown cell*. The *code cell* contains executable source code and can be used to produce results. The *markdown cell* contains rich formatted texts which support the markdown.

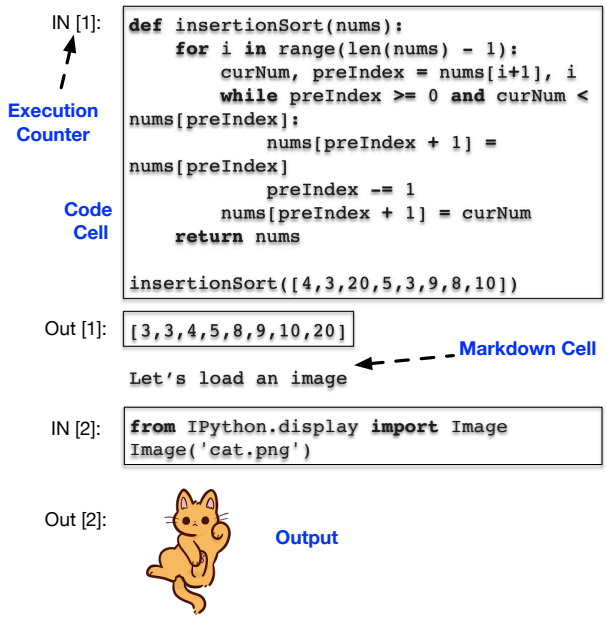

Figure 1: Jupyter Notebook Example

Fig. 1 illustrates a Jupyter notebook. It contains one markdown cell and two code cells. On the left of cells, there are *execution counters*, which indicate the execution order of these cells. When these cells are executed, the outputs are displayed afterward. However, execution order only indicates the order of these cells rather than any logical relationship between cells.

2.2 Jupyter Platform

The Jupyter platform composes of four parts: a web browser, a notebook server, a notebook document, and a kernel [58]. The overall architecture is shown in Fig. 2.

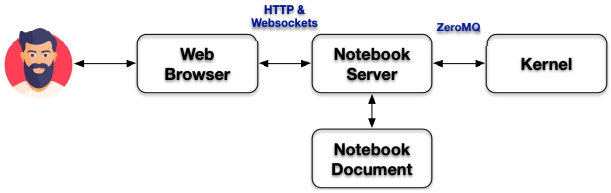

Figure 2: Jupyter Architecture

Web Browser. Users interact with the web browser. The web browser can be considered as the frontend for the user. With this, users can manipulate notebooks.

Notebook document. A notebook is a file encoded with JSON, whose extension is `.ipynb`. The notebook can be displayed with the web browser. The code snippets, texts, and other markdown notes are stored in the editable notebook.

Notebook Server. When users interact with the web browser, requests are sent to the notebook server. Requests can be either HTTP or WebSocket requests. When users require their code snippets to be executed, the notebook server sends them to the kernel over ZeroMQ sockets [97]. The kernel executes the code snippets and returns the results to the notebook server. Then, the notebook server returns the updated HTML page to users [58].

Kernel. The kernel in the Jupyter platform normally refers to `IPython kernel`, which is in charge of running code. For other programming languages, there are other kernels, such as IRKernel for R [18] and IJulia kernel for Julia [15].

2.3 Code Repositories of Jupyter Platform

Table 1: The key Code Repositories in Jupyter

Code Repo.	Functionality	Component in Fig.2
Jupytercore [61]	Core common functionality of Jupyter platform	All
Jupyterclient [60]	API for managing and communicating with kernels	Kernel, Server
IPykernel [16]	IPython kernel for Jupyter platform	Kernel
Jupyterserver [62]	Backend server for Jupyter notebook	Server
Notebook(Rep) [59]	User interface for Jupyter notebook	Web browser

The key repositories in the Jupyter platform are shown in Table 1, including `Jupytercore` [61], `Jupyterclient` [60], `IPykernel` [16], `Jupyterserver` [62] and `Notebook(Rep)` [59]. The first column shows the name of the repository. The second column describes the functionality of the repository. The last column shows the relations between the code repositories and components in Fig. 2. For example, `Jupyterserver` [62] is the repository for implementing the server. It is worth mentioning that the term "Notebook(Rep)" represents the code repository in the Jupyter platform for loading and rendering a Jupyter notebook (i.e., a document). Furthermore, we also leverage the `pydeps` [77], a Python dependency visualization library, to display the dependencies of these repositories inside the Jupyter platform. Due to the size of the generated graph, we make it accessible on our artifact [2].

3 METHODOLOGY AND CLASSIFICATION

3.1 Data Collection, Labelling and Classification

To collect and label the data used in our study, we propose a semi-automatic approach as shown in Fig. 3. Specifically, in step ❶, we collect the *closed* and *merged* pull requests that fix bugs from the aforementioned five repositories that were created on or before March 15, 2022 via GitHub APIs. The *closed* and *merged* pull requests (PR) indicate that the bugs have been fixed by developers. To assist us in better understanding the fixed PRs, we also require that the selected pull requests must be associated with Issues. To filter out non bug-fixing PRs, we follow the guidance in the existing research [8, 13, 90, 98]. Thus, in step ❷, we reserve a list of words that have similar meanings to "bugs". The list contains: fix, defect, error, bug, issue, mistake, incorrect, fault, and flaw. Next, we search each word in both tags and titles. If a PR contains at least one keyword, we consider it as a bug-fixing PR. As a result, we

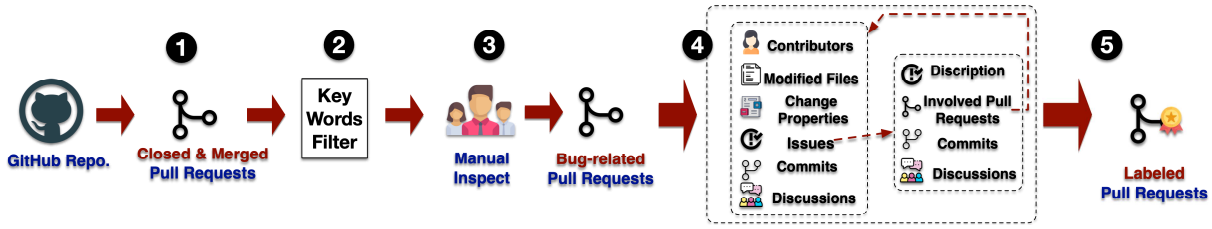

Figure 3: Overview of Data Collection and Labelling

obtain 510 bug-fix PRs in total. To eliminate false positives, in step ❸, we manually inspect these 510 PRs and related Issues to reduce false positives. We remove some PRs that do not relate to bugs in the project. We result in 387 PRs, with 10 PRs from Jupytercore; 30 from Jupyterclient; 56 from Jupyterserver; 57 from IPykernel; and 234 from Notebook(Rep). To characterize Jupyter bugs, we focus on labeling them from two perspectives: (1) the **root causes** that reflect the errors made by developers; (2) the **symptoms** that the bugs exhibit as represented by incorrect behaviors. Thus, in step ❹, for each PR, we collect the following information: contributors, modified files, change properties (e.g., how the files are modified), Issues linked to this PR, all related commits, and discussions under this PR. Furthermore, we also collect the data for any Issue linked to this PR, including the Issue's description, all PRs related to the Issue, all related commits, and discussions under the Issue. As an Issue can map to multiple PRs and a PR can be related to multiple Issues, we iteratively crawl all related Issues and PRs. The collected data can assist us in labeling and classifying these PRs.

Taxonomy. To reduce bias during classification, we ask two authors of this paper to analyze PRs separately (step ❺) to label the PRs' root causes and symptoms. If there is a conflict, another author is required to label the bug. To label and classify PRs, in this paper, we reference and adjust the taxonomy used in the existing research [13, 84, 87] to suit the Jupyter platform. Furthermore, we adopt an open-coding scheme [13] to expand the list of the root causes. That is, if a bug does not fall into any category, the author does a manual analysis to identify its label for the root causes. By doing this, we can expand the list of root causes. A similar procedure is conducted to set up the taxonomy of the symptoms.

Implementation. We implement Python scripts to automate step ❶, ❷, and ❹.

