Automated Modernization of Machine Learning Engineering Notebooks for Reproducibility

BIHUI JIN, University of Waterloo, Canada
KAIYUAN WANG, Google Inc., USA
PENGYU NIE, University of Waterloo, Canada

Interactive computational notebooks (e.g., Jupyter notebooks) are widely used in machine learning engineering (MLE) to program and share end-to-end pipelines, from data preparation to model training and evaluation. However, *environmental erosion*—the rapid evolution of hardware and software ecosystems for machine learning—has rendered many published MLE notebooks non-reproducible in contemporary environments, hindering code reuse and scientific progress. To quantify this gap, we study 12,106 notebooks selected from 75 popular Kaggle competitions: only 26% remain reproducible today. Crucially, we find that environment backporting, i.e., downgrading dependencies to match the submission time, does not improve reproducibility (decreased to 12%) but rather introduces additional failure modes.

To address environmental erosion, we design and implement MLEModernizer, an LLM-driven agentic framework that treats the contemporary environment as a fixed constraint and modernizes notebook code to restore reproducibility. MLEModernizer iteratively executes notebooks, collects execution feedback, and applies three types of targeted fixes: error-repair, runtime-reduction, and score-calibration. Evaluated on 8,210 notebooks that are non-reproducible under the baseline environment, MLEModernizer makes 3,292 (40.1%, GPT-5.2) and 3,683 (44.9%, GPT-OSS-120b) notebooks reproducible. MLEModernizer presents a best-effort automated recovery and modernization technique that can improve reproducibility for a subset of notebooks. Practitioners can leverage MLEModernizer to validate, reuse, and maintain MLE artifacts as the hardware and software ecosystems continue to evolve.

CCS Concepts: • **Software and its engineering** → **Software evolution**; **Software maintenance tools**; • **Computing methodologies** → *Machine learning*.

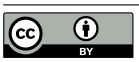
Additional Key Words and Phrases: Machine learning engineering, code modernization, Jupyter notebooks

ACM Reference Format:
Bihui Jin, Kaiyuan Wang, and Pengyu Nie. 2026. Automated Modernization of Machine Learning Engineering Notebooks for Reproducibility. *Proc. ACM Softw. Eng.* 3, ISSTA, Article ISSTA075 (October 2026), 22 pages. https://doi.org/10.1145/3832166

1 Introduction

Interactive computational notebooks, such as Jupyter notebooks [16], are widely adopted in machine learning and data science [17, 19, 24, 29], thanks to their combination of interactive code execution and rich visualizations [22]. In machine learning engineering (MLE), Python Jupyter notebooks (hereafter referred to as *notebooks*) serve as the primary code artifacts for documenting and sharing end-to-end machine learning pipelines, from data preparation to model training and evaluation [19, 24]. Many machine learning research prototypes are shared in this format [17, 19].

Authors' Contact Information: Bihui Jin, University of Waterloo, Waterloo, Canada, bihui.jin@uwaterloo.ca; Kaiyuan Wang, Google Inc., Mountain View, USA, kaiyuanw@google.com; Pengyu Nie, University of Waterloo, Waterloo, Canada, pynie@uwaterloo.ca.



ACM 2994-970X/2026/10-ARTISSTA075
https://doi.org/10.1145/3832166

Reproducibility is a crucial property in MLE to support trustworthy scientific progress in academia, and to facilitate code reuse in industry. Nonetheless, MLE notebooks increasingly suffer from the *environmental erosion* problem: as hardware and software ecosystems rapidly evolve, re-executing older notebooks in a current environment often fails or yields different results, and reconstructing the exact historical environment is rarely feasible [8, 19, 21]. This motivates a practical question for both research and reuse: can we reproduce the reported scores of previously published notebooks on the latest hardware and software stack?

To assess this, we conduct a motivation study on 12,106 notebooks selected from 75 popular Kaggle competitions [2]. We re-execute the notebooks using a recent Kaggle Docker container (aligned with how Kaggle executes submissions), classifying reproducibility via repeated executions and one-sample t-tests. Our study shows that only 26% of the notebooks are reproducible; the reproducibility gap persists despite a relatively standardized ecosystem [19, 20].

The standard industry response of environment backporting, i.e., downgrading dependencies to match the notebook's submission time, does not close this gap. Since Kaggle's Docker images older than two years are no longer available, we implement a backporting algorithm that downgrades dependencies to the submission timestamp. Surprisingly, only 12% of notebooks remain reproducible after backporting, substantially lower than the baseline reproducibility rate of 26%. This result exposes a fundamental flaw in the prevailing paradigm: treating the environment as a variable to be reconstructed and the notebook code as an immutable artifact. We therefore shift our focus to modernizing the notebook code to restore reproducibility, so that notebooks are modernized to run and reproduce under the current environment rather than the historical one. Repair in this setting cannot stop at "green" (no errors): an MLE notebook that runs but whose accuracy drops from 95% to 60% is not acceptable. We thus treat score-calibration—moving the reproduced score toward the reported Kaggle score within the reproducible band—as a first-class repair target alongside error-repair and runtime-reduction.

