Source-Code Support for Replicating a Bugs Dataset

Samuel Grayson
Dept. of Computer Science
University of Illinois at Urbana Champaign
Urbana, IL
orcid.org/0000-0001-5411-356X

Faustino Aguilar
Dept. of Computer Engineering
University of Panama
Panama City, Panama
orcid.org/0009-0000-1375-1143

Darko Marinov
Dept. of Computer Science
University of Illinois at Urbana Champaign
Urbana, IL
orcid.org/0000-0001-5023-3492

***Abstract*—We present our experience on replicating a bug dataset for the Python programming language. We assess the reproducibility of the original dataset less than three years after its original publication. The bug dataset provides some information about the software environment, but this information can be incomplete or it can decay into something uninstallable. We rectify as many of these problems as we can and redesign the original dataset to be more easily reusable and reproducible by future authors. Based on our experience, we offer suggestions to Python artifact authors to improve their reproducibility.**

***Index Terms*—reproducibility, bug database, python**

I. Introduction

BugsInPy˜[1] is a curated dataset of real-world bugs in large Python projects. BugsInPy is intended to be used by researchers to develop and evaluate software testing and debugging tools for Python on a diverse set of real-world bugs from multiple projects. This can help to ensure that the tools are effective in detecting real-world bugs.

The BugsInPy dataset contains a variety of information about each bug, including:

- A buggy commit
- A fixed commit
- Python version used
- Test cases that indicate the bugs presence

BugsInPy includes a database abstraction layer and a test execution framework. The database abstraction layer provides a way to access the dataset in a structured way. The test execution framework allows researchers to run the test cases that reveal the bugs.

BugsInPy is a valuable resource for researchers who are developing and evaluating software defect detection tools. The dataset provides a large and diverse set of bugs that can be used to test the effectiveness of these tools. BugsInPy also includes a variety of tools and resources that can help researchers to use the dataset effectively.

We sought to use the BugsInPy framework to verify that the test cases could be set up, that the buggy commit fails, and that the fixed commit passes. This is a *reproduction*˜[1] of the original work, since we are using the original framework.

Our contributions are:

- A new script which can invoke the BugsInPy framework *en masse*.
- Modifications to the BugsInPy framework to install and use the correct version of Python.
- The results of which bugs were reproducible (software environment installs, buggy version fails exactly that one test, fixed version passes all tests).

This article proceeds with the methodology section, which explains what tools we used to run these codes reproducible. Then we summarize the results of our executions and analyze the failures. Finally, we engage in an open-ended discussion of our experiment with advice to authors of reproducible artifacts and those seeking to reproduce artifacts.

II. Methodology

We use a provide a container build script to provide a consitent starting point in which our scripts will install the correct software environment. Running in a container also sandboxes modifications that the BugsInPy script wants to make (e.g., modifying `~/.bashrc`). While this image is available in this registry[2], we suggest users build the image themselves rather than depending on this registry to remain available.

For each bug, The BugsInPy script ignores the specified version of Python, deferring to the system default Python instead. Presumably, the BugsInPy authors manually changed their system's version of Python according to the specification of each bug, but this is not an automated process, and that makes it difficult for future users. We modified this script to install the correct version of Python using Conda. Conda is a cross-platform package manager. Packages installed by Conda neither use nor modify the system version of those packages, so Conda can support different environments each with their own possibly conflicting requirements. Conda package repositories store packages containing pre-compiled binaries and metadata for each platform, so installing is rather quick.

The original BugsInPy scripts install all Python packages using Pip. Pip is the default package builder and installer for Python. Pip can invoke the compiler to build dependencies from source [3] or download prebuilt binary files. Projects do this for their own C source, however they don't do this for shared libraries, because the build system can be quite complex, and system

administrators would rather have one version of that shared library on the system managed by the system package manager.

System administrators prefer to rely on the system package manager for managing shared libraries because it ensures consistency, stability, security, efficient resource utilization, and ease of administration. By delegating the responsibility of library management to the system package manager, administrators can leverage the robust infrastructure provided by the operating system and reduce the complexities associated with building and maintaining shared libraries from source within individual projects.