3.2 Root Causes

According to the process described in Sec. 3.1, the root causes are the following 11 categories:

• **Algorithm/Method (Alg/Meth):** The logic in the implementation of an algorithm or method is incorrect. For example, [25] shows an incorrect implementation of the code cell replacement function.

• **Assignment/Initialization (Ass/Ini):** A variable is incorrectly assigned, or mishandling of the initializations (e.g., class initialization function error, attribute assignment error in `.css` file). For example, the bug [37] is caused by lacking of settings for `padding-top` in an `.less` file which is responsible for page rendering.

• **Checking:** Lack of necessary checks that lead to an error. Checking errors can be explicit (e.g., missing `try...catch..`,`if...else` statements) or implicit. Taking the bug[38] as an example, as shown in List. 1, the developer adds `flush()` instead of `try...catch..` or `if...else` in line#5 to check the status of the executed cells, which belongs to this category.

Listing 1: An Implicit *Checking* Example from Ipykernel#390

```
1  def dispatch_shell ( self , stream. msg):
2          ...
3          self . _publish_status (...)
4  +  // flush to ensure reply is sent before handling the request
5  +  stream. flush (zmq.POLLOUT)
```

• **Logic:** Incorrect condition expressions lead to an error. For example, the bug [31] is caused by using the incorrect variable (i.e., use the key of the dictionary instead of the corresponding value) in the conditional expression.

• **Data:** Incorrect manipulation of data items, such as incompatible types in assignments, inappropriate class inheritance, wrong type conversions, and incorrect definitions of data structures. For example, the bug [35] is caused by incorrect type conversion.

• **External Interface (Exter-API):** Misuse of third-party libraries or interfaces from other systems, such as incorrect function parameters passing, and invocations of deprecated functions. For example, the bug [48] is caused by using deprecated APIs from the third-party library ZMQ [97].

• **Internal Interface (Inter-API):** Misuse of interfaces from other components of the Jupyter platform, such as incorrect function parameters passing, and invoking inappropriate functions. For example, the bug [55] is caused by passing incorrect parameters when invoking an API in Notebook(Rep).

• **Timing/Performance (Time/Perf):** Timing or performance problems, such as race condition, misuse of asynchronous/synchronous, and inappropriate use of multi-threading. For example, the bug [47] is caused by a race condition between the restart module and the shutdown module in the Jupyterclient.

• **Configuration (Config):** Misconfiguration of files for compilation, build, test, and installation (e.g., incompatible third-party library versions, and inappropriate package importation). For example, the bug [54] is caused by using an outdated third-party library anyio [1], which is incompatible with Python 3.6.

• **Non-functional (Non):** Non-functional errors that do not directly affect the use of the Jupyter platform (e.g., inappropriate description in error traceback and log). For example, the bug [44] is caused by incorrect log messages which can make users confused.

• **Others:** Other root causes that cannot be classified into any of the above categories. For example, the bug [20] is caused by `Datetime` objects do not follow time data standard ISO8601 [11].

3.3 Symptoms

The complete list of symptoms is in the following 11 categories:

• **Crash:** Critical errors that lead to a crash at runtime. For example, Issue [22] reports that the web browser crashes after the user uploads large files to the Jupyter platform.

• **Hang:** The kernel (i.e., IPykernel) or terminal (e.g., terminal in Jupyterclient) is not responding (e.g., fails to kill the kernel, infinite loop). For example, Issue [46] reports that running certain code (i.e., `%gui tk`) makes the IPykernel has no response and gets stuck.

• **Build:** The errors during installation or building. For example, the installation failure reported in [45] is due to the lack of the dependency library `ipyparallel` [17] in IPykernel.

• **Display and GUI (DGUI):** The GUI-related or display-related errors (e.g., missing widgets, misalignment of the icons and fonts). For example, a long line of code can be overlapped with the border of the code cell [26].

• **Launch:** Any errors that occur during Launch. For example, the user fails to open a notebook (i.e., `.ipynb` file) after setting the environment variable (i.e., `JUPYTER_PATH`) to a custom value [42].

• **IO:** Incorrect behaviors when performing inputs/outputs to and interacting with the Jupyter platform. For example, the user cannot delete empty lines by pressing Backspace key on the keyboard [41].

• **Security and Safety (SS):** Errors cause security vulnerabilities that can be exploited, such as incorrect permissions and information leakage. For example, the exploitable vulnerability reported in [23] can lead to the leakage of the user information.

• **Test:** Errors found during the testing phase by causing test suites to fail, usually have no direct symptom description.

• **Unreported (Un):** Error in which bug reporters do not give a clear symptom description. For example, the user only reports that the Jupyter platform does not work with tornado6 [88] but does not give any further description [39].

• **Deprecation (De):** The errors related to outdated third-party libraries/modules. For example, PR [50] shows that using a deprecated regex API [78] triggers a warning message from Jupyterserver.

• **Others:** Other symptoms that cannot be regarded as one of the above categories. For example, the notebook (i.e., `.ipynb` file) is cleared after renaming the file [40].

3.4 Research Questions (RQ)

Our study aims to answer the following five research questions:

• **RQ1: What are common root causes and how often do they occur?** Studying root causes contributes to the understanding of the nature of bugs in the Jupyter platform. The classification of root causes and the statistics of their distribution are the basis for further analysis.

• **RQ2: What are common symptoms and how often do they occur?** Symptoms are the intuitive expression of errors, which provide developers with a hint for solving the errors. Exploring symptoms helps developers understand the Jupyter platform's bugs.

• **RQ3: What are the connections between root causes and symptoms of Jupyter platform bugs?** Existing studies show that the root causes and the symptoms of bugs are intrinsically related [3, 13, 19]. Learning whether a root cause can be linked to a symptom can help developers to further understand and detect Jupyter bugs.

• **RQ4: What are the challenges in detecting Jupyter platform bugs?** Detecting bugs is an essential prerequisite for bug analysis and fixing. In this RQ, we analyze the challenges in bug detection.

• **RQ5: What are the challenges in fixing Jupyter platform bugs?** After successfully locating the source of the bug, how to fix it efficiently is the final hurdle. It allows developers to fix bugs more efficiently.

4 RESULTS

4.1 RQ1: Root Causes

Methodology. The methodology is described in Sec. 3.

Table 2: Distribution of Root Causes

Root Cause	J.core[1]	J.client[1]	IPykernel	J.server[1]	Note.[1]	$Total_{Cause}$
Alg/Meth	1	6	**20**	**15**	54	**96(24.81%)**
Ass/Ini	2	4	9	7	**67**	**89(23.00%)**
Checking	**4**	**7**	11	9	32	**63(16.28%)**
Logic	0	5	4	3	23	**35(9.04%)**
Data	0	0	2	0	1	**3(0.78%)**
Exter-API	1	2	3	2	2	**10 (2.58%)**
Inter-API	0	1	1	1	7	**10 (2.58%)**
Time/Perf	0	1	0	1	2	**4 (1.03%)**
Config	2	2	5	**15**	37	**61 (15.76%)**
Non	0	0	1	2	6	**9 (2.32%)**
Others	0	2	1	1	3	**7 (1.81%)**
$Total_{Module}$	**10**	**30**	**57**	**56**	**234**	**387**

[1] J.core for Jupytercore, J.clicent for Jupyterclient, J.server for Jupyterserver, Note. for Notebook(Rep)

Results. Table. 2 shows the distribution of Jupyter platform bugs by root cause categories. For each component of the Jupyter platform, its most frequent root cause is as follows:

• For Jupytercore and Jupyterclient, **Checking** is the top root cause, accounting for 40.00% of all bugs in Jupytercore and 23.33% of all bugs in Jupyterclient;

• For IPykernel, **Alg/Meth** is the top root cause (35.08% of all bugs in IPykernel).

• For Jupyterserver, **Alg/Meth** and **Config** are tied for first place (both 26.79% of all bugs in Jupyterserver).

• For Notebook(Rep), **Ass/Ini** is the top root cause (28.63% of all bugs in Notebook(Rep)).

