Unit Testing Practices in Jupyter Notebooks

Jesse Abdul, Hok Wai Chan, Jingyi He, Tevin Takata
Department of Information and Computer Science
University of Hawaii at Manoa
Honolulu, HI
jdcabdul@hawaii.edu
hokwai@hawaii.edu
jhe3@hawaii.edu
takatat9@hawaii.edu

***Abstract*—Computational notebooks, such as Jupyter Notebooks, are widely used in data science and scientific computing, but their adoption of software engineering best practices like unit testing remains understudied. This paper investigates the prevalence, structure, and impact of unit testing in Jupyter Notebooks through a large-scale empirical analysis of 966,231 notebooks. We identify only 1,448 (0.25%) notebooks with probable unit tests, revealing low adoption rates despite the benefits for reliability and maintainability. Notebooks with unit tests exhibit higher modularity, documentation, and code reuse, suggesting a correlation between testing and improved software quality. However, inconsistencies in test implementation and unused imports highlight gaps in testing practices. We discuss implications for tooling, education, and best practices to enhance testing adoption in notebook-based workflows. This study bridges the gap between interactive computing and software engineering, offering actionable insights for researchers and practitioners.**

I. Introduction

Jupyter notebooks are primarily used for data science, statistics, machine learning, or scientific research. The focus of the typical notebook practitioner is on exploring, analyzing, and visualizing data instead of developing production-level software. Notebook practitioners are not typically classically trained software engineers who are trained in software development methodologies [1]. Software engineers focus on developing software systems for reliability, maintainability, scalability, and quality. Testing is one of the fundamental software development principles that help to ensure the quality and maintainability of software.

A. Identifying Key Gaps in Current Approaches

Computational notebooks (e.g., Jupyter Notebooks or Google Colab) have become essential tools in data science and scientific computing. While they offer unique benefits for interactive computing and literate programming, unit testing practices in notebooks remain understudied. Software testing in general, and unit testing specifically, is not typically a primary focus for notebook developers. Referencing solutions identified or developed in other research could be very helpful in understanding this gap.

B. The Need for New Solutions and Perspectives

Analyzing a dataset of notebooks will help to identify the prevalence of this issue in real-world examples. By searching for keywords related to unit testing frameworks for different notebook languages (e.g., unittest for Python), we can determine the percentage of notebooks in the dataset that have some level of unit testing implemented. Further, manually inspecting notebooks with unit tests can help determine code coverage and other relevant metrics.

Additionally, notebooks that do and do not have unit testing implemented can be inspected using code quality tools to identify common issues. This analysis will allow us to assess how pervasive code quality issues are in notebooks with and without unit tests. Understanding these differences can highlight areas where improvements are needed and where unit testing could provide the most benefit.

C. Potential Impact on Software Quality and Maintenance

Implementing unit testing as part of the software development life cycle for notebooks can have a substantial impact on the quality and maintenance of code. Ensuring reliability through unit testing allows issues to be identified earlier in the development process, helping to confirm that functions behave as expected. When updates are made, unit tests can be run to verify that changes do not inadvertently break the code. Reproducibility, a fundamental principle of scientific research and data science, is also enhanced through unit testing, as it helps ensure that code produces consistent results based on given inputs.

Unit testing also plays a crucial role in debugging by isolating code problems, allowing practitioners to resolve issues efficiently. Without unit tests, errors can be harder to track down, leading to longer debugging times and potential inconsistencies in results. Furthermore, collaboration becomes significantly easier when unit tests are in place as they provide a clear framework for modifying and extending the code. Practitioners who did not originally develop the code can run and modify the unit tests to ensure that their contributions do not break the project.

By addressing these challenges, we can promote the integration of testing practices within computational notebooks, ultimately improving software quality and maintenance practices in data science and scientific computing.

II. Goal

The primary objective of this research is to investigate the prevalence and implementation of unit testing among practitioners utilizing computational notebooks. The study aims to assess how frequently practitioners incorporate unit tests into their notebook-driven workflows, identify the strategies employed for unit testing in notebooks – such as common unit testing frameworks – and explore the relationship between unit testing and the overall quality of code within notebooks. Additionally, the research will analyze the impact of unit testing on software quality and maintainability on notebooks.

Analyzing the prevalence and quality of unit testing implementation will help identify the scope of the problem that needs to be addressed to improve the quality and maintenance of notebooks. Once we understand the extent of the issue, we can examine the consequences of not implementing unit testing. This research can contribute to improving best practices in data science and scientific computing by reducing bugs and errors in notebook projects. Additionally, the findings may provide practical guidance for new developers, integrating the best practices identified into their workflow.

The immediate benefits of implementing unit testing in computational notebooks include increased code reliability, improved reproducibility, reduced debugging time, higher productivity, easier collaboration, and greater developer confidence in the code. In the long term, unit testing can enhance code maintainability, improve overall code quality, reduce technical debt, and facilitate a smoother transition from development to production. More broadly, adopting software engineering processes in notebook-based development can promote a quality-oriented approach to data science, benefiting the community as a whole.

III. Research Questions

RQ1: How frequently do practitioners include unit tests in Jupyter Notebooks?

Previous studies have primarily concentrated on reproducibility challenges within Jupyter Notebooks [2], but limited research has been conducted on quantifying unit testing adoption rates. This study seeks to fill this gap by providing empirical evidence of the prevalence of unit testing in Jupyter Notebooks.

By leveraging a large dataset, this research provides a quantitative perspective on how structured unit testing practices are being adopted within Jupyter Notebooks. This approach allows for a comprehensive analysis of testing trends and framework usage, offering new insights into testing behaviors among practitioners.

A clearer understanding of the prevalence of unit testing in notebooks will enable the development of recommendations aimed at increasing testing adoption. By identifying gaps and best practices, this research will contribute to improving software quality and overall maintainability in Jupyter Notebook-based development.

RQ2: Which unit testing frameworks are commonly used in Jupyter Notebooks?

While most studies focus on unit testing best practices for traditional software engineering, Jupyter Notebooks introduce unique challenges that require specialized approaches. This research highlights those challenges and examines how unit testing can be effectively integrated into notebook-based workflows.

This study offers a structured analysis of testing frameworks used in Jupyter Notebooks by identifying and categorizing real-world usage patterns in an established dataset. By doing so, it provides new insights into how different frameworks are utilized and their effectiveness in a notebook environment.