It's worth noting that the most common usage of pip is to install packages from the Python Package Index (PyPI) using requirement specifiers. As specified in the official Python Packaging documentation, a requirement specifier typically consists of a project name followed by an optional version specifier. The specification for requirement specifiers can be found in PEP 440, which provides a comprehensive guide to the currently supported specifiers˜[4].

Therefore, some packages may require additional system libraries or dependencies that cannot be installed solely through pip. For example, Matplotlib, a popular Python plotting library, has certain system-level dependencies that pip cannot automatically handle. These dependencies typically include external libraries such as libpng, freetype, or Tk, which are required for Matplotlib to function properly˜[5]. Consequently, if a bug in a project's environment depends on Matplotlib, attempting to install and run that project on a clean machine without the necessary system libraries would result in installation failures.

It is important to note that in such cases, it becomes the responsibility of the user or system administrator to ensure that the required system libraries are installed manually before attempting to install the package with pip. The Matplotlib documentation provides detailed instructions on how to install the necessary system-level dependencies for different platforms˜[6]. By following these instructions and setting up the required libraries, users can successfully install and utilize Matplotlib and any other package with similar external dependencies.

Presumably the original authors manually modified their system to have these system libraries; in our case, we identify packages that Pip cannot install on vanilla Debian and simply install those with Conda instead.

Building the environment from source can be quite costly, so we reuse environments when the Python package requirements and Python versions are exactly the same.

The following is a pseudo-code representing the approach used to reproduce the BugsInPy dataset:

```
for each project in BugsInPy dataset:
  For each bug for that project:
    For version in {buggy, fixed}:
      Checkout the buggy version
      Env ID := hash(requirements.txt + python version)
      If no env with Env ID already exists:
        Use Conda to set up the env
      Activate env with Env ID
      Report environment errors
      Run tests
      Report test results
      Deactivate env
  done
done
```

The provided code snippet demonstrates a common approach to managing environments by using a hash based on the project's Python package requirements and Python version. Here is a high-level explanation of the code's functionality:

1. The code first checkout the respective version (buggy or fixed) of the project code to analyze and sets up the environment by sourcing the configuration from the conda package manager. This configuration allows the script to utilize conda-specific commands and environment variables. To run checkout the project we use the BugsInPy framework script `bugsinpy-checkout`
2. Next, a unique hash is generated based on the project's Python package requirements and the specified Python version. This hash is typically created using a combination of the project's requirements file and the Python version. By combining these elements and generating a hash, a unique identifier is created for the specific environment.
3. The script checks if an environment with the generated hash name already exists using the conda `env list` command. This step ensures that duplicate environments are not created unnecessarily.
4. If the environment does not exist, the script proceeds to create a new conda environment. The environment is named using the generated hash and is configured with the specified Python version and any additional required packages. If an environment with the same ID already exists, skip environment creation and continue with the next bug or version.
5. Activate the environment associated with the Env ID. This step ensures that the subsequent steps are executed within the desired environment.
6. Report any environment errors or issues encountered during the environment setup process.
7. Run tests or any other required analysis on the project code within the activated environment. To run the tests we use the BugsInPy framework script `bugsinpy-test`
8. Report the results of the tests, including any bugs or failures identified during the analysis.
9. Deactivate the environment to conclude the analysis for the current bug and version.
10. Move on to the next bug or version in the iteration until all bugs and versions for the project have been analyzed.
11. Repeat the above steps for each project in the BugsInPy dataset.

By using a hash-based approach, the code enables environment

reuse when the Python package requirements and Python version remain the same. This optimization helps reduce the time and resources required for environment setup, as the costly process of building environments from source can be bypassed. Instead, existing environments can be reused, providing a more efficient workflow for developers and ensuring consistency in the project's execution environment.