• Overall, Alg/Meth, Ass/Ini, Checking, and Config are the top four frequent root causes for the Jupyter, accounting for 24.81%, 23.00%, 16.28%, and 15.76% of all studied bugs, respectively.

Alg/Meth is the most frequently occurring category. We find that for different components, the algorithms with bugs have different characteristics. Specifically, according to Fig. 2 and Table. 1, every component has its main functionality. Algorithms with bugs often have a strong relationship with the functionality of their components. Recall that Alg/Meth is the top root cause of bugs in Jupyterserver and IPykernel. For Jupyterserver, 8 out of 15 PRs modify the file `handlers.py`, suggesting that algorithms related to server handlers are prone to bugs. For IPykernel, half of the PRs modify `kernelbase.py` or `iostream.py`, suggesting that algorithms related to kernel read and write are not robust.

Finding 1

Alg/Meth is the most frequent root cause, accounting for 24.81% of Jupyter platform bugs.

Ass/Ini is the second most frequent root cause. As defined in Sec. 3.2, this category involves the assignment or initialization errors or mishandling of the initializations. Recall that Ass/Ini is the top root cause of bugs in Notebook (Rep). Notebook(Rep) involves front-end browser pages that interact with users and consist of files in the format of `.html`, `.css`, `.js`, `.less`. It can be error-prone to manually set up all attributes for front-end UI elements in Notebook(Rep). Even simple mistakes (i.e., Ass/Ini bugs) can lead to various errors, such as page rendering errors, front-end interaction failures, and other problems. In addition, some page rendering errors are difficult to find through tests and are often noticed by users or other developers. Taking Notebook(Rep)'s PR#4236 [37] as an example, an icon is overlaid on the docstring of the tooltip. Such an incorrect layout prevents the user from reading the full description of the function signature. The difference between the normal and incorrect UI is so subtle that it can be easily overlooked.

Finding 2

Ass/Ini is the second most frequent root cause, accounting for 23.00% of Jupyter platform bugs. This category of bugs is rampant in Notebook(Rep) and sometimes hard to detect.

Checking is the third most frequent root cause. This category of bugs is relatively trivial to fix, usually by adding `if...else` or `try...catch`. Recall that Checking is the top root cause of bugs in Jupytercore or Jupyterclient.

Jupytercore offers the core common functionality of the Jupyter platform. Thus, it involves a significant amount of checks, such as checks for inputs, configurations, and options. Lacking such checks can make the Juypter platform fail to work properly. For example, PR [43] reports a bug caused by lacking file permission checks on Windows. Jupyterclient offers APIs for starting, managing, and communicating with Jupyter kernels. When invoking these APIs, developers have to pass arguments to these APIs. Inside these APIs, sufficient checks must be conducted on these passed arguments. Insufficient checks can lead to exceptions. For example, PR [33] reports a bug due to the lack of checking whether the `meth.__doc__` is `None` or not.

Finding 3

Checking is the third most frequent root cause, accounting for 16.28% of Jupyter platform bugs. Under this category, exception-related bugs are potentially detectable and fixable by existing tools.

Config is also a serious root cause that occupies a significant percentage. As defined in Sec. 3.2, this category involves misconfiguration of files, such as files for compilation and build. Recall that Config is the top root cause of bugs in Jupyterserver. According to the dependency graph of the Jupyter platform (see our artifact [2]), Jupyterserver has 14 upstream dependencies. The most common Config errors in Jupyterserver are the incorrect settings of third-party libraries' versions, such as traitlets [89], anyio [1]. As soon as the third-party dependency library conflicts occur, the developer needs to change the configuration of the related libraries. Modifying the configuration is not as thorny as fixing code-related issues. However, it is hard to quickly locate the library causing the conflict among the vast number of dependency libraries.

Finding 4

Config is another serious root cause, accounting for 15.76% of Jupyter platform bugs. The incorrect settings of third-party libraries' versions are the source of Config bugs.

4.2 RQ2: Symptoms

Methodology. The detailed methodology is described in Sec. 3.

Table 3: Distribution of Symptoms

Symptom	J.core	J.client	IPykernel	J.server	Note.	$Total_{Symptom}$
Build	2	1	4	2	13	**22 (5.68%)**
Crash	1	4	2	2	4	**13 (3.36%)**
DGUI	1	0	4	2	66	**73 (18.86%)**
Hang	0	3	6	6	2	**17 (4.39%)**
IO	2	6	**21**	8	**78**	**115 (29.72%)**
Launch	**3**	**8**	3	**14**	39	**67 (17.31%)**
SS	0	0	0	0	4	**4 (1.03%)**
Test	1	2	6	7	11	**27 (6.98%)**
Un	0	2	2	8	8	**20 (5.17%)**
De	0	2	3	2	2	**9 (2.33%)**
Others	0	2	6	5	7	**20 (5.17%)**
$Total_{Module}$	**10**	**30**	**57**	**56**	**234**	**387**

[1] J.core for Jupytercore, J.clicent for Jupyterclient, J.server for Jupyterserver, Note. for Notebook(Rep)

Results. Table. 3 shows the distribution of different symptoms for Jupyter platform bugs.

- For Jupytercore, Jupyterclient, and Jupyterserver, **Launch** is the top frequent symptom, accounting for 30.00% of all bugs in Jupytercore, 26.67% of all bugs in Jupyterclient, and 25.00% of all bugs in Jupyterserver.
- For IPykernel, Notebook(Rep), **IO** is the top frequent symptom, accounting for 36.84% of all bugs in IPykernel and 33.33% of all bugs in Notebook(Rep).
- Furthermore, IO, DGUI, and Launch are the top three frequent symptoms, accounting for 29.72%, 18.86%, and 17.31% of all studied bugs, respectively.

Among all symptoms, IO is the most frequently reported symptom. Specifically, the IO symptom appears in all components. Recall that IO is the top symptom of bugs in Notebook(Rep) and IPykernel.

For Notebook(Rep), it implements the frontend for the end-users. As it directly interacts with users, the IO errors are mostly reported in Notebook(Rep). For example, PR [32] reports a bug that the Edit/View buttons are either not working or working incorrectly.

Sometimes, IOs can be complex. For example, users need to query some information from a server and run a code snippet. Such complex IO requests can only be processed with backend components (e.g.,Jupterclient, IPykernel). An error in a backend component can lead to an incorrect result, which is finally presented to end-users. That is the reason why IO errors can also be reported in backend components. For IPykernel, as shown in Fig. 2, the users' requests are sent by Notebook Server via ZeroMQ. The `iostream.py` in the IPykernel is in charge of parsing the ZeroMQ messages. Error in

Table 4: Connections between Root Causes and Symptoms

	Crash	Hang	Build	Launch	IO	DGUI	SS	Test	Un	De	Others	$Total_{Causes}$
Alg/Meth	4	10	1	14	**40 (41.67%)**	13	1	4	4	0	5	**96**
Ass/Ini	1	1	3	13	24	**41 (46.07%)**	0	2	2	0	2	**89**
Checking	5	4	4	15	**19 (30.16%)**	3	3	5	1	0	4	**63**
Data	0	0	0	0	0	2	0	0	1	0	0	**3**
Exter-API	1	0	0	0	0	0	0	0	0	9	0	**10**
Inter-API	1	0	0	2	5	1	0	0	1	0	0	**10**
Logic	1	0	1	9	15	3	0	2	2	0	2	**35**
Non	0	0	1	1	0	2	0	0	1	0	4	**9**
Time/Perf	0	0	0	1	2	0	0	1	0	0	0	**4**
Config	0	2	**12 (19.67%)**	11	9	8	0	10	7	0	2	**61**
Others	0	0	0	1	1	0	0	3	1	0	1	**7**
$Total_{Symptoms}$	**13**	**17**	**22**	**67**	**115**	**73**	**4**	**27**	**20**	**9**	**20**	**387**

the IPykernel (i.e., `iostream.py`) can return the incorrect results to users, which results in the IO symptom.

Finding 5

IO is the most frequently reported symptom, accounting for 115 out of 387. The IO symptom appears in all components of the Jupyter. IO is the top frequent symptom in Notebook(Rep) and IPykernel.