By understanding the most commonly used unit testing frameworks for Jupyter Notebooks, this research will contribute to the development of best practices that improve software quality, reproducibility, and maintainability in notebook-based development.

RQ3: How are unit tests structured within Jupyter Notebooks?

While traditional software engineering has extensive documentation on testing, there is limited knowledge about specific strategies for unit testing within Jupyter Notebooks. This research seeks to bridge this gap by analyzing how unit testing is integrated into notebook workflows.

This study categorizes real-world testing approaches within Jupyter Notebooks and identifies existing gaps in notebook testing methodologies. By examining different strategies, this research provides insights into how practitioners structure tests in computational notebooks.

By identifying unit testing strategies in Jupyter Notebooks, this research contributes to the development of best practices that improve the reliability and maintainability of computational notebook code. These findings will help guide practitioners in adopting more structured testing methodologies.

RQ4: How does the presence of unit tests affect code quality and maintenance in Jupyter Notebooks?

Previous studies have also explored code quality in Jupyter Notebooks [2]; however, the role of unit testing in enhancing notebook quality remains largely unexplored. This research seeks to address this gap by evaluating the impact of unit testing on various software quality attributes.

This study takes a novel approach by systematically evaluating whether unit testing leads to measurable improvements in software quality and maintainability in Jupyter Notebooks. By analyzing real-world usage patterns, the research will provide evidence of the benefits and limitations of unit testing in this context.

By demonstrating the benefits of unit testing in Jupyter Notebooks, this research contributes to the development of best practices that promote more structured, maintainable, and reproducible software development workflows. These findings will help guide practitioners in adopting more rigorous testing methodologies within the notebook environment.

A. Contribution

This study provides empirical evidence for unit testing prevalence in Jupyter Notebooks, filling a gap in prior research

by quantifying testing adoption. By analyzing commonly used unit testing frameworks, this research identifies the frameworks most compatible with Jupyter Notebooks, allowing developers to make informed decisions about their testing strategies. Additionally, it categorizes unit testing structures within notebooks, helping developers integrate testing into their workflows more efficiently and effectively.

This study also evaluates the impact of unit testing on Jupyter Notebook code quality, offering insights into how testing influences maintainability, readability, and reusability. By establishing best practices for unit testing in notebooks, the study aims to enhance software quality and maintenance, benefiting data science and research computing communities. Through these contributions, this research extends the current body of knowledge and serves as a foundational reference for developers, researchers, and educators striving to improve software engineering practices in computational notebooks.

This study introduces structured unit testing methodologies to Jupyter Notebooks, an environment that traditionally lacks formal testing practices. By incorporating established software engineering principles, this research aims to bridge the gap between interactive computing and structured software development.

Structured unit testing enhances code maintainability and reusability, ensuring that Jupyter Notebook code remains modular and easier to manage over time. Through empirical analysis, this study provides insights into the adoption of unit testing in notebooks, quantifying its prevalence and impact on software quality.

This research facilitates the adoption of rigorous testing practices among notebook practitioners by offering guidance on integrating unit testing into notebook workflows. The study also aims to improve existing testing frameworks by identifying gaps and proposing enhancements that make them more suitable for notebooks.

Additionally, this work lays the foundation for future research into automation, continuous integration, and testing solutions specifically tailored to computational notebooks. By addressing these areas, this research pushes the boundaries of software maintenance into an evolving domain, ensuring improved reliability and sustainability of notebook-driven development.

IV. Literature Review

De Santana et al. conducted a large-scale empirical investigation into the types of bugs and challenges faced by Jupyter Notebook users. By mining 14,740 commits from 105 GitHub repositories and analyzing 30,416 Stack Overflow posts, researchers identified common bug categories and their root causes, ultimately proposing a taxonomy for bugs in Jupyter projects. The findings highlighted unique challenges in notebook environments, such as issues arising from out-of-order cell execution. However, while comprehensive in identifying and categorizing bugs, the research did not specifically focus on the role of unit testing in preventing or mitigating these issues. The study primarily centered on bug occurrence and categorization rather than exploring preventive measures or best practices for testing within notebooks. The proposed research builds upon these findings by specifically investigating the prevalence and implementation of unit testing in Jupyter Notebooks. By focusing on unit testing practices, the study aims to identify strategies that enhance code quality and maintainability, addressing some of the challenges highlighted in the empirical analysis.

Wang et al. analyzed the root causes of crashes in machine learning notebooks on platforms like GitHub and Kaggle. Results showed that 87% of crashes were due to factors such as API misuse, data confusion, notebook-specific issues, environment problems, and implementation errors. Many failures stem from notebook-specific challenges, including out-of-order execution and environment inconsistencies. While effective in diagnosing crash causes, the research did not explore systematic approaches, such as unit testing, to prevent these issues. Furthermore, there was a lack of discussion on testing frameworks or methodologies that could address the identified root causes. The proposed study seeks to fill this gap by examining how unit testing frameworks and practices can mitigate the common issues identified in the analysis. By exploring the adoption of unit testing, the research aims to provide actionable insights to reduce notebook-specific problems and enhance overall code reliability.

Prabhu et al. revealed that many researchers develop their own software without formal training in software engineering. High-performance computing (HPC) was found to be underutilized due to a lack of expertise, and researchers often struggled with optimizing code efficiency, leading to suboptimal resource use. While providing valuable insights into computational science practices, the findings were limited by a focus on a single institution, which may not be representative of broader trends. Additionally, the emphasis was primarily on computational power and optimization, neglecting key aspects such as reproducibility, testing, and software sustainability. While highlighting gaps in software engineering practices, the study did not explore unit testing in computational notebooks. The proposed research addresses these missing aspects by assessing unit testing adoption, frameworks, and their impact on maintainability in Jupyter Notebooks. By focusing on structured testing methodologies, the study aims to improve reproducibility and software quality.