III. Results

A. Why the percentage of success and failure for the reproducibility of bugs?

The percentage of success and failure in reproducing bugs in the BugsInPy dataset provides valuable insights into the overall reproducibility of the bugs. Success indicates that the bug was successfully reproduced and the corresponding test passed, verifying the correctness of the fix. On the other hand, failure indicates that either the bug could not be reproduced or the test failed, indicating the need for further investigation or fixes.

The success and failure percentages offer important information about the quality and reliability of the codebase, as well as the effectiveness of the bug fixing process. They help identify projects with higher rates of successful bug reproduction, indicating better code quality and easier bug fixing processes. Analyzing these percentages can inform project maintainers about areas that require attention and improvement. It helps prioritize bug fixes, identify patterns in the types of bugs encountered, and allocate resources to improve the overall reliability and stability of the software.

The following is a table with the percentage of bug reproducibility for each project:

TABLE I
EXECUTION OF BUGGY VERSION IN BUGSINPY DATASET

result	error	fail	pass	total	fail%
(PySnooper, buggy)	1	1	1	3	33.33
(ansible, buggy)	0	18	0	18	100.00
(black, buggy)	0	22	1	23	95.65
(cookiecutter, buggy)	0	4	0	4	100.00
(fastapi, buggy)	0	16	0	16	100.00
(httpie, buggy)	0	5	0	5	100.00
(keras, buggy)	3	41	1	45	91.11
(luigi, buggy)	0	33	0	33	100.00
(matplotlib, buggy)	3	27	0	30	90.00
(pandas, buggy)	1	168	0	169	99.41
(sanic, buggy)	0	5	0	5	100.00
(scrapy, buggy)	0	40	0	40	100.00
(spacy, buggy)	1	9	0	10	90.00
(thefuck, buggy)	0	32	0	32	100.00
(tornado, buggy)	0	16	0	16	100.00
(tqdm, buggy)	0	9	0	9	100.00
(youtube-dl, buggy)	0	43	0	43	100.00
Total (Buggy)	9	489	3	501	97.60

The tables provided above showcase the percentages of bug reproducibility for each project, categorized as "fail," "pass," or "error." The term "fail" refers to an expected failure in the buggy version, indicating successful bug reproduction. Conversely,

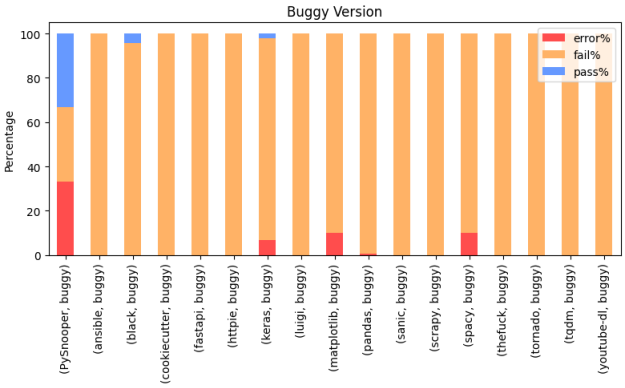

Fig. 1. Reproducibility Results - Buggy Version

TABLE II
EXECUTION OF FIXED VERSION IN BUGSINPY DATASET

result	error	fail	pass	total	pass%
(PySnooper, fixed)	1	0	2	3	66.67
(ansible, fixed)	0	0	18	18	100.00
(black, fixed)	0	0	23	23	100.00
(cookiecutter, fixed)	0	0	4	4	100.00
(fastapi, fixed)	0	0	16	16	100.00
(httpie, fixed)	0	0	5	5	100.00
(keras, fixed)	3	0	42	45	93.33
(luigi, fixed)	0	6	27	33	81.82
(matplotlib, fixed)	3	1	26	30	86.67
(pandas, fixed)	4	0	165	169	97.63
(sanic, fixed)	0	0	5	5	100.00
(scrapy, fixed)	0	2	38	40	95.00
(spacy, fixed)	1	0	9	10	90.00
(thefuck, fixed)	0	0	32	32	100.00
(tornado, fixed)	0	0	16	16	100.00
(tqdm, fixed)	0	0	9	9	100.00
(youtube-dl, fixed)	0	0	43	43	100.00
Total (Fixed)	12	9	480	501	95.81