DGUI is the second most frequent symptom. Most DGUI errors appear in Notebook(Rep) as it is mainly in charge of interaction with end-users. Many front-end errors and rendering errors (i.e., DGUIs) are rooted in Notebook(Rep). Some DGUI problems come from the back-end components (e.g., Jupytercore, IPykernel, Jupyterserver). The reason for this is that these components are in charge of processing the requests (e.g., compiling a code segment) from the front end. Then, the final results are returned to the front end for rendering. The DGUI symptom can appear when there are some wrong implementations in these components. For example, when a user wants to query files, this request is processed by `FileManager` in the Juptercore and returned to the user via the web browser. Incorrect implementation of `FileManager` in the Juptercore can lead to an unexpected display for the user (i.e., a DGUI symptom).

Finding 6

DGUI is the second most frequent symptom, accounting for 18.86% of all bugs. The DGUI symptom appears in Jupytercore, IPykernel, Jupyterserver, and Notebook(Rep), with the most occurrence in Notebook(Rep).

The third most prominent symptom is Launch. The Launch symptom appears in all components of Jupyter. The reason for this is that the execution of the Jupyter platform involves launching all the components of Jupyter. Besides, it is worth noting that Launch is the top frequent symptom in Jupytercore, Jupyterclient, and Jupyterserver.

• The Launch errors are reported in Jupytercore mainly due to the lacks of checks in command-line options. In general, users can leverage the `jupyter notebook` command to launch the Jupyter platform. This command (i.e., `jupyter notebook`) offers some options (e.g.,–port, –no-browser) to support specific needs [56]. Such options are processed in Jupytercore. The lacks of checks in invalid command options can lead to a Launch error when the user executes the Jupyter with such options.

• In Jupyterclient, Launch is the top frequent symptom. According to Table. 1, Jupyterclient is in charge of managing and communicating with IPykernel. Thus, the errors in the Jupyterclient can affect the usage and launch of IPykernel (e.g., IPykernel cannot start at launch). Launching the Jupyter platform requires the IPykernel to be launched as well. The launching problem in IPykernel causes the Launch symptom. For example, PR [49] reports a bug that the errors in the Jupyterclient make that the Jupyterclient sends multiple requests to restart the IPykernel. This is because the Jupyterclient fails to check the status (e.g., starting, closed) of the IPykernel. As a result, the Jupyter platform fails to launch.

• In Jupyterserver, the most frequent symptom is also Launch. In general, we find two possible reasons:

(1) The Jupyterserver offers the backend server for the Jupyter platform. In the Jupyterserver, the ServerApp (i.e.,`serverapp.py` in Jupyterserver) is the core part, which connects all components in the Jupyterserver [63]. When starting the Jupyter platform, the Jupyterserver is launched as well [64]. The errors (e.g., incorrect assignment, failure to process command-line options) in the ServerApp (the `serverapp.py`) can halt the launch process of the Jupyterserver. As a result, users encounter a Launch error. The examples can be found in PR [52] and PR [51].

(2) When users open a Jupyter notebook with the command `jupyter notebook <target>.ipynb`, the frontend of the Jupyter platform sends a POST request to the Jupyterserver [57, 65]. The detailed introduction of the POST request can be found in [57]. When receiving this request, the Mapping Kernel Manager (MKM) inside the Jupyterserver calls the Kernel Manager (KM) inside the Jupyterclient to launch the IPykernel [65] to parse the input notebook file. The errors in the MKM can make the IPykernel fails to connect and communicate with the Jupyterserver (e.g., IPykernel cannot be launched, IPykernel stops responding). As a result, an error (e.g., time out error) is reported during the launching process of the Jupyter platform. An example can be found in [53].

Finding 7

Launch is the third most prominent symptom, accounting for 67 out of 387 symptoms. The Launch symptom appears in every component. In Jupytercore, Jupyterclient, and Jupyterserver, Launch is the top frequent symptom that occurs most frequently.

4.3 RQ3: Connections between Root Causes and Symptoms

Methodology. Based on the statistics in RQ1 and RQ2, we further calculate relationship between root causes and symptoms. Furthermore, we combine the four most significant root causes with their highest number of symptoms into four groups. We only regard these four groups as noteworthy connections.

Results. Table 4 shows the frequency relationship between root causes and symptoms. The top four noteworthy connections are:
(1) The connection between Alg/Meth and IO.
(2) The connection between Ass/Ini and DGUI.
(3) The connection between Checking and IO.
(4) The connection between Config and Build.

For (1) and (3), IO bugs are observed in the frontend (i.e.,web browser), as the user interacts with Jupyter through this component.

For (1), Alg/Meth can be the root cause of the IO symptom. When users interact with the Jupyter platform, algorithms, and methods in the Jupyter platform are in charge of processing users' requests and returning results to users. Thus, errors in Alg/Meth can make users fail to get the correct results or even crash when interacting with the Jupyter platform. For example, PR [21] reports a bug in copying and pasting multiple cells. Such a bug is due to the incorrect implementation of the corresponding algorithm. Thus, Alg/Meth can be the root cause of the IO symptom.

For (3), Checking can also be the root cause of IO symptoms. When interacting with users, the Jupyter platform needs to perform a series of checks to ensure correctness, such as whether users' inputs are valid and acceptable. Lacking such checks can lead to unexpected outputs or errors (i.e., the IO symptom). Taking Notebook PR#1011 [27] as an example, the user discovers a bug when using Jupyter in the Firefox browser. The user reports that two same consecutive pop-ups ask users to confirm whether they want to exit after clicking the "Close and Halt" button. The reason is that the browser fires the `beforeunload` event in Javascript after clicking the button. Once the event is detected, a pop-up window appears. Firefox triggers this event twice when a browser window with unsaved content is about to close. While other browsers only fire the event once. Jupyter platform fails to check the user's browser type, which causes the bug. Thus, Checking can be the root cause of the IO symptom.

Finding 8

For IO sympotoms, Alg/Meth and Checking can be the main root causes.

Listing 2: An *Ass/Ini* Example from Notebook#4236

```
1  .tooltiptext{
2     padding-right:30px
3  + /*avoid the ui-icon(s) from overlapping the tooltip*/
4  +  padding-top:30px;}
```

For(2), functional modules responsible for the display and GUI rely on front-end programming languages, which involve a huge volume of settings for the properties and attributes in UI elements. Such settings are implemented with a series of assignment statements and initializations (Ass/Ini) in files (e.g., `.html`, `.css`, `.js`). Thus, the errors in Ass/Ini can result in page rendering errors. Taking Notebook(Rep)'s PR#4236 [37] in Sec. 4.1 as an example, the commit for fixing this bug is shown in List. 2. The source of the bug is that the attribute `padding-top` is not set. As a result, such errors can lead to the DGUI symptom. In summary, for the DGUI, Ass/Ini can be the root cause.

Finding 9

The bugs whose symptom is DGUI can be closely related to Ass/Ini. The massive and easily overlooked attribute assignments in front-end programming contribute to the strong connection.

For (4), the Config refers to the misconfiguration of files. The functions responsible for the installation and building rely on many configuration files. Wrong settings in such configuration files can cause Jupyter to behave incorrectly during the installation or building process, resulting in the Build symptom. For example, in Notebook(Rep) Issue#1977 [30], a user fails to deploy the Dev. version of the Jupyter. By manually inspecting the Issue in the Notebook(Rep), we find that the incorrect configuration files lead to this error. Also, in Notebook(Rep) Issue#52 [24], incorrect configuration of Dockerfile leads to the failure of building Notebook(Rep) images. Thus, there is a relation between the Config and Build.

Finding 10

Most bugs with symptoms reported as Build are related to improper configuration. Incorrect settings of the configuration files lead to the errors during the Build. It suggests a relation between the Config and Build.