Fangohr et al. introduced nbval, a tool that automates the validation of Jupyter Notebooks by re-executing and comparing outputs. This approach enhances reproducibility by detecting changes in results due to software updates and supports Continuous Integration, allowing early detection of broken notebooks. Despite its advantages, nbval has some limitations, such as providing limited debugging feedback beyond reporting mismatched outputs. Its reliance on pytest may also pose a barrier for researchers who lack software testing experience, and the tool is primarily designed for Python notebooks, limiting its applicability to other Jupyter-supported languages. While nbval validates notebooks by comparing outputs, it does not enforce structured unit testing. The proposed study extends

TABLE I
LITERATURE REVIEW PAPERS

Study	Year	Topic	Key Findings
de Santana et al.	2022	Bug Taxonomy & Challenges	Investigated bug types and challenges in Jupyter Notebooks, analyzing 14,740 commits from 105 GitHub repositories and 30,416 Stack Overflow posts. Identified common bug categories and causes.
Wang et al.	2024	Failure Analysis in ML Notebooks	Found that 87% of crashes were caused by API misuse, data confusion, environment inconsistencies, and implementation errors.
Prabhu et al.	2011	Software Engineering Practices in Research	Found that many researchers develop software without formal software engineering training, leading to suboptimal resource utilization and lack of reproducibility.
Fangohr et al.	2020	Test Framework/Library	nbval automates notebook validation by re-executing and comparing outputs, supporting CI integration.
Santos et al.	n.d.	Developer Perspectives on Unit Testing	Surveyed 32 developers, finding that unit testing is valued but often done informally due to lack of training and time.
Pimentel et a.	2019	Reproducibility & Testing in Notebooks	Found that only 24.11% of notebooks executed without errors, and 4.03% produced the same results upon re-execution. Only 1.54% contained test-related modules.

this research by exploring systematic unit testing practices beyond simple output validation. By investigating best practices for integrating unit testing into computational notebooks, the research aims to provide guidance on improving software quality and long-term maintenance.

Santos et al. surveyed 32 developers from two Brazilian companies about their experiences with unit testing. Results indicated that developers generally value unit testing but face challenges such as a lack of training and time constraints. A moderate correlation was found between motivation and organizational support for unit testing, but in practice, testing was often performed in an ad-hoc manner rather than systematically. Despite offering valuable insights into unit testing challenges, the study's focus on only two companies limits its generalizability. Additionally, there was no specific analysis of unit testing practices in computational notebooks like Jupyter. The proposed research builds on these findings by focusing on unit testing in Jupyter Notebooks, addressing the gap in research on structured testing in notebook-based development. By analyzing real-world notebook usage, the study aims to propose structured solutions for improving unit testing practices in Jupyter environments.

Pimentel et al. examined the quality and reproducibility of Jupyter Notebooks on a large scale. Out of 863,878 attempted executions of valid notebooks, only 24.11% ran without errors, and only 4.03% produced the same results. Additionally, only 1.54% of the notebooks contained imported modules with keywords such as "test" or "mock," indicating a limited use of unit testing in these environments. While providing valuable quantitative data on the state of reproducibility in Jupyter Notebooks, the research did not explicitly discuss unit testing practices in detail. The proposed study expands on this work by further investigating the usage of unit testing frameworks and their impact on notebook reproducibility. By analyzing how unit testing can enhance the reliability of computational notebooks, the research seeks to provide concrete recommendations for improving testing practices in Jupyter-based workflows.

V. METHODOLOGY

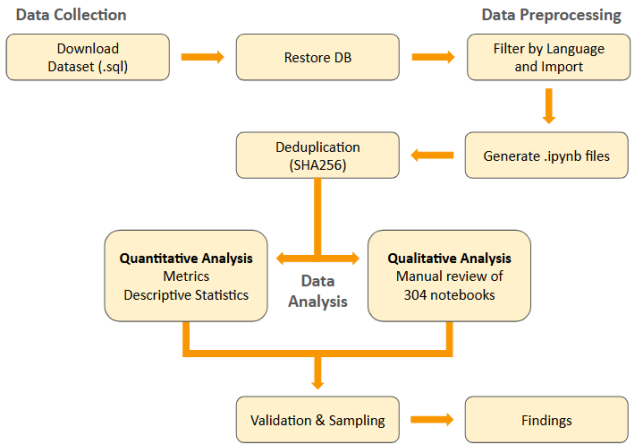

Fig. 1. Outline of the methodology used to analyze unit testing practices in Jupyter Notebooks, from dataset preparation to final findings.

A. Proposed Methodology

Figure 1 shows an overview of the proposed methodology. The diagram outlines the full pipeline, including data collection, preprocessing, SHA256-based deduplication, quantitative and qualitative analysis, and validation, culminating in the generation of research findings. The following sections will elaborate on the individual steps within each methodology stage.

B. Data Collection and Sources

The dataset used in this study was sourced from a PostgreSQL database backup file that contained a large collection of Jupyter notebooks. This dataset was originally compiled using the GitHub API and published as part of the paper titled "A Large-Scale Comparison of Python Code in Jupyter Notebooks and Scripts" [8]. The version used in this study includes software quality metrics for each notebook as well as for individual code and markdown cells. The data was collected from public GitHub repositories using the GitHub API between September and October of 2020.

To prepare the data for analysis, the PostgreSQL .sql backup file was downloaded from Zenodo and restored using a PostgreSQL database. The psql command-line utility was used to import the data.

C. Dataset Characteristics

The dataset used in this study contains 847,883 Python Jupyter notebooks and over 16 million associated code and markdown cells. Of these, 830,728 notebooks include detailed software quality metrics necessary for quantitative analysis.

A subset of 2,561 notebooks initially matched one or more variations of the four target unit testing libraries: unittest, pytest, doctest, or ipytest. After removing 1,113 duplicate notebooks based on matching file hashes, a total of 1,448 unique Python notebooks remained that imported one or more target test libraries and contained one or more recognizable unit test assertions. The filtering criteria ensured that only notebooks with confirmed testing logic (e.g., assert statements, doctest prompts, or method calls like assertEqual) were retained for inclusion.

A summary of this filtering process is shown in Figure 1 and detailed in Table 2 of the results section. This staged filtering helped to produce a high-confidence set of testing notebooks, used for RQ1–RQ4.

Additionally, 59 notebooks were found to import more than one of the target unit testing libraries, suggesting that hybrid testing strategies are present in some notebooks. The dataset also included anomalies such as aliased imports (e.g., import unittest as ut) and partial imports of individual test methods or submodules. These patterns are more difficult to detect using static string matching and may have led to underreporting in some cases. These cases reflect the diversity and complexity of how testing frameworks are used in practice.

All notebooks in the dataset included valid and complete metadata, such as notebook_id, notebook_language, cell_num, source, and notebook_hash. The metadata_full field was also present and well-formed across all included notebooks, supporting valid .ipynb file generation. Exported notebooks were verified against Jupyter schema requirements and preserved the correct cell execution order.