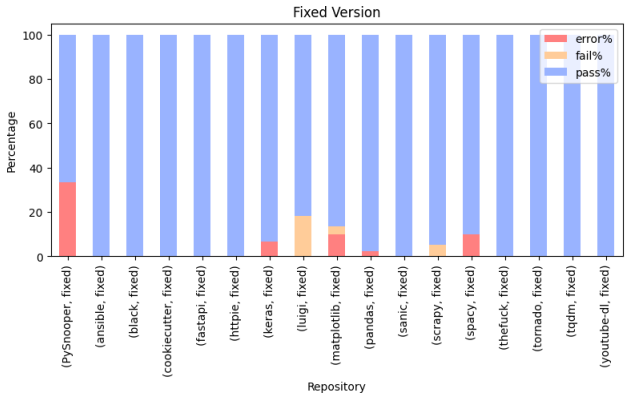

Fig. 2. Reproducibility Results - Fixed Version

"pass" indicates the successful resolution of the bug in the fixed version, with corresponding test cases passing. "Error" represents cases where a hard error prevents the execution of the test. Understanding these categories is essential for interpreting the reproducibility results accurately and assessing the effectiveness of the bug fixing process.

The results demonstrate that the overall reproducibility of bugs in the BugsInPy dataset has improved significantly, with over 95% of all bugs being successfully reproduced and passing the tests. This indicates a high level of confidence in the correctness of the bug fixes and the reliability of the codebase.

Analyzing the specific project results, we can observe variations in the success rates. Some projects, such as Ansible, Cookiecutter, FastAPI, Httpie, Sanic, Thefuck, Tornado, and Tqdm, show a 100% success rate in reproducing and passing the bug tests for both the buggy and fixed versions. On the other hand, projects like Pandas, Keras, Scrapy, and Matplotlib have a lower success rate in reproducing and passing the bug tests.

These results provide valuable insights into the quality, reliability, and effectiveness of the bug fixing process for each project. Project maintainers can utilize these findings to prioritize bug fixes, allocate resources for improving code quality, and enhance the overall reliability and stability of the software.

By achieving a high success rate in reproducing bugs and passing tests, the BugsInPy datasetdemonstrates the effectiveness of the bug fixing process and highlights the overall quality of the codebase. With over 95% of bugs being successfully reproduced and passing the tests, developers can have a high degree of confidence in the correctness of the fixes and the reliability of the software.

The success rate also indicates that the projects in the BugsInPy dataset have undergone rigorous testing and debugging processes. This level of thoroughness contributes to improved software quality and can inspire trust among users and stakeholders.

The variations in success rates among different projects provide further insights. Projects with a 100% success rate in reproducing and passing tests demonstrate excellent code quality and a robust bug fixing process. Conversely, projects with a lower success rate may require additional attention and improvements in their debugging and testing practices.

The total line aggregates the numbers from all projects and calculates the overall percentages of success or failure for the reproducibility of bugs.

The table below presents the running time required to reproduce bugs in each project within the BugsInPy dataset, along with the respective containers:

The projects are sorted based on their running time in descending order, with the project "pandas" having the highest running time of 963 minutes, followed by "luigi," "scrapy," and so on.

TABLE III
REPRODUCTION TIME FOR BUGS IN EACH PROJECT

Project	Running Time (minutes)	Container
pandas	963	/bugsinpy-pandas-1
luigi	510	/bugsinpy-luigi-1
scrapy	268	/bugsinpy-scrapy-1
keras	230	/bugsinpy-keras-1
black	214	/bugsinpy-black-1
fastapi	197	/bugsinpy-fastapi-1
thefuck	195	/bugsinpy-thefuck-1
sanic	136	/bugsinpy-sanic-1
spacy	131	/bugsinpy-spacy-1
ansible	80	/bugsinpy-ansible-1
tqdm	36	/bugsinpy-tqdm-1
youtube-dl	59	/bugsinpy-youtube-dl-1
cookiecutter	40	/bugsinpy-cookiecutter-1
httpie	39	/bugsinpy-httpie-1
matplotlib	26	/bugsinpy-matplotlib-1
tornado	14	/bugsinpy-tornado-1
pysnooper	5	/bugsinpy-pysnooper-1