Table 5: Ranking of Symptoms for Bug Detection

Symptom	Issues				
	#Comment	#Participant	Contributor(%)	External(%)	Ranking
Crash	6.92	3.85	38%	31%	3
Hang	4.18	2.47	59%	29%	7
Build	5.91	2.95	50%	0%	8
Launch	**10.24**	**5.01**	**39%**	**15%**	**2**
IO	5.93	3.34	55%	10%	6
DGUI	6.37	3.33	41%	14%	4
SS	**8.25**	**4.75**	**25%**	**25%**	**1**
Test	4.96	2.26	85%	7%	9
Un	2.30	1.95	80%	0%	11
De	6.67	4.67	33%	0%	15
Others	2.55	2.10	70%	5%	10
Average	**6.33**	**3.42**	**52%**	**11%**	-

4.4 RQ4: Challenges in Bug Detection

Methodology. To detect a bug, developers first need to reproduce the symptoms reported in the Issues. Then, developers locate possible faulty code snippets based on the reported symptoms. Next, developers analyze the root cause of the bug and finally proceed to bug fixing. To find the challenge of detecting Jupyter bugs, we leverage four indicators adopted by previous studies [13, 91] on Issues to quantify the difficulty in detecting bugs.

• **Comments Num (#Comment)**: The average number of comments under Issues. A high #Comment indicates that a bug requires many discussions among users and developers.
• **Participants Num (#Participant)**: The average number of participants under Issues. A high #Participant indicates that a bug requires the combined efforts of multiple users and developers to be discovered and solved.
• **Contributor Raised Issue Rate (%Contributor)**: The rate of Issues raised by developers. A high rate means a bug is more likely to be detected by the developers, suggesting it is easier to detect the bug; and
• **External Issue Rate (%External)**: The rate of Issues located in external repositories (e.g., PR in Jupytercore but the related Issues in Jupyterserver). A high rate suggests that the bug crosses different components. It indicates the complexity of the bug.

In summary, #Comment and #Participant are positively related to the difficulty of detecting bugs. Furthermore, %External is positively related to the complexity of bugs. %Contributor is negatively related to the complexity of bugs. We adopt the non-dominated ranking [7, 72] to sort the difficulty in bug detection. Compared to sorting algorithms that aggregate objects using linear or nonlinear functions, non-dominated ranking orders multiple objects without aggregation (i.e., no weights need to be tuned). The only parameter to set in the non-dominated ranking is the optimal value of each indicator. To eliminate the effect of different magnitudes of indicators, we first normalize each indicator as follows:

$$\frac{indicator_j^i - indicator_{min}^i}{indicator_{max}^i - indicator_{min}^i},$$

where $1 \leq i \leq 4$ and $1 \leq j \leq 11$. For example, #Comment of Launch in Table. 5 is $indicator_4^1$. Thus, for positively related indicators, the optimal value is 1, and vice versa is 0. Non-dominated ranking sorts by calculating the euclidean distance from the normalized indicators to their optimal values.

Results. From the perspective of symptoms, SS-related bugs rank first in Table. 5, with 8.25 for #Comment, 4.75 for #Participant, 25% for Contributor, and 25% for External of all SS Issues.

►**Challenge 1: Reproducing and locating SS-related bugs based on abstract symptom description.** By manually analyzing the Issues on SS, we find that half of the Issues on SS do not have the specific error message and reproduce method. Taking Notebook(Rep) Issue#2503 [34] as an example, a user requests that Jupyter should support binding to Unix sockets to avoid exposing IP and port information. Such security concern makes it difficult for developers to reproduce the bugs, especially when there is no error message reported. Thus, developers first attempt to reproduce the SS-related bugs based on their background knowledge.

After reproducing the SS-related bugs, developers need to locate the relative buggy code. However, it is hard to define the "buggy code" for an SS-related bug. Different from a traditional bug detection and fixing process, an SS-related bug may not have unique solutions. Normally, developers have to inspect all feasible solutions before determining the "buggy code". For example, in Notebook(Rep) Issue#1074 [28], to avoid running the web browser in the root mode, developers propose different possible solutions:

• Solution (1): When a user opens the web browser in root mode, the web browser can send a warning message to the user;
• Solution (2): The kernel can deny users from running in root mode; and
• Solution (3): When the user launches the web browser, the front-end checks whether the Jupyter platform is running in root mode. If so, it can raise an assertion error and shut down the web browser.

Among these three possible solutions, (1) is considered to be ineffective and is abandoned by developers. (2) can restrict users from using IPykernel in many scenarios. For example, some users need to run the console in root mode in IPykernel. (3) successfully prevents users from running the web browser in root mode without any side effects. As a result, solution (3) is selected. Then, the developer locates buggy code snippets and fixes this Issue.

Finding 11

According to the rank in Table. 5, SS-related bugs are challenging to detect. Reproducing and locating SS-related bugs based on abstract symptom descriptions is a challenge for SS-related bugs because there can be multiple solutions for an SS-related bug. Normally, developers have to inspect all feasible solutions before determining the "buggy code".

As shown in Table. 5, bugs related to Launch rank second. It is worth noting that Launch has the highest value of #Comment and #Participant, which suggests that detecting a Launch-related bug requires much effort among users and developers. By manually inspecting the conversations in the Issues and the commits in the pull requests, we conclude the second challenge in bug detection: Multiple root causes can lead to the same symptom.

► **Challenge 2: Multiple root causes can lead to the same symptom.** For some Launch-related bugs, we find that multiple root causes can lead to the same symptom. For example, in Notebook(Rep) Issue#448, users encounter the error "the Jupyter is not a command" when launching the web browser. Two root causes can lead to this symptom. On the one hand, this symptom can be triggered by the Alg/Meth-related bug in the third-party library IJulia, which makes the conflicts between IJulia and Notebook(Rep). On the other hand, it can also be triggered by the Checking-related bug in Jupytercore, which leads to the Mac OSX does not add Jupyter to the system path when installing. These two root causes independently cause the same symptom. Developers can only solve this Issue completely when they find both root causes. Thus, it is challenging for developers to systematically find all root causes and detect bugs based on a single symptom.

Finding 12

Launch ranks second in Table. 5. The same symptom can be related to multiple root causes is the challenge for detecting Jupyter's bugs. Developers can only solve the Issue completely by finding all the root causes leading to the same symptom.

4.5 RQ5: Challenges in Bug Fixing

Methodology. Bugs are mostly raised in Issues and fixed by developers in PRs. So to figure out the challenges in the bug fixing, we utilize the four indicators under PRs [13, 91] as follows:

Table 6: Ranking of Root Causes for Bug Fixing

Root Cause	Pull Requests				
	#Conversation	#Commit	LOCA	LOCD	Ranking
Alg/Meth	**8.46**	**3.32**	**76.26**	**27.08**	**1**
Ass/Ini	4.90	1.71	11.15	7.39	8
Checking	4.62	1.81	21.63	4.46	7
Logic	5.89	1.86	25.57	7.71	6
Data	2.67	1.67	39.33	20.00	9
Exter-API	3.30	1.50	9.40	20.50	11
Inter-API	3.90	2.40	32.30	14.40	5
Time/Perf	3.75	3.75	32.75	9.5	4
Config	**4.8**	**2.88**	**21.65**	**41.05**	**2**
Non	3.56	1.44	5.67	41.56	10
Others	6.17	1.83	50.5	11.3	3
Average	**5.71**	**2.35**	**33.48**	**18.60**	-

• **Conversations Num (#Conversation)**: The average number of conversations under PRs. A complex fix requires developers to work together to discuss plans and details. The higher the #Conversation, the more difficult the bug is to fix;
• **Commits Num (#Commit)**: The average number of commits under PRs. The higher the #Commit, the more attempts it requires to fix the bug;
• **Lines of Code Added (LOCA)**: The average number of added lines of code under PRs. The higher the LOCA, the more effort is needed to modify the code; and
• **Lines of Code Deleted (LOCD)**: The average number of deleted lines of code under PRs. Similar to LOCA, the higher the LOCD, the more effort is needed to modify the code.

Therefore, all four indicators are positively related to the difficulty of bug fixing. Non-dominated ranking described in Sec. 4.4 is also employed here for ranking. Note that for LOCA and LOCD, we use their sum rather than independently in sorting.

Results. Table. 6 suggests that bugs caused by Alg/Meth require the most effort to fix, which is consistent with finding 1 in Sec. 4.1.