While the dataset is highly structured and internally consistent, it excludes notebooks that define test logic without importing a recognizable testing library. Additionally, the dataset reflects only publicly available notebooks as of Fall 2020 and excludes private repositories and those governed by restrictive licenses. Since test detection was performed through static analysis of import strings and source code, dynamically generated or runtime-imported frameworks may not have been captured. Malformed or unstructured import statements could also lead to false negatives, although these were not treated as missing data.

D. Data Processing

There were multiple layers of notebook filtering that were implemented with custom database queries. First, notebooks with a defined notebook_language of Python or a specific version of python were filtered. The presence of the target test libraries (unittest, pytest, doctest) was then detected using regular expressions applied to the notebook_imports field to ensure that no variations of these unit tests were included in the study.

A PHP script was developed to execute a custom query developed to extract the notebooks from the database that imported the target test libraries as well as all associated markdown and code cell source content. The PHP script saved each notebook in the git repository, and filenames were assigned based on notebook_id (e.g., 123456.ipynb) to maintain traceability. To enable unique identification of individual Python notebook files, the SHA256 hash for each notebook file was generated and saved in a custom table in the PostgreSQL database.

A custom database query was developed to identify duplicate notebooks based on the file hash values. Next, a custom query was executed to identify a unique notebook for each set of duplicate notebooks based on the corresponding file hash. This process retained only the first notebook in each duplicate group, which was then combined with the set of non-duplicate notebooks using the SQL logic to provide the full set of unique Python notebooks suitable for manual analysis.

A custom database query was developed to identify unique notebooks that imported more than one of the target unit testing frameworks (e.g. unittest and pytest), so these notebooks could be prioritized for manual review.

Variations of test library import strings were extracted using PostgreSQL regular expressions to detect patterns such as import unittest2, from pytest import ..., or aliased imports. The query result was exported with the number of notebooks that imported each variation of unittest, pytest, and doctest detected. These variations were manually researched to determine if they should be included in the study.

To count probable unit test assertions, a conservative matching strategy was used. Regular expressions were defined for each unit testing library and covered both current and deprecated assertion methods (e.g., assertEquals, assertRaisesRegexp) for unittest. Native assert statements were also counted when one or more test libraries were present.

E. Technical Implementation

The development environment for this study incorporated a variety of tools and programming languages to support data extraction, transformation, and analysis. The primary

programming languages used were SQL, PHP, and Linux shell scripting. These were chosen for their compatibility with the dataset format, ease of integration, and automation capabilities.

The core tools and their versions included PostgreSQL 17.2 (64-bit) as the database engine, pgAdmin 8.14 (64-bit) for database management and query execution, PHP 8.4.5 (64-bit) for scripting and file handling, Git Bash 2.47.1.windows.2 (64-bit) for executing shell scripts, and Git for Windows 2.47.1.2 (64-bit) for source control and interaction with GitHub, where the codebase was maintained. The system operated on Windows 11 (64-bit) Home Edition.

As for external dependencies, PHP was configured to enable the pdo_pgsql extension, allowing interaction with the PostgreSQL database. PostgreSQL was selected because the dataset was provided as a .sql backup file. PgAdmin was included for visual interaction with the database. PHP was chosen due to its ease of setup for handling database queries and generating .ipynb files, as well as computing SHA256 file hashes. Git and GitHub were used to version control and manage the project repository, and Git Bash allowed automation of script execution, including calling PHP from shell.

Configuration settings were managed using custom scripts. A shell script configuration file (config.sh) specified the path to the runtime php.ini file. Similarly, a PHP configuration file (config.active.php) stored the PostgreSQL connection credentials, which were read by the main PHP script responsible for generating notebook files. Full configuration instructions and setup steps were documented in the project's README.md file.

A number of custom tools and scripts were developed to streamline data processing. SQL scripts defined and saved indexes (define_notebook_indexes.sql) to speed up query execution. View definitions were saved in database_view.sql to simplify query syntax and improve reusability. A drop_all_views.sql script was also created to reset and rebuild views during development. The actual queries used to extract and process notebook data were documented in database_queries.sql. Custom PHP scripts handled the generation of .ipynb files from the database, while a shell script automated the execution of these PHP scripts.

The key algorithm implemented in PHP executed a query to retrieve unit testing notebooks along with associated code and markdown cells in sequential order, based on the cell_num field. The script initialized a $current_notebook_id variable to track the notebook being processed. As it iterated through each row of the result set, it checked for changes in the notebook ID. When a new notebook ID was detected, the script finalized the previous notebook by compiling its cells into a notebook object, saving it to disk as ¡notebook_id¿.ipynb, and computing a SHA256 file hash for deduplication. After the final row was processed, the last notebook was saved in the same manner. These hashes were stored in a separate database table for later use in identifying duplicates.

All code developed for this project was carefully documented. SQL view definitions were pushed to the GitHub repository along with explanatory commit messages. Column-level comments were included in the PostgreSQL data dictionary. Queries were annotated with in-line comments in database_queries.sql, and the README provided a comprehensive overview of each query's purpose. PHP files included both inline and header comments detailing function logic and algorithmic decisions. Likewise, the shell scripts were annotated, and their usage and configuration were explained in the README file.

To statistically compare notebook-level metric distributions between unit test and non-unit test notebooks, Mann–Whitney U tests were performed using R. The analysis was conducted directly on the PostgreSQL database using the DBI and RPostgres packages to establish a secure connection. Individual metric columns were queried in grouped form based on notebook classification and pulled into R as separate vectors for statistical comparison. Each test was performed using the wilcox.test() function with exact = FALSE and conf.int = TRUE parameters to handle the large sample sizes involved. P-values and test statistics were exported as structured CSV files for reporting and visualization. This approach allowed for efficient, reproducible, and scalable inferential testing without exporting the full dataset to external files. It also ensured that the statistical analysis remained tightly integrated with the filtering and preprocessing logic implemented in the PostgreSQL database.

F. Analysis Process

For each research question, both quantitative and qualitative data were collected in order to gain a comprehensive understanding of unit testing practices in Jupyter Notebooks. This mixed-method approach allowed us to address the prevalence and structural characteristics of unit tests, as well as their potential impact on code quality.