The running time to reproduce the bugs varies across different projects in the BugsInPy dataset. The reproduction process was conducted on a virtual machine (VM) with 8GB of RAM, 4 x86 cores, and 100GB of free disk space. The time taken for bug reproduction depends on various factors, including the complexity of the codebase, the number of bugs in the project, and the specific characteristics of each bug.

Additionally, it is noteworthy that the "pandas" project has the most bugs registered in the dataset, with a total of 169 bugs out of the 501 bugs analyzed. The reproduction of bugs in the "pandas" project required a relatively longer running time, taking approximately 963 minutes. The extended time may be attributed to factors such as the complexity of the codebase, the number of bugs present in the project, and the specific characteristics of the bugs encountered during the reproduction process.

Please note that the provided running times are specific to the reproduction process on the given VM configuration, which had 8GB of RAM, 4 cores, and 100GB of free disk space. Reproduction times can vary depending on hardware resources, system configurations, and other environmental factors.

The inclusion of Python version information in the bug dataset is crucial for ensuring the reproducibility of tests executed for bug detection and fixing. When developers encounter a bug, having access to the specific Python version used during its occurrence enables them to reproduce the bug in a controlled environment.

The table provides a summary of Python versions and their corresponding counts in the bug dataset mentioned in the "bugsinpy" paper. The dataset contains bug information for various Python repositories, with each bug having a corresponding "bug.info" file specifying the Python version.

TABLE IV
PYTHON VERSIONS IN BUGSINPY DATASET

python_version	count	percentage
3.6.9	31	6.19%
3.7.0	58	11.58%
3.7.3	50	9.98%
3.7.4	33	6.59%
3.7.7	9	1.80%
3.8.1	33	6.59%
3.8.3	287	57.28%
Total	501	100%

The table includes the Python version, the count of bugs associated with each version, and the percentage of bugs represented by each version out of the total count. The percentages give us insights into the distribution of bugs across different Python versions in the dataset.

Analyzing the table, we observe that the majority of bugs (57.28%) are reported in Python version 3.8.3, followed by version 3.7.0 (11.58%). Versions 3.7.3, 3.6.9, and 3.8.1 account for approximately 9.98%, 6.19%, and 6.59% of the bugs, respectively. Additionally, versions 3.7.4 and 3.7.7 contribute to 6.59% and 1.80% of the bugs, respectively.

These percentages provide valuable insights into the distribution of bugs among different Python versions in the dataset. By referring to the "bugsinpy" paper and the associated bug.info files, further analysis can be performed to understand any patterns or trends related to specific Python versions and their corresponding bug occurrences.

By using the same Python version specified in the bug, developers can accurately replicate the conditions under which the bug manifested. This consistency in the Python environment helps in understanding the root cause of the bug and facilitates the debugging process. It allows developers to examine the code, libraries, and dependencies associated with that particular Python version, increasing the likelihood of identifying and resolving the issue effectively.

Reproducibility plays a significant role in software development, particularly in bug fixing and testing. Having access to the exact Python version used during the bug occurrence enhances the accuracy and reliability of the testing process. Developers can execute the same code with the same environment, ensuring that any fixes or improvements applied can be tested and validated consistently.

In summary, the inclusion of Python version information in the bug dataset not only provides insights into the distribution of bugs across different versions but also enables developers to reproduce and investigate the bugs more effectively. This contributes to improved bug detection, diagnosis, and ultimately, the development of more reliable and stable software systems.