► **Challenge 1: Fixing one bug in a muti-module platform can introduce extra bugs.** Alg/Meth is the most difficult category of bugs to fix. This is especially the case for a muti-module platform, such as the Jupyter. The Jupyter platform is a complex software composed of many modules and functions. Fixing a bug in one function can interference the functionality of other relative modules in the Jupyter platform (e.g., incorrect output, hang, throw an error). As a result, such a fix can introduce new bugs to the Jupyter platform. Even worse, sometimes these new bugs can not be captured by running regression tests. It is because the executing traces of newly introduced bugs can not be covered by the regression test suites. To capture these new bugs, extra test cases are required. However, developers may not be aware of these new bugs, which makes the bug fixing process hard and challenging. It suggests that the Jupyter platform needs a more comprehensive and automated testing tool to help developers maintain their software efficiently.

Finding 13

Alg/Meth is the most difficult category of bugs to fix in a muti-module platform like Jupyter. This is because fixing one bug can interference the functionality of other relative modules. As a result, extra bugs can be introduced. Even worse, some of them (i.e., extra bugs) cannot be captured with the regression test suite.

The ranking in Table. 5 suggests that Config can be another root cause worth studying in-depth. Recall the finding 4 in Sec. 4.1, bugs caused by Config often have a close connection with third-party libraries.

► **Challenge 2: Resolving third-party library conflicts in Config.** A third-party library can conflict with its repository (e.g., Notebook(Rep)) or another third-party library in the Jupyter platform.

The former conflict represents that the repository uses an incorrect version of the third-party library, which results in the modules in the repository working abnormally. This can lead to a variety of symptoms, such as front-end page rendering errors and user-entered code not displayed properly. The bug in Notebook(Rep) Issue#3629 [36] is an example. The user cannot open any notebook due to lack of the module `nbconvert.exporters.base`. Such module is introduced in `nbconvert` 5.0.0 [73] library, but Jupyter platform fails to set the minimum version of `nbconvert`, which make users with outdated `nbconvert` encounter this bug.

The latter conflict arises because both libraries contain files, modules, or executables with the same name but different contents. Using the incorrect files or modules can lead to errors. Taking Notebook(Rep) PR#1547 [29] as an example, the root of the bug is the existence of two `codemirror.js`. One is imported by the developer and another one is bundled by Webpack [95]. The Jupyter platform uses the bundled version, which results in the bug.

Currently, developers resolve dependency conflicts only by following a guess-and-check strategy. Specifically, developers first presume the potential problematic third-party library based on which modules work incorrectly. Next, they manually check the relevant code to verify whether this library is the culprit of the bug. This process is time-consuming and often requires multiple developers to work together to figure out the source of the conflict. A tool that can automatically detect which dependency libraries conflict can help developers fix these bugs more efficiently.

Finding 14

Config bugs are the second most difficult to fix, with the main challenge being identifying and resolving third-party dependency library conflicts. Incorporating an automated dependency conflict monitoring tool into Jupyter can help developers overcome this challenge.

5 DISCUSSION

5.1 Implications for Further Research

In September 2018, more than 2.5 million Jupyter notebook repositories are hosted on GitHub, which are 10 times more than that in 2015 [82]. Ensuring the robustness of the Jupyter platform not only benefits the Jupyter developers but also benefits the users' experience of the Jupyter platform. Thus, we propose the following suggestions for stakeholders.

For Jupyter developers, the suggestions are:
• Finding 2 and 9 suggest that developers should be careful when editing the front-end UI attributes. Mistakes in these attributes can lead to DGUI errors. Such mistakes can be easily overlooked.
• Finding 4 and 14 suggest that developers should continuously pay attention to configurations, especially for configuring third-party dependencies. Furthermore, downstream components (e.g., Notebook(Rep)) can be more likely to have Config bugs and dependency conflicts, which can be hard to detect.
• Finding 12 reveals that multiple root causes can lead to the same symptom. It suggests that developers should enhance the debugging and logging systems inside the Jupyter platform. Such enhancement can assist developers in exploring and fixing all bugs that can lead to a symptom.
For tool builders and researchers, the suggestions are:
• Automation Testing Tools for Repairs: Finding 13 implies that some bug fixes can introduce extra bugs to the Jupyter platform as the executing traces of some extra bugs can be excluded from the current regression test suites in the Jupyter. Thus, tool builders are supposed to build test case generators based on repairs.
• Detecting and Fixing Tools for Dependency Conflicts: Jupyter has a great number of third-party dependency libraries (e.g., CodeMirror [6], Tornado [88]). According to finding 14, dependency conflicts bugs are plentiful in Config bugs. Developers can currently only determine the source of conflicts with manual checking, which is time-consuming and unreliable. Therefore, tool builders can help maintain Jupyter by developing tools for detecting and fixing dependency conflicts.
• Tools for detecting front-end page rendering errors: Finding 2 and 9 suggest that errors in setting attributes for front-end UI elements can lead to various page rendering errors. Such errors can be easily overlooked. Thus, developers can build tools to detect such page rendering errors.

5.2 Threats to Validity

Threats to internal validity. The most dominant internal validity stems from deviations and errors in bug classification. To minimize bias, we follow the data collection procedure presented in an ICSE'20 paper [13]. To ensure that we can focus on the real bugs and their fixes, we identify closed and merged PRs with a list of words that have the meaning of bugs, as stated in detail in sec. 3.1. Finally, we manually checked all PRs and their related issues to filter out the irrelevant parts. To ensure the reliability of the bug classification results, two authors analyze the bugs separately and discuss the discrepancy with another author until reaching unification. We adopt the taxonomy used in the existing research [13, 84, 87] to label the bugs. Through these efforts, we mitigate the threats to internal validity to the greatest extent possible.
Threats to external validity. We select the closed and merged pull requests with their relevant issues until March 15, 2022. The opened pull requests and their related issues are not counted in statistics, making us miss some bugs that are continuously tracked. Also, the result of the symptoms can deviate from facts because some issues may have been updated recently. Therefore, readers should be cautious when applying our findings.

6 RELATED WORK

Despite Jupyter's popularity, research works on Jupyter and Jupyter notebooks are still limited. The existing research can be grouped into the following categories:
Code Reproducibility and Reuse on Jupyter notebooks. Pimentel et al. conducted a large-scale study on the quality and reproducibility of Jupyter notebooks [76]. They studied 1.4 million notebooks from GitHub, showing that only 24.11% of Jupyter notebooks can be executed without exception, and only around 4% of notebooks can be reproduced with the same results. Koenzen et al. [69] explored the way of code duplications in Jupyter notebooks and identified the potential barriers to code reuse. Besides, Wang et al. [92] first studied whether existing notebooks can be executed successfully (i.e., reproducibility). Then, they proposed a prototype named Osiris, which takes a notebook as an input and outputs the possible execution schemes to reproduce the notebook.
Restore/Repair and Code Quality on Jupyter notebook. Zhu et al. [99] proposed RELANCER, which is an automatic technique that can restore the executability of broken Jupyter notebooks by upgrading deprecated APIs. Wang et al. [94] developed SnifferDog to restore the execution environments for executing Jupyter notebooks. Specifically, SnifferDog first collects the APIs of Python packages to build the database and then analyzes the notebooks to determine the candidate packages and versions. Wang et al. 's work [93] conducted a preliminary study on code quality for Jupyter notebooks. They found that the existing notes are with poor quality codes, which requires quality control on Jupyter notebooks.
Empirical Study on Software Bugs. Franco et al. [8] classified 269 numerical bugs from five famous numerical libraries and further analyzed their occurrence frequency, symptoms, and fixes. Romano et al. [80] analyzed 1054 bugs in three widely-used open-source WebAssembly compilers, including AssemblyScript, Emscripten, and Rustc/Wasm-Bindgen. Garcia et al. [13] categorized the root cause and symptom of 499 bugs in two representative autonomous vehicle software (i.e., Apollo and Autoware). Shen et al. [86] classified the root cause, symptom, and occurrence stage of 603 bugs in three popular deep learning (DL) compilers (i.e., TVM, Glow, and nGraph). Gao et al. [12] analyzed and classified 103 crash recovery bugs in four open-source distributed systems. Eghbali et al. [9] conducted a comprehensive analysis of 204 string-related bugs in a dataset containing 13 top-starred JavaScript projects on GitHub.

