Reproduction and Improvement on the BugsInPy Dataset

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

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

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* [2] 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 would like to answer the following questions:

RQ1. How many of the bugs in BugsInPy are reproudcible with no "extra" work? For a bug to be reproducible, the software environment should install without failure, the buggy verison should fail the identified test case, and the fixed version should pass.

RQ2. How many of those which are not reproducible can we rescue? We rescue a bug by modifying the scripts and data such that the bug is reproducible.

As part of our rescue, we made the following changes:

1. We use 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 [3], we suggest users build the image themselves rather than depending on this registry to remain available.
2. 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.

3. The original BugsInPy scripts install all Python packages using Pip package manager. Pip can invoke the compiler to build dependencies from source [4] or download prebuilt binary files. However, 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, 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 vanilla Ubuntu or Debian system without the necessary system libraries would result in installation failures. 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 Ubuntu or Debian and simply install those with Conda instead.
4. 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. This optimization helps reduce the time and resources required for environment setup, as the costly process of building environments from source can be bypassed. While it is tempting to use the same Conda environment for each project, there are multiple occasions where different bugs of the same project require different dependencies. For example, `ansible/bugs/{1,11,14}/requirements.txt` all vary subtly.
5. The BugsInPy framework correctly recognizes that installing the dependencies line by line `cat requirements.txt | filter | xargs -n 1 pip install`, rather than using `pip install -r requirements.txt`, bypasses certain restrictions imposed by pip. Specifically, when installing all dependencies at once, pip may ignore very old packages. However, installing the dependencies sequentially allows us to reproduce the bugs accurately. However, this results in failed installations for projects that include the `-e git+https://...` syntax in their `requirements.txt` file, since they would get passed along as `pip install -e` and `pip install git+https://...`. This revised code ensures that each line from the requirements.txt file is properly processed and passed as an argument to the `pip install` command.

By utilizing the modified approach, we were able to resolve the installation issues for specific repository dependencies. For instance, the installation of the `luigi` library from a specific GitHub repository now proceeds as expected, as evidenced by the successful cloning and checkout of the specified commit.

To address this issue, we have opened a pull request in the original repository [6]. This fix is crucial, as it impacts the reproducibility of bugs in projects such as black, cookiecutter, keras, luigi, pandas, sanic, and thefuck.