In the quantitative analysis phase, data from notebook metadata and code cell metrics were extracted to systematically evaluate unit testing practices. We identified the most important metrics that would allow us to thoroughly investigate our research questions. Some of these key metrics include cyclomatic complexity, comment count, and notebook imports. Descriptive statistics, such as frequencies, means, and standard deviations, were computed to analyze the distribution and prevalence of these metrics in the dataset. These statistical measures provided a foundational understanding of unit testing adoption patterns and their relationship to code quality indicators in Jupyter Notebooks. This included a Mann-Whitney U test (Wilcoxon rank-sum test) to evaluate whether median values differed significantly across notebook groups. P-values were reported for each metric (see Table 5), and a five-number summary was generated for metrics with valid data ranges (see Table 4). Some metrics, such as classes_comments and comments_per_class, were excluded due to invalid negative values. Others (e.g., coupling_between_functions) were excluded due to unreasonably high maximum values, possibly indicating data artifacts. These exclusions were justified to avoid skewing the analysis.

The qualitative analysis involved a manual review of a statistically significant sample of 313 Jupyter Notebooks, based on a 95% confidence level with a 5% margin of error. Each notebook was reviewed by two independent reviewers to ensure reliability and reduce individual bias. This manual inspection was performed to systematically document coding patterns and unit testing practices, including unit test structure (e.g., modular design), integration within the notebook (e.g., placement relative to production code), and code smells (e.g., duplicated assertions, overly complex test logic). This qualitative analysis allowed us to explore coding patterns and test practices that may not be captured through automated metrics, offering deeper insight into real-world testing behaviors within notebook environments.

To validate the data, we combined manual and automated techniques to verify its correctness. Notebooks were considered valid for analysis if they used a recognized testing library such as pytest, unittest, or doctest, contained at least one actual test function or assertion, and were not tutorial or classroom notebooks, which often lack adherence to real-world development practices. These criteria ensured that our dataset remained focused on relevant, production-oriented content.

VI. Results

RQ1: How frequently do practitioners include unit tests in Jupyter Notebooks?

To address this question, a multi-stage filtering pipeline was applied to an initial dataset of 966,231 Jupyter notebooks collected from public GitHub repositories. The first stage isolated notebooks that used Python as their primary language (n = 847,883), followed by a restriction to those with analyzable code metrics (n = 830,728), and finally deduplication based on content hashes to eliminate redundant notebook files (n = 582,623). These counts are summarized in Table 1.

From the deduplicated dataset, notebooks were filtered based on the presence of at least one import of a recognized Python unit testing framework (unittest, pytest, ipytest, or doctest), yielding 1,661 candidate notebooks. A final filtering step confirmed the presence of active test logic (e.g., assertion statements or doctest prompts), resulting in 1,448 unique Python notebooks containing probable unit tests. The full filtering pipeline is visualized in Figure 2.

This represents approximately 0.25% of all unique Python notebooks with available code metrics, indicating that while unit testing is present, it remains uncommon in practice within the Jupyter notebook ecosystem.

TABLE II
Notebook counts across filtering stage

Filtering Stage	# Notebooks
Jupyter Notebooks	966,231
Python Notebooks	847,883
Notebooks w/ Analyzable Metrics	830,728
Duduplicated Notebooks w/ Metrics	582,623
Imported Unit Test Library	1,661
Probable Unit Tests (For RQ1/RQ2	1,448

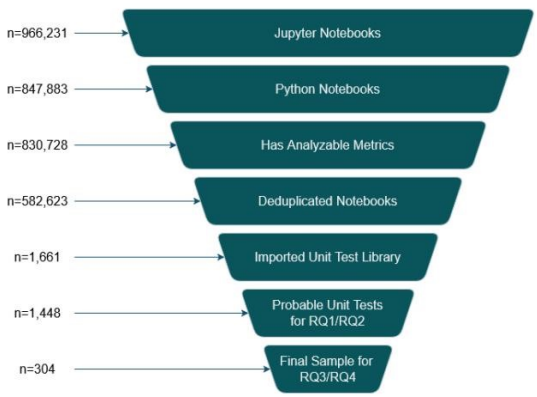

Fig. 2. The filtering pipeline used to identify notebooks with probable unit tests. The chart illustrates the narrowing of the initial dataset (n = 966,231) to a final set of 1,448 unique Python notebooks that imported a unit testing library and contained test logic.

RQ2: Which unit testing frameworks are commonly used in Jupyter Notebooks?

To answer RQ2, notebooks containing probable unit tests were analyzed based on which unit testing libraries were imported. Among the 1,448 unique Python notebooks that passed the filtering process, unittest was by far the most commonly used framework, appearing in approximately 80% of test-related notebooks (see Table 3).

Doctest was the second most prevalent, found in roughly 9% of testing notebooks, followed by pytest, which was found in about 6.5% of testing notebooks. The use of ipytest, a tool specifically designed for running tests inside Jupyter notebooks, was rare, appearing alone or in combination with other frameworks in only a handful of cases.

In addition to identifying the most commonly used unit testing frameworks, we also examined how often multiple libraries were used in combination within the same notebook. Most notebooks imported only a single framework, with combinations appearing infrequently. The most common combination was pytest with ipytest, found in 30 notebooks (2.1% of test notebooks). This pairing is logical, given that ipytest is designed to run pytest tests directly within Jupyter notebooks. The second most frequent combination was unittest and doctest, appearing in 18 notebooks (1.2%). Other combinations, such as unittest with pytest or ipytest, and notebooks importing three frameworks simultaneously, were rare. Notably, no notebooks in the dataset imported all four target libraries together. These results suggest that when developers use multiple testing frameworks in the same notebook, they tend to pair tools that serve complementary purposes or workflows, but most prefer to work within a single testing paradigm. When looking at the few notebooks that use a combination of testing frameworks, we can see that they are all tutorial notebooks educating on how to perform unit testing in Python.

These findings suggest that practitioners heavily favor built-in or low-setup testing frameworks like unittest and doctest

when working within the notebook environment. Modern frameworks like pytest, despite their popularity in general Python development, remain relatively underutilized in the Jupyter context, potentially due to their command-line orientation or additional installation requirements.

TABLE III
DISTRIBUTION ON UNIT TESTING FRAMEWORKS IN PYTHON NOTEBOOKS

Category	# Notebooks	% of Notebooks
unittest	1,161	80.1796
doctest	133	9.1851
pytest	95	6.5608
pytest, ipytest	30	2.0718
unittest, doctest	18	1.2431
ipytest	3	0.2072
unittest, pytest	2	0.1381
unittest, ipytest	2	0.1381
unittest, pytest, doctest	2	0.1381
pytest, doctest	1	0.0691
unittest, pytest, ipytest	1	0.0691
Notebooks w/ Probable Unit Tests	1448	100

RQ3: How are unit tests structured within Jupyter Notebooks?