In summary, the existing research works are more focused on restoring Jupyter notebooks' environment and making code reproducible. However, the bugs on the Jupyter are not researched. Our study fulfills this gap and assists Jupyter developers in bug detection and fixing on the Jupyter.

7 CONCLUSION AND FUTURE WORK

Given the fact that Jupyter is the most widely used platform for developing Jupyter notebooks, it is critical that software-engineering researchers build a robust and secure Jupyter platform. In this paper, we conduct a systematic study on bugs for the Jupyter platform. We identify 11 root causes and 11 symptoms across five components in the Jupyter platform from 387 Jupyter bugs. Both researchers and developers can benefit from this study. For developers, we summarize 14 findings to help them deal with bugs. For

researchers, we distill the challenges which require additional research effort (see Sec.5.1). In the future, we plan to propose techniques for automatic test suite generation for the Jupyter platform, especially for testing program repairs. Additionally, we plan to refine our study's results to build a benchmark dataset for automatic program repair techiques for the Jupyter platform.

8 DATA AVAILABILITY

The results and data can be found at: [2].

REFERENCES

[1] Anyio. 2022. AnyIO: an asynchronous networking and concurrency library. https://github.com/agronholm/anyio.
[2] Online Artifact. 2022. Online Artifact. https://sites.google.com/view/jupyter-bugs/.
[3] Thomas Ball, Mayur Naik, and Sriram K. Rajamani. 2003. From symptom to cause: localizing errors in counterexample traces. In *Proceedings of POPL*. 97–105.
[4] Lorena A. Barba. 2021. The Python/Jupyter Ecosystem: Today's Problem-Solving Environment for Computational Science. *Computing in Science & Engineering* 23, 3 (2021), 5–9.
[5] Marijan Beg, Juliette Taka, Thomas Kluyver, Alexander Konovalov, Min Ragan-Kelly, Nicolas M. Thiéry, and Hans Fangohr. 2021. Using Jupyter for Reproducible Scientific Workflows. *Computing in Science & Engineering* 23, 2 (2021), 36–46.
[6] codemirror. 2022. CodeMirror. https://github.com/codemirror/CodeMirror.
[7] Kalyanmoy Deb, Samir Agrawal, Amrit Pratap, and T. Meyarivan. 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Trans. Evol. Comput.* 6, 2 (2002), 182–197.
[8] Anthony Di Franco, Hui Guo, and Cindy Rubio-González. 2017. A comprehensive study of real-world numerical bug characteristics. In *Proceedings of ASE*. 509–519.
[9] Aryaz Eghbali and Michael Pradel. 2020. No Strings Attached: An Empirical Study of String-related Software Bugs. In *Proceedings of ASE*. 956–967.
[10] Hans Fangohr, Thomas Kluyver, and Massimo DiPierro. 2021. Jupyter in Computational Science. *Computing in Science & Engineering* 23, 2 (2021), 5–6.
[11] International Organization for Standardization(ISO). 2022. ISO8601: Date and time format. https://www.iso.org/iso-8601-date-and-time-format.html.
[12] Yu Gao, Wensheng Dou, Feng Qin, Chushu Gao, Dong Wang, Jun Wei, Ruirui Huang, Li Zhou, and Yongming Wu. 2018. An empirical study on crash recovery bugs in large-scale distributed systems. In *Proceedings of ESEC/FSE*, Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu (Eds.). 539–550.
[13] Joshua Garcia, Yang Feng, Junjie Shen, Sumaya Almanee, Yuan Xia, and Qi Alfred Chen. 2020. A Comprehensive Study of Autonomous Vehicle Bugs. In *Proceedings of ICSE*. 385–396.
[14] Brian E. Granger and Fernando Pérez. 2021. Jupyter: Thinking and Storytelling With Code and Data. *Computing in Science & Engineering* 23, 2 (2021), 7–14.
[15] IJuliaKernel. 2022. IJuliaKernel: Julia kernel for Jupyter. https://juliapackages.com/p/ijulia.
[16] IPykernel. 2022. IPykernel. https://github.com/ipython/ipykernel.
[17] IPython. 2022. IPython Parallel. https://github.com/ipython/ipyparallel.
[18] IRkernel. 2022. IRkernel: R kernel for Jupyter. https://github.com/IRkernel/IRkernel.
[19] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. 2019. A Comprehensive Study on Deep Learning Bug Characteristics. In *Proceedings of ESEC/FSE*. 510–520.
[20] Jupyter. 2015. Ipykernel pull#18. https://github.com/ipython/ipykernel/pull/18.
[21] Jupyter. 2015. Jupyter Notebook Issue#792. https://github.com/jupyter/notebook/issues/792.
[22] Jupyter. 2015. Jupyter Notebook Issue#96. https://github.com/jupyter/notebook/issues/96.
[23] Jupyter. 2015. Jupyter Notebook PR#22. https://github.com/jupyter/notebook/pull/22.
[24] Jupyter. 2015. Notebook issue#52. https://github.com/jupyter/notebook/issues/52.
[25] Jupyter. 2015. Notebook pull#799. https://github.com/jupyter/notebook/pull/799.
[26] Jupyter. 2016. Jupyter Notebook Issue#1003. https://github.com/jupyter/notebook/issues/1003.
[27] Jupyter. 2016. Jupyter Notebook PR#1011. https://github.com/jupyter/notebook/pull/1011.
[28] Jupyter. 2016. Notebook issue#1074. https://github.com/jupyter/notebook/issues/1074.
[29] Jupyter. 2016. Notebook issue#1547. https://github.com/jupyter/notebook/issues/1547.
[30] Jupyter. 2016. Notebook issue#1977. https://github.com/jupyter/notebook/issues/1977.
[31] Jupyter. 2016. Notebook pull#1652. https://github.com/jupyter/notebook/pull/1652.
[32] Jupyter. 2017. Jupyter Notebook Issue#2203. https://github.com/jupyter/notebook/issues/2203.
[33] Jupyter. 2017. Jupyterclient pull#254. https://github.com/jupyter/jupyter_client/pull/254.
[34] Jupyter. 2017. Notebook issue#2503. https://github.com/jupyter/notebook/issues/2503.
[35] Jupyter. 2017. Notebook pull#2159. https://github.com/jupyter/notebook/pull/2159.
[36] Jupyter. 2018. Notebook issue#3629. https://github.com/jupyter/notebook/issues/3629.
[37] Jupyter. 2018. Notebook pull#4236. https://github.com/jupyter/notebook/pull/4236.
[38] Jupyter. 2019. Ipykernel pull#390. https://github.com/ipython/ipykernel/pull/390.
[39] Jupyter. 2019. Jupyterserver Issue#42. https://github.com/jupyter-server/jupyter_server/issues/42.
[40] Jupyter. 2020. Jupyter Notebook Issue#5190. https://github.com/jupyter/notebook/issues/5190.
[41] Jupyter. 2020. Jupyter Notebook Issue#5502. https://github.com/jupyter/notebook/issues/5502.
[42] Jupyter. 2020. Jupyterclient Issue#591. https://github.com/jupyter/jupyter_client/issues/591.
[43] Jupyter. 2020. Jupytercore PR#183. https://github.com/jupyter/jupyter_core/pull/183.
[44] Jupyter. 2020. Notebook pull#5136. https://github.com/jupyter/notebook/pull/5136.
[45] Jupyter. 2021. Ipykernel Issue#694. https://github.com/ipython/ipykernel/issues/694.
[46] Jupyter. 2021. Ipykernel Issue#742. https://github.com/ipython/ipykernel/issues/742.
[47] Jupyter. 2021. Jupyterclient pull#607. https://github.com/jupyter/jupyter_client/pull/607.
[48] Jupyter. 2021. Jupyterclient pull#703. https://github.com/jupyter/jupyter_client/pull/703.
[49] Jupyter. 2021. Jupyterclient pull#717. https://github.com/jupyter/jupyter_client/pull/717.
[50] Jupyter. 2021. Jupyterserver Issue#591. https://github.com/jupyter-server/jupyter_server/issues/59
[51] Jupyter. 2021. Jupyterserver pull#380. https://github.com/jupyter-server/jupyter_server/pull/380.
[52] Jupyter. 2021. Jupyterserver pull#473. https://github.com/jupyter-server/jupyter_server/pull/473.
[53] Jupyter. 2021. Jupyterserver pull#482. https://github.com/jupyter-server/jupyter_server/pull/482.
[54] Jupyter. 2021. Jupyterserver pull#521. https://github.com/jupyter-server/jupyter_server/issues/521.
[55] Jupyter. 2021. Notebook pull#6160. https://github.com/jupyter/notebook/pull/6160.
[56] Jupyter. 2022. Basic Steps to Run Jupyter. https://docs.jupyter.org/en/latest/running.html#basic-steps.
[57] Jupyter. 2022. Detailed POST Inofrmation of Jupyterserver. https://petstore.swagger.io/?url=https://raw.githubusercontent.com/jupyter/jupyter_server/master
[58] Jupyter. 2022. Jupyter. https://jupyter.org/.
[59] Jupyter. 2022. Jupyter Notebook. https://github.com/jupyter/notebook.
[60] Jupyter. 2022. jupyterclient. https://github.com/jupyter/jupyter_client.
[61] Jupyter. 2022. jupytercore. https://github.com/jupyter/jupyter_core.
[62] Jupyter. 2022. Jupyterserver. https://github.com/jupyter-server/jupyter_server.
[63] Jupyter. 2022. Jupyterserver Architecture. https://jupyter-server.readthedocs.io/en/latest/developers/architecture.html.
[64] Jupyter. 2022. Launching a bare Jupyterserver. https://jupyter-server.readthedocs.io/en/latest/users/launching.html.
[65] Jupyter. 2022. Workflow in Jupyterserver. https://jupyter-server.readthedocs.io/en/latest/developers/architecture.html#create-session-workf
[66] DaYe Kang, Tony Ho, Nicolai Marquardt, Bilge Mutlu, and Andrea Bianchi. 2021. ToonNote: Improving Communication in Computational Notebooks Using Interactive Data Comics. In *Proceedings CHI*.
[67] Mary Beth Kery and Brad A. Myers. 2018. Interactions for Untangling Messy History in a Computational Notebook. In *Symposium on VL/HCC*. 147–155.
[68] D. E. Knuth. 1984. Literate Programming. *Comput. J.* 27, 2 (1984), 97–111.
[69] Andreas P. Koenzen, Neil A. Ernst, and Margaret-Anne D. Storey. 2020. Code Duplication and Reuse in Jupyter Notebooks. In *Symposium on VL/HCC*. 1–9.
[70] David Koop and Jay Patel. 2017. Dataflow notebooks: encoding and tracking dependencies of cells. In *9th USENIX Workshop on TaPP*.
[71] Xingjun Li, Yuanxin Wang, Hong Wang, Yang Wang, and Jian Zhao. 2021. NBSearch: Semantic Search and Visual Exploration of Computational Notebooks. In *Proceedings of CHI*.
[72] George Mathew and Kathryn T. Stolee. 2021. Cross-language code search using static and dynamic analyses. In *Proceedings of ESEC/FSE*. 205–217.
[73] nbconvert. 2022. nbconvert: Jupyter Notebook Conversion. https://github.com/jupyter/nbconvert.
[74] Hai Nguyen, David A Case, and Alexander S Rose. 2018. NGLview–interactive molecular graphics for Jupyter notebooks. *Bioinformatics* 34, 7 (2018), 1241–1242.
[75] Fernando Perez and Brian E. Granger. 2007. IPython: A System for Interactive Scientific Computing. *Computing in Science Engineering* 9 (2007), 21–29.
[76] João Felipe Pimentel, Leonardo Murta, Vanessa Braganholo, and Juliana Freire. 2019. A Large-Scale Study About Quality and Reproducibility of Jupyter Notebooks. In *Proceedings of MSR*. 507–517.
[77] pydeps. 2022. pydeps. https://pydeps.readthedocs.io/.
[78] Python. 2022. Python regex library. https://docs.python.org/3/library/re.html.
[79] Mohammed Suhail Rehman. 2019. Towards understanding data analysis workflows using a large notebook corpus. In *Proceedings of ICMD*. 1841–1843.
[80] Alan Romano, Xinyue Liu, Yonghwi Kwon, and Weihang Wang. 2021. An Empirical Study of Bugs in WebAssembly Compilers. In *Proceedings of ASE*. 42–54.
[81] Adam Rule, Ian Drosos, Aurélien Tabard, and James D Hollan. 2018. Aiding collaborative reuse of computational notebooks with annotated cell folding. *Proceedings of CHI* (2018), 1–12.
[82] Adam Rule, Aurélien Tabard, and James D Hollan. 2018. Exploration and explanation in computational notebooks. In *Proceedings of CHI*. 1–12.
[83] Sheeba Samuel and Birgitta König-Ries. 2021. ReproduceMeGit: A Visualization Tool for Analyzing Reproducibility of Jupyter Notebooks. In *Proceedings of IPAW*,