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}:
      bugsinpy-checkout
      envID = hash(requirements.txt + $python_version)
      if not conda env exists $envID:
        conda create env $envID
        conda install python=$python_version
        conda install $special_case_pkgs
        pip install requirements.txt
      conda activate $envID
      Report environment errors
      bugsinpy-test
      Report test results
      conda deactivate
  done
done
```

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. 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.

III. Results

TABLE I
REPRODUCTION OF BUGS IN BUGSINPY WITHOUT MODIFICATION

Project	Error	Both-pass	Both-fail	Expected	Total
PySnooper	1 (33%)	1 (33%)	0 (0%)	1 (33%)	3 (100%)
Continue...					
All		*continue...*			

RQ1. With the original BugsInPy framework, we achieved the following bug reproduction rates Table I. In that table, the results we can get are: - **Error**: Some part of the installation of the software environment needed to test the bug failed. - **Both-pass**: Both versions pass; we would expect the buggy version to fail. - **Both-fail**: Both versions fail; we would expect the fixed version to pass. - **Inverted**: The buggy version passes, and the fixed version fails. We do not observe this case. - **Expected**: The buggy version fails, and the fixed version passes. We consider *only* these bugs, "reproduced".

In that table, for each project, we show the raw count as well as percentage of outcomes for all bugs in that project. The last row shows the raw count as well as percentage of outcomes for all bugs in the dataset.

RQ2. We were able to rescue many of the broken test cases, as shown in Table I. In that table, we show the quantity in our rescued dataset and the delta from the unmodified quantity, denoted x. For example, "x-2=1" means the rescued dataset has 2 fewer errors than the unmodified, for a total of 1 error.

TABLE II
REPRODUCTION OF BUGS IN BUGSINPY WITHOUT MODIFICATION

Project	Error	Both-pass	Both-fail	Expected	Total
PySnooper	x-2=1 x-33%=33%	x+1=1 x+33%=33%	0 x+0%=0%	1 x+33%=33%	3 (100%)
Continue...					
All		*continue...*			

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.

Note that our bugs are *repeatable*, which means that we can run our code twice and get the same result [2].

By achieving a high success rate in reproducing bugs and passing tests, the BugsInPy dataset demonstrates 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, researchers have more bugs at their disposal for evaluating fuzzing, automatic program repair repair, and other research techniques.

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.

Table III presents the running time required to reproduce bugs in each project within the BugsInPy dataset, along with the respective containers. 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 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)
pandas	963
luigi	510
scrapy	268
keras	230
black	214
fastapi	197
thefuck	195
sanic	136
spacy	131
ansible	80
tqdm	36
youtube-dl	59
cookiecutter	40
httpie	39
matplotlib	26
tornado	14
pysnooper	5

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.

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. Often, packages required by the software environment are only installable in a specific version of Python.

TABLE IV
PYTHON VERSIONS IN BUGSINPY DATASET

Python	Bugs	
3.6.9	31	6%
3.7.0	58	12%
3.7.3	50	10%
3.7.4	33	7%
3.7.7	9	2%
3.8.1	33	7%
3.8.3	287	57%
Total	501	100%

Table V provides a summary of Python versions and their corresponding counts in the BugsInPy dataset. The percentages give us insights into the distribution of bugs across different Python versions in the dataset. The dataset certainly shows its age; Python 3.6 and 3.7 reached their official end-of-life and are not supported by CPython developers [7].

IV. DISCUSSION

A. What makes our reproduction easy?

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

1. **Automatation**: The `bugsinpy-testall` script provides an automated approach to reproducing and testing bugs in Python projects. The script streamlines the reproduction process, minimizes manual effort, and ensures we use a consistent procedure on each project. Note that these automation scripts must be carefully written and maintained. In particular, the original scripts did not have `set -e`, so some intermediate step might fail without alerting the user.
2. **Environment/Package manager**: Conda environment/package manager contributes to easy management of project dependencies. The key insight is that Conda can install packages to a local environment without interferring with global, system-wide packages. This makes it possible to define project-specific versions of libraries which would be normally managed by a platform-specific system-wide package manager.
3. **Lack of non-deterministic bugs**: None of the bugs in our dataset are race-conditions. Our scope is limited to constructing a reproducible software environment consistent with the original bug, and then the bug will show itself deterministically.

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. **Resource constraints during building**: Projects with a large number of bugs can pose challenges in terms of the time and resources required for bug reproduction. Sometimes, the software environment will involve a computationally-expensive step of building software from source. Reproducing and testing a significant number of bugs within limited resources may result in longer reproduction times and potential resource limitations.
2. **Lack of storage**: Our script ends up with quite a few Conda environments. These can be quite expensive to store, and we can't, for example, archive our environments in GitHub, due to space constraints.
3. **Missing packages in Conda**: Unfortunately, not all Pip packages and versions exist in our selected Conda repositories. Conda often only has
4. **Missing/yanked packages in PyPI**:
5. **Lack of expected output**

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.

1) Infrastructure changes: 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.

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 [8]. 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 [9]. 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] A. I. staff. "Artifact review and badging." (Aug. 24, 2020), [Online]. Available: https://www.acm.org/publications/policies/artifact-review-and-badging-current (visited on 01/19/2023).

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

[4] "Cmdoption-no-binary - 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).

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

[6] "Fixes -e option requires 1 argument. by faustinoaq · pull request #68 · soarsmu/BugsInPy," GitHub. (), [Online]. Available: https://github.com/soarsmu/BugsInPy/pull/68 (visited on 07/17/2023).

[7] C. developers. "Status of python versions," Python Developer's Guide. (), [Online]. Available: https://devguide.python.org/versions/ (visited on 07/17/2023).

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

[9] "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.