A random sample of 304 notebooks was manually reviewed from the 1,448 identified as containing probable unit tests (see Figure 2). These notebooks were examined to understand test structure, usage context, and code organization. We found that unit tests are usually formatted in two different ways: inline testing immediately after functions, and grouped testing at the end. Inline testing occurs when a test is written in the code cell immediately following the implemented function. Grouped testing refers to consolidating all test code into one cell at the end of the notebook. Other observed structures include the use of external testing files or scripts, where a subset of notebooks offload testing to separate Python scripts or `.py` files that import functions from the notebook. Additionally, some notebooks demonstrate minimal or partial test coverage, containing only a small number of test cases or testing only specific blocks of code. A number of notebooks included to-do notes or incomplete test implementations, where a testing library (e.g., `unittest` or `pytest`) is imported but not actually used, often accompanied by comments such as "to be implemented. There were also instances of mixing testing styles, such as combining `unittest.TestCase` classes with standalone `assert` statements, or using pytest-like test functions without importing the `pytest` library. An example from notebook ID 366433 shows the use of the command `%%run_pytest --doctest-modules`, which runs `pytest` and includes doctests from defined modules. Inline assertions using basic `assert` statements were a common, lightweight testing approach. Some notebooks used custom testing approaches, defining ad hoc test functions or relying on print-based validation to simulate unit testing. Table 4 summarizes the observed unit testing formats in Jupyter Notebooks.

RQ4: How does the presence of unit tests affect code quality and maintenance in Jupyter Notebooks?

To assess whether unit tests are associated with measurable differences in code quality and maintainability, this study compared notebook-level metrics between two groups: 1,448 unique Python notebooks that contained probable unit tests and 581,175 notebooks that did not. Notebook metrics were extracted from a public dataset of GitHub notebooks [8] and analyzed using a combination of five-number summaries and Mann–Whitney U tests to compare the distributions of values across both groups.

After reviewing the full set of available metrics, a data validation step was performed to ensure only meaningful comparisons were included. Metrics with invalid values (e.g., negative comment counts in comments_per_class and classes_comments) or non-significant p-values (e.g., ccn and code_cells_count) were excluded. The remaining metrics, each of which showed a statistically significant difference ($p < .05$) in the Mann–Whitney U test, are included in Table 5 and Table 6. Figure 3 presents a visual comparison of the median values for these metrics between the two groups.

The results suggest that notebooks with unit tests exhibit several characteristics consistent with better structure, documentation, and modularity. For example, the median defined_functions_uses in unit test notebooks was 9, compared to 0 in notebooks without unit tests. This indicates that code in test-inclusive notebooks tends to reuse functions more frequently, which may support better modular design and reduce duplication. Similarly, the median halstead score, a measure of computational complexity, was nearly three times higher in unit test notebooks (4.67 vs. 1.57), suggesting that these notebooks tend to contain more sophisticated logic or functionality.

Documentation-related metrics also showed meaningful differences. Notebooks with unit tests had substantially higher median values for comments_count (17 vs. 2), extended_comments_count (28 vs. 9), and blank_lines_count (30 vs. 10), reflecting clearer annotation, greater spacing, and potentially improved readability. These trends reinforce the idea that developers who write unit tests may also follow better documentation practices.

In terms of structure, unit test notebooks included more classes (classes median = 1 vs. 0) and made greater use of both API-level functions and Python built-ins (API_functions_uses, build_in_functions_uses, and build_in_functions_count). These findings suggest that unit test notebooks often use more modular, reusable components and are less reliant on ad-hoc scripting.

Additional metrics like npavg (number of parameters per function) and sloc (source lines of code) were also higher in the test-inclusive group, consistent with larger or more complete implementations. Although some metrics such as coupling_between_methods showed extreme maximum values, the median differences still indicate meaningful structural variation.

In combination, these results indicate that the presence of unit tests is strongly associated with improved code quality, modularity, and documentation within Jupyter Notebooks.

TABLE IV
OBSERVED UNIT TESTING FORMATS IN JUPYTER NOTEBOOKS

Testing Format	Description
Inline Testing	Tests written in the code cell immediately following the implemented function being tested. One of the most popular ways to test a function within Jupyter Notebooks.
Grouped Testing	All test code consolidated into one cell at the end of the notebook. One of the most popular ways to test a function within Jupyter Notebooks.
External Testing	Testing offloaded to separate Python scripts or .py files that import functions from the notebook. Less popular and used when dealing with multiple languages.
Minimal/Partial Testing	Notebooks containing only a small number of test cases or testing only specific blocks of code. Used for quick, immediate checks of functions.
Unimplemented Testing	Testing library imported but not used, often with ”to be implemented” comments.
Mixed Testing Styles	Combination of different testing approaches (e.g., unittest.TestCase classes with standalone assert statements, or pytest-like functions without importing pytest).
Inline Assertions	Lightweight testing using basic assert statements directly in code cells.
Custom Testing Approaches	Ad hoc test functions or print-based validation to simulate unit testing.

While causality cannot be established from this analysis alone, the consistency of the differences across a wide range of metrics suggests that unit tests may serve as a proxy for broader software engineering best practices. Developers who write tests may also be more likely to structure, comment, and refactor their code more thoroughly. This conclusion is further supported by the robust statistical significance of the observed differences and the size of the dataset.

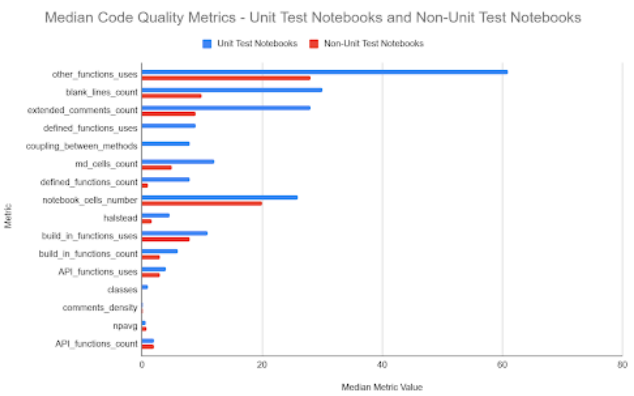

Fig. 3. Median Code Quality Metrics by Notebook Type. This horizontal bar chart shows the median values of selected notebook metrics, comparing notebooks with probable unit tests to those without. Metrics were selected based on statistical significance and data validity. Unit test notebooks consistently show higher medians across most quality dimensions.