IV. DISCUSSION

A. What makes our reproduction easy?

The ease of bug reproduction in the BugsInPy dataset can be attributed to several factors:

1. Standardized and Automated Approach: The `bugsinpy-testall` script provides a standardized and automated approach to reproducing and testing bugs in Python projects. By automating essential steps such as environment setup, code checkout, compilation (if required), and test execution, the script streamlines the reproduction process and minimizes manual effort.
2. Conda for Dependency Management: Leveraging Conda as the package and environment manager contributes to easy management of project dependencies. Conda ensures consistent environments across different systems, simplifying the setup process and minimizing compatibility issues.
3. Comprehensive Logging: The `bugsinpy-testall` script maintains detailed logs of the test results, storing them in `~/projects/output.csv`. This centralized logging system enables easy analysis of the test outcomes, provides a historical record for reference, and facilitates tracking the progress of bug reproduction.

These factors collectively contribute to the ease of reproducing bugs in the BugsInPy dataset, providing a reliable and efficient framework for bug analysis and investigation.

B. What makes our reproduction hard?

Despite the facilitative factors mentioned above, bug reproduction can still present challenges due to the following factors:

1. Complex Codebases: Some projects in the BugsInPy dataset may have intricate and extensive codebases, making it challenging to identify the specific code sections or interactions responsible for the bugs. Analyzing complex codebases requires careful examination and understanding of the project's architecture and design.
2. Interdependencies and Compatibility: Projects often rely on external libraries, tools, and frameworks. Interdependencies and compatibility issues between different versions of these dependencies can complicate the bug reproduction process. Ensuring that all the necessary dependencies are correctly installed and compatible with each other can be time-consuming and require additional effort.
3. Intermittent or Context-Specific Bugs: Some bugs may manifest only under specific conditions or in certain environments. Reproducing these intermittent or context-specific bugs consistently can be challenging. It may require identifying and setting up the precise conditions required for the bug to occur, which can be complex and time-consuming.
4. Resource Constraints: Projects with a large number of bugs or a high failure rate can pose challenges in terms

of the time and resources required for bug reproduction. Reproducing and testing a significant number of bugs within limited resources may result in longer reproduction times and potential resource limitations.

Addressing these challenges requires careful consideration of project-specific factors and may involve additional research, debugging techniques, and resources to ensure accurate and reliable bug reproduction.

C. Recommendations to BugsInPy users

For users of the BugsInPy dataset, the following recommendations can enhance their bug reproduction and analysis experience:

1. Thorough Documentation: Provide detailed documentation accompanying the dataset, including clear instructions on setting up the required environments, dependencies, and any additional configuration steps. This documentation should cover the specific steps necessary to reproduce bugs successfully.
2. Reproducibility Guidelines: Include guidelines or best practices for bug reproduction to ensure consistent and reliable results across different users and systems. These guidelines can cover aspects such as creating isolated environments, using version control for code checkout, and capturing relevant debugging information.
3. Collaboration and Community Feedback: Foster a collaborative environment among BugsInPy users to share insights, experiences, and potential challenges faced during bug reproduction. Encouraging community participation and feedback can help identify common issues, improve reproducibility practices, and provide support to fellow users.

D. Recommendations to artifact authors

For authors providing artifacts in the BugsInPy dataset, the following recommendations can enhance the reproducibility of their artifacts:

1. Detailed Artifact Descriptions: Provide comprehensive documentation accompanying the artifacts, including detailed instructions on setting up the required environments, dependencies, and specific configuration steps. This documentation should guide users in reproducing the bugs effectively.
2. Version Control: Ensure that the artifact repository includes version control information for the codebase, making it easier for users to checkout and analyze specific versions.
3. Bug Context Information: Include relevant information about the bug context, such as the steps to trigger the bug, expected behavior, and any special conditions required for reproduction. This information assists users in replicating the bugs accurately.