Vol. 12839. 201–206.
[84] Carolyn B. Seaman, Forrest Shull, Myrna Regardie, Denis Elbert, Raimund L. Feldmann, Yuepu Guo, and Sally Godfrey. 2008. Defect Categorization: Making Use of a Decade of Widely Varying Historical Data. In *Proceedings of ESEM*. 149–157.
[85] Helen Shen. 2014. Interactive notebooks: Sharing the code. *Nature* 515, 7525 (2014), 151–152.
[86] Qingchao Shen, Haoyang Ma, Junjie Chen, Yongqiang Tian, Shing-Chi Cheung, and Xiang Chen. 2021. A comprehensive study of deep learning compiler bugs. In *Proceedings of ESEC/FSE*, Diomidis Spinellis, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta (Eds.). 968–980.
[87] Ferdian Thung, Shaowei Wang, David Lo, and Lingxiao Jiang. 2012. An Empirical Study of Bugs in Machine Learning Systems. In *Proceedings of ISSRE*. 271–280.
[88] Tornado. 2022. Tornado: a Python web framework. https://www.tornadoweb.org/en/stable/.
[89] Traitlets. 2021. Traitlets framework. https://github.com/ipython/traitlets.
[90] Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar Devanbu, and Vladimir Filkov. 2015. Quality and Productivity Outcomes Relating to Continuous Integration in GitHub. In *Proceedings of ESEC/FSE*. 805–816.
[91] Dinghua Wang, Shuqing Li, Guanping Xiao, Yepang Liu, and Yulei Sui. 2021. An exploratory study of autopilot software bugs in unmanned aerial vehicles. In *Proceedings of ESEC/FSE*. 20–31.
[92] Jiawei Wang, Tzu-yang Kuo, Li Li, and Andreas Zeller. 2020. Assessing and Restoring Reproducibility of Jupyter Notebooks. In *Proceedings of ASE*. 138–149.
[93] Jiawei Wang, Li Li, and Andreas Zeller. 2020. Better Code, Better Sharing: On the Need of Analyzing Jupyter Notebooks. In *Proceedings of the ICSE-NIER*. 53–56.
[94] Jiawei Wang, Li Li, and Andreas Zeller. 2021. Restoring Execution Environments of Jupyter Notebooks. In *Proceedings of ICSE*. 1622–1633.
[95] webpack. 2022. Webpack: a module bundler for JavaScript. https://webpack.js.org/.
[96] Nathaniel Weinman, Steven M Drucker, Titus Barik, and Robert DeLine. 2021. Fork It: Supporting stateful alternatives in computational notebooks. In *Proceedings of CHI*. 1–12.
[97] ZeroMQ. 2022. ZeroMQ. https://zeromq.org/.
[98] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. 2018. An Empirical Study on TensorFlow Program Bugs. In *Proceedings of ISSTA*. 129–140.
[99] Chenguang Zhu, Ripon K. Saha, Mukul R. Prasad, and Sarfraz Khurshid. 2021. Restoring the Executability of Jupyter Notebooks by Automatic Upgrade of Deprecated APIs. In *Proceedings of ASE*. 240–252.