VII. THREATS TO VALIDITY

This study has several limitations that may affect the interpretation and generalizability of its findings. One concern is the potential for manual review bias, as the interpretation of test structure and quality involves a degree of subjectivity. Although each notebook was reviewed by two researchers to reduce individual bias, variations in judgment could still influence the results. The study is also limited to Python notebooks, which excludes testing behaviors in other languages supported by Jupyter, such as R or Julia. Although Python is the most widely used language in this context, this constraint narrows the applicability of the findings to multi-language notebook environments.

Another limitation involves code duplication and repository noise. Despite efforts to filter out duplicates, forks, and clones of popular repositories may remain in the dataset, which can distort the overall distribution of testing patterns. In addition, the automated methods used to identify unit test logic did not always account for edge cases, such as code embedded in comments or cells containing a mixture of Python and other languages like SQL. These scenarios introduce potential misclassifications and reduce the precision of automated detection.

The dataset used in this study was collected in 2020, and as such, it may not reflect current trends in testing practices or the latest developments in testing tools. This temporal constraint limits the applicability of the findings to modern-day workflows. Another issue is the presence of external files in some repositories. Files such as standalone Python scripts or JavaScript files may contain testing logic that complements the notebook but falls outside the scope of this analysis. This exclusion creates a blind spot in fully understanding the testing landscape of a given project. Lastly, some testing behaviors are influenced by the execution environment. Tests may behave differently when run in a terminal compared to within a Jupyter Notebook, and tools such as ipytest may yield inconsistent results depending on the kernel state or execution order. These environment-specific factors were not examined in this study and represent another area of potential variability.

In this study, unit tests were primarily identified through the presence of Python assert statements and related unit test method calls using regular expression matching in PostgreSQL. While this approach facilitated scalable analysis across thousands of notebooks, it is limited by the inability of SQL-based regular expressions to interpret Python syntax accurately. Specifically, PostgreSQL cannot reliably distinguish between executable code and text contained within single-line comments (#) or multiline strings (”’ or ”””), both

TABLE V
STATISTICAL COMPARISON OF NOTEBOOK METRICS

Metric	Unit Test Notebooks					Non-Unit Test Notebooks				
	Min	Q1	Median	Q3	Max	Min	Q1	Median	3	Max
other_function_uses	0	18	61	72	2687	0	11	28	61	5593
blank_lines_count	0	17	30	47	2183	0	2	10	26	2827
extended_comments_count	0	11	28	65	750	0	1	9	27	4670
defined_function_uses	0	2	9	11	2059	0	0	0	5	909
coupling_between_methods	0	0	8	11	1341.25	0	0	0	0	6328
md_cell_count	0	3	12	13	156	0	0	5	13	1464
defined_functions_count	0	6	8	10	226	0	0	1	3	236
notebook_cells_number	1	13	26	30.25	569	1	10	20	37	2109
halstead	0	2	4.6667	7.1513	237	0	0.4423	1.5714	3.9474	3563.6
build_in_function_uses	0	6	11	17	733	0	2	8	20	2076
build_in_function_count	0	3	6	6	27	0	1	3	5	42
API_function_uses	0	2	4	9	190	0	0	3	9	4056
classes	0	1	1	2	29	0	0	0	0	148
comments_density	0	0.1163	0.1955	0.3110	0.7358	0	0.0253	0.1111	0.2239	6
npavg	0	0.5375	0.6639	0.8242	3.7368	0	0.5093	0.7344	0.9664	17.6
API_functions_count	0	2	2	5	46	0	0	2	6	194
mean_new_methods	0	0	0	1	25	0	0	0	0	56
mean_attributes_count	0	0	0	0.5	77	0	0	0	0	70
mean_override_methods	0	0	0	0	4	0	0	0	0	8

Note: Unit Test Notebooks (n=1,448), Non-Unit Test Notebooks (n=581,175). Only metrics with valid values and statistically significant differences ($p < .05$) were included.

TABLE VI
MANN–WHITNEY U TEST RESULTS FOR NOTEBOOK QUALITY METRICS

Metric	***p*-value**
classes	0
mean_new_methods	0
mean_attributes_count	0
coupling_between_functions	0
coupling_between_methods	0
defined_functions_count	0
defined_functions_uses	1.49×10^{-235}
blank_lines_count	7.85×10^{-202}
halstead	1.37×10^{-159}
extended_comments_count	1.87×10^{-143}
comments_count	3.10×10^{-137}
comments_density	1.32×10^{-94}
built_ing_functions_count	8.10×10^{-81}
sloc	2.26×10^{-68}
extended_comments_density	1.24×10^{-58}
mean_override_methods	6.89×10^{-48}
other_functions_uses	5.55×10^{-47}
md_cells_count	6.34×10^{-46}
API_functions_uses	7.52×10^{-20}
built_in_functions_uses	8.16×10^{-20}
API_functions_count	2.52×10^{-19}
notebook_cells_number	3.62×10^{-8}
npavg	2.48×10^{-7}
coupling_between_cells	4.78×10^{-6}

This table reports the Mann–Whitney U test statistics and p-values comparing unit test and non-unit test notebooks across the selected metrics. All included metrics are statistically significant at $p < .05$.

of which may include unit test keywords or method calls without executing them. As a result, some false positives (e.g., commented-out test code) and false negatives (e.g., multi-line assertions or uncommon formatting) are possible. A Python-native parser such as the ast or tokenize module would offer greater precision but was not feasible for full-scale, database-driven analysis. To mitigate these limitations, conservative filtering was applied using an exhaustive list of known assertion methods to reduce misclassification, and resulting metrics were interpreted with appropriate caution. Additionally, all notebooks flagged by PostgreSQL as containing no unit tests were manually verified by one researcher to ensure validity.

In the analysis for RQ4, several metrics were excluded due to data validity concerns or a lack of statistical significance. Metrics such as comments_per_class and classes_comments were removed because they included negative values, which are not meaningful for count-based fields and likely indicate data processing errors or placeholder values. Additionally, metrics with extreme or unverifiable maximum values, such as coupling_between_functions, were excluded to prevent distortion of the statistical summaries. Metrics that did not demonstrate statistically significant differences between groups in the Mann–Whitney U test ($p \geq .05$), including ccn, code_cells_count, and unused_imports_total, were also excluded from the five-number summary and visualization. These exclusions ensured that the analysis focused on robust and interpretable metrics but may have omitted potentially relevant dimensions of code quality due to data quality limitations.