E. Threats to Validity

1) Is it possible that our reproduction is not working when it should be?: While the BugsInPy dataset aims to provide reproducible bugs, there may be cases where bug reproduction does not yield the expected results. Several factors can contribute to such situations:

1. Misconfiguration or Environmental Variations: Differences in environmental configurations or setups between the original bug reports and the user's system can impact bug reproduction. Variations in dependencies, system configurations, or codebase versions can result in discrepancies in the bug reproduction process.
2. Incomplete Bug Information: In some cases, the bug reports or associated artifacts may lack crucial details or steps necessary for accurate reproduction. Incomplete or ambiguous bug information can hinder the reproducibility process and lead to inconsistent results.

To mitigate these threats, clear and detailed documentation, version control information, and collaboration within the BugsInPy community are crucial. Users should carefully follow the provided instructions, document any deviations or issues encountered during bug reproduction, and actively engage in discussions to address any uncertainties or challenges.

2) Is it possible that our reproduction won't be reproducible by others?: Reproducibility can also be influenced by the unique environments and configurations of different users. Factors such as variations in operating systems, library versions, hardware capabilities, and other system-specific settings can impact the reproducibility of bugs.

To enhance the reproducibility of bugs by others, it is essential to provide detailed and comprehensive documentation that includes:

1. Simple Environment Setup Instructions: Provide detailed instructions for setting up the required environment, including the specific versions of Python, libraries, and tools. Specify any additional dependencies or configurations necessary for accurate bug reproduction.
2. Version Control Information: Ensure that the artifact repository includes version control information, such as the specific commit or tag used for the bug reproduction. This allows users to precisely replicate the codebase used during bug identification and fixing.
3. Reproduction Steps: Clearly outline the steps required to reproduce the bug, including any necessary inputs, specific code paths, or special conditions. Provide sample code or scripts that demonstrate the bug behavior to aid in accurate bug reproduction.
4. Debugging Information: Document any relevant debugging information or techniques employed during the bug reproduction process. This can include log files, stack traces, or other diagnostic outputs that can assist users in understanding and resolving the bug.

5. Collaboration and Support Channels: Establish channels for users to seek support and engage in discussions related to bug reproduction. This can be in the form of mailing lists, forums, or dedicated chat platforms where users can share their experiences, ask questions, and receive assistance from the artifact authors and the BugsInPy community.

In addition to the previous discussion, it is worth mentioning that the BugsInPy dataset initially relied on the original container˜[7]. This container was based on Ubuntu 18.04 and presented several issues when attempting to activate the environment.

Some of the issues encountered included misconfigured paths for Python and the environment, which hindered the successful activation of the environment. Despite our efforts to address these issues by attempting to fix the outdated environment and unsupported libraries, it became apparent that a more robust solution was required.

To overcome these challenges, we opted to create our own Docker image based on a well-maintained and supported image recommended by the official Anaconda documentation˜[8]. By using a reliable and up-to-date Docker image, we were able to ensure a stable and functional environment for reproducing the bugs in the BugsInPy dataset.

This decision not only resolved the issues faced with the original container but also provided a more reliable foundation for the bug reproduction process. It allowed us to create a consistent and controlled environment that facilitated accurate bug reproduction and reliable test results.

By transitioning to our custom Docker image based on a supported Anaconda environment, we enhanced the reproducibility and reliability of the bug reproduction process in BugsInPy.

V. Conclusion

The study presented in this paper demonstrates the effectiveness of the BugsInPy dataset in reproducing and testing bugs in Python projects. The standardized and automated approach provided by the `bugsinpy-testall` script, coupled with the use of Conda for dependency management, streamlines the bug reproduction process and enhances its ease.

The high success rate in reproducing bugs, with over 95% of bugs passing the tests, indicates the reliability and accuracy of the bug fixes in the dataset. This is evident from the results table below:

TABLE V
Total Reproduction in BugsInPy Dataset

result	error	fail	pass	total	results%
Total (Buggy)	9	489	3	501	97.60% fail
Total (Fixed)	12	9	480	501	95.81% pass

The table shows that out of the 501 bugs analyzed, 97.60% of the bugs were classified as failures (fail), indicating the presence of issues or errors in the codebase. On the other hand, 95.81% of the bugs were classified as successful (pass), indicating that the bug fixes were effective in resolving the reported issues.

However, challenges still exist in reproducing certain bugs due to complex codebases, interdependencies, intermittent nature, and resource constraints. Addressing these challenges requires further research, debugging techniques, and allocation of appropriate resources.

To improve the reproducibility of BugsInPy, it is recommended to enhance the documentation and description of software environments, making reproducibility tools more accessible and user-friendly. Collaboration within the BugsInPy community and engagement with artifact authors can also foster a supportive and collaborative environment for addressing challenges and enhancing the reproducibility process.

In conclusion, the BugsInPy dataset provides a valuable resource for studying and understanding bugs in Python projects. By incorporating the recommendations provided and addressing the threats to validity, researchers and practitioners can leverage this dataset to improve software reliability, enhance bug fixing processes, and further advance the field of software engineering research.

References

[1] R. Widyasari, S. Q. Sim, C. Lok, *et al.*, "BugsInPy: A database of existing bugs in python programs to enable controlled testing and debugging studies," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020, New York, NY, USA: Association for Computing Machinery, Nov. 8, 2020, pp. 1556–1560, ISBN: 978-1-4503-7043-1. DOI: 10.1145/3368089.3417943. [Online]. Available: https://doi.org/10.1145/3368089.3417943 (visited on 07/08/2023).

[2] "Faustinoaq/bugsinpy-testall - docker image — docker hub." (), [Online]. Available: https://hub.docker.com/r/faustinoaq/bugsinpy-testall (visited on 07/17/2023).

[3] "Pip install - pip documentation v23.2." (), [Online]. Available: https://pip.pypa.io/en/stable/cli/pip_install/#cmdoption-no-binary (visited on 07/17/2023).

[4] "Installing packages — python packaging user guide." (), [Online]. Available: https://packaging.python.org/en/latest/tutorials/installing-packages/#installing-from-pypi (visited on 07/17/2023).

[5] "Installation — matplotlib 3.7.2 documentation." (), [Online]. Available: https://matplotlib.org/stable/users/installing/index.html (visited on 07/17/2023).

[6] "Contributing — matplotlib 3.7.2 documentation." (), [Online]. Available: https://matplotlib.org/stable/devel/index.html#building-matplotlib (visited on 07/17/2023).

[7] "Soarsmu/bugsinpy - docker image — docker hub." (), [Online]. Available: https://hub.docker.com/r/soarsmu/bugsinpy (visited on 07/17/2023).
[8] "Docker — anaconda documentation." (), [Online]. Available: https://docs.anaconda.com/free/anaconda/applications/docker/ (visited on 07/17/2023).

APPENDIX
CODE, DATA, AND REPRODUCING

A snapshot of the latest state of this code can be found at: https://github.com/reproducing-research-projects/BugsInPy/releases/tag/v0.0.1.

A rolling release of the code can be found at: https://github.com/reproducing-research-projects/BugsInPy.

In the snapshot:

- `v0.0.1.zip` holds a human-readable BugsInPy framework code compressed in zip format.
- `v0.0.1.tar.gz` holds a human-readable BugsInPy framework code compressed in gzip format.

In the rolling release:

- `Dockerfile` docker file setup to build projects images.
- `docker-compose.yml` docker orchestration to run projects containers.
- `framework/bin/bugsinpy-testall` script to automate execution of BugsInPy framework scripts.

To reproduce all bugs in httpie for example, run:

```
# rm projects/bugsinpy-index.csv
# docker compose up setup httpie --build
Cleaning up temp folder ...
Reproducing bugs please wait ...
------------------------
httpie,1,buggy,fail
httpie,1,fixed,pass
httpie,2,buggy,fail
httpie,2,fixed,pass
httpie,3,buggy,fail
httpie,3,fixed,pass
httpie,4,buggy,fail
httpie,4,fixed,pass
httpie,5,buggy,fail
httpie,5,fixed,pass
```

After which, the results will be here:

- `bugsinpy-index.csv` This is the new result index.