VIII. Conclusion

This study reveals important insights into the current state of unit testing practices in Jupyter Notebooks. One recurring pattern identified is the prevalence of unused testing-related imports, such as unittest, pytest, and doctest, which are frequently present but not followed by any actual test code. These unused imports may reflect an intent to implement tests or reliance on predefined templates, but they ultimately contribute to a false sense of test coverage. In addition, testing is often informal and inconsistent, typically using inline assert statements or minimal test logic without the structure provided by standard frameworks.

When compared to prior research, our findings confirm the underutilization of unit testing in notebooks. In our dataset, only 0.25% of notebooks contained probable unit tests, which aligns with earlier studies reporting 1.54% of notebooks containing keywords such as "test" or "mock" [1]. This consistency across studies underscores a persistent gap in the adoption of robust testing practices within notebook-based development.

From a practical perspective, even basic testing within notebooks can have a meaningful impact on software maintenance. Inline assertions and lightweight validation techniques help catch errors early, improve code clarity, and reinforce intended behavior. These benefits are especially important in exploratory data analysis and collaborative environments. However, inconsistencies in testing practices, partial implementations, and vague intentions often reduce the long-term effectiveness and maintainability of notebook code.

To improve testing quality, we recommend that developers use standard testing frameworks such as unittest, pytest, and doctest directly within notebooks. Tools like ipytest can facilitate the execution of structured tests in an interactive context. Developers should avoid including unused imports or placeholder comments and should instead make test locations explicit, either by placing them immediately after function definitions or by grouping them in a clearly designated section at the end of the notebook.

Several best practices emerged from this study. Developers are encouraged to use inline assertions for immediate feedback, maintain a consistent location for test code within the notebook, and consider leveraging doctest in markdown cells or function docstrings to align documentation with validation. Tools such as ipytest can also be adopted to support a more interactive but structured testing workflow.

These insights offer practical guidance for improving testing practices in Jupyter Notebooks and emphasize the importance of applying sound software engineering principles in interactive and data-driven programming environments.

IX. Future Work

This study investigates unit testing practices within Jupyter Notebooks, analyzing a dataset of 966,231 notebooks from which 1,448 unique notebooks were identified as potentially containing unit tests. A manually reviewed sample of 304 notebooks was used to examine how testing is structured and which patterns are most common. Our findings show that while many notebooks import popular testing libraries such as unittest, pytest, and doctest, actual usage of these tools is often minimal or entirely absent. Instead, testing within notebooks is typically informal, dominated by simple inline assert statements and ad hoc validation logic. Structured tools like ipytest are rarely used. When tests are present, they tend to appear either directly after function definitions or grouped in a dedicated section at the end of the notebook. These patterns highlight a clear gap between the intent to implement testing and the actual practices followed within the notebook environment.

Future research can expand upon this work in several important directions. A longitudinal analysis could examine how unit testing practices change over time, especially in collaborative or actively maintained notebook-based projects. Since the dataset for this study was collected in 2020, a new round of data collection using the same deduplication and test detection techniques could be applied to modern Jupyter notebooks. This would allow for a meaningful comparison of code quality trends and help assess whether the adoption of testing practices has improved alongside the growing emphasis on reproducibility and maintainability in scientific workflows. In addition, interviews with notebook authors could offer valuable qualitative insights into the motivations, challenges, and decision-making processes related to testing in notebooks. Another potential direction involves dynamic execution analysis, in which notebooks are executed to evaluate actual test outcomes and runtime behavior. This would provide a deeper understanding of how effective and reliable these testing approaches are in real-world scenarios. Together, these directions offer opportunities to further improve the state of software engineering practices in interactive, data-driven programming environments.

References

[1] Adam Rule, Aurélien Tabard, and James D. Hollan. 2018. Exploration and Explanation in Computational Notebooks. In Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems (CHI '18). Association for Computing Machinery, New York, NY, USA, Paper 32, 1–12. https://doi-org.eres.library.manoa.hawaii.edu/10.1145/3173574.3173606.

[2] J. F. Pimentel, L. Murta, V. Braganholo and J. Freire, "A Large-Scale Study About Quality and Reproducibility of Jupyter Notebooks," 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR), Montreal, QC, Canada, 2019, pp. 507-517, doi: 10.1109/MSR.2019.00077.

[3] Wang, Y., Meijer, W., López, J. A. H., Nilsson, U., & Varró, D., "Why do machine learning notebooks crash?," arXiv.org, Nov. 25, 2024, available at: https://arxiv.org/abs/2411.16795, accessed: February 2025.

[4] de Santana, T. L., Neto, P. A. da M. S., de Almeida, E. S., & Ahmed, I., Bug Analysis in Jupyter Notebook Projects: An Empirical Study, arXiv.org, Oct. 13, 2022, available at: https://arxiv.org/abs/2210.06893, accessed: February 2025.

[5] Prakash Prabhu, Thomas B. Jablin, Arun Raman, Yun Zhang, Jialu Huang, Hanjun Kim, Nick P. Johnson, Feng Liu, Soumyadeep Ghosh, Stephen Beard, Taewook Oh, Matthew Zoufaly, David Walker, and David I. August, "A survey of the practice of computational science," State of the Practice Reports (SC '11), Association for Computing Machinery, New York, NY, USA, Article 19, 1–12, available at: https://doi.org/10.1145/2063348.2063374, accessed: February 2025.

[6] Fangohr, H., Fauske, V., Kluyver, T., Albert, M., Laslett, O., Cortés-Ortuño, D., Beg, M., & Ragan-Kelly, M., "Testing with jupyter notebooks: Notebook validation (nbval) plug-in for pytest," arXiv.org, Jan. 13, 2020, available at: https://arxiv.org/abs/2001.04808, accessed: February 2025.

[7] Santos, L., Santos, F., Parreira, R., & Mello, R. de, "Investigating the developer's perceptions of unit testing and its practice," Anais da Escola Regional de Engenharia de Software (ERES), available at: https://sol.sbc.org.br/index.php/eres/article/view/27044, accessed: February 2025.

[8] Grotov, K., Titov, S., Sotnikov, V., Golubev, Y., & Bryksin, T. (2022). Dataset of Jupyter Notebooks from the paper "A Large-Scale Comparison of Python Code in Jupyter Notebooks and Scripts" [Dataset]. Zenodo. https://doi.org/10.5281/ZENODO.6383114