Recent advances in large language models (LLMs) have shown promising coding capabilities, especially for program repair [1, 27] and code evolution tasks, such as adapting deprecated APIs [6, 25]. In this work, we explore the use of LLMs to *modernize* MLE notebooks so that they can become reproducible in contemporary environments. We design and implement MLEModernizer, an LLM-driven agentic framework that iteratively fixes a notebook toward reproducibility. Each iteration begins with MLEModernizer executing the notebook and recording its score deviation (from the reported score), runtime, and any error messages during execution. By analyzing the execution context, MLEModernizer prepares a targeted LLM prompt for one of the three fix types: error-repair, score-calibration, or runtime-reduction. Then, MLEModernizer utilizes the LLM to generate a patch at the file level (i.e., the entire notebook), enabling it to identify the root cause of non-reproducibility and propose a comprehensive solution covering multiple code locations (thus reducing the number of fix iterations needed).

To evaluate MLEModernizer, we apply it to 8,210 non-reproducible notebooks in our dataset, using GPT-5.2 as the primary base LLM and GPT-OSS-120b as a comparison model. MLEModernizer successfully modernizes 40.1%–44.9% of notebooks, making 3,292 notebooks reproducible with GPT-5.2 and 3,683 reproducible with GPT-OSS-120b. Notably, 895 notebooks (10.9%) become reproducible despite residual errors (Error Reproducible); these errors do not prevent the notebooks from meeting the score-based criterion, but they may indicate unresolved issues in the accompanying visualization or analysis code. MLEModernizer is also economically viable: it requires 10.1 fix iterations on average and costs an average of $0.65 USD per notebook, making it practical for large-scale industrial use. We further validate the modernization outcomes by manually inspecting a statistically sampled subset of before-and-after notebook pairs to assess functional equivalence, complemented by a systematic analysis of error types and failure modes. We find that although

GPT-OSS-120b is slightly more robust in achieving reproducibility, GPT-5.2 produces a substantially higher proportion of functionally equivalent modernizations (71.5% vs. 52.7%).

Our work has broad implications for the MLE community. By enabling the automated modernization of MLE notebooks, MLEModernizer can facilitate the validation and reuse of MLE pipelines even as software and hardware stacks rapidly evolve. Because exact reproduction may be impossible due to the evolved environment, the best-effort modernization by our approach can serve as a starting point for manual refinement. Researchers can use MLEModernizer to reproduce legacy notebooks, reducing the tedious manual effort, thereby facilitating knowledge transfer in ML research. Engineers can use MLEModernizer to help update legacy MLE pipelines and enhance maintainability and reproducibility, thereby building user trust in their long-term usability.

The main contributions of this work include:

- **Motivation Study**. We conduct a reproducibility study of MLE notebooks mined from popular Kaggle competitions and show that only a minority remain reproducible today. We further demonstrate that environment backporting does not close the reproducibility gap.
- **Technique**. We develop MLEModernizer, an LLM-driven agentic framework as an initial attempt to recover and modernize MLE notebooks for reproducibility in contemporary environments via three targeted fix types: error-repair, runtime-reduction, and score-calibration.
- **Evaluation**. We evaluate MLEModernizer and find that it successfully modernizes 3,292 (40.1%) of the non-reproducible notebooks in our dataset, with a low cost of $0.65 USD per notebook, demonstrating practicality for large-scale use.
- **Dataset**. We curate a dataset of real-world Kaggle MLE notebooks with execution traces and reproducibility labels, enabling future research on MLE notebooks.

Our code and data are publicly available at https://github.com/Bihui-Jin/MLEModernizer.

2 Motivation

To demonstrate the reproducibility challenges in MLE notebooks, we show an example notebook submitted to the "Tabular Playground Series" Kaggle competition[1] in Figure 1. Kaggle is a prominent platform for MLE competitions and code sharing. The MLE pipeline in this notebook contains four steps (i.e., cells labeled as "In[x]" and "Out[x]"): (1) loading the training set; (2) configuring an XGBoost classifier model with hyper-parameters; (3) cross-validating the model on a subset of the training set, which is useful for hyper-parameter tuning; (4) training the model on the full training set, then applying the model on the test set, and finally generating a CSV file with predictions for submission. The left part of Figure 1 shows the successful run on Kaggle when the notebook was submitted, where the notebook obtained a score (prediction accuracy) of 0.87511 in the competition. However, as shown in the right part of Figure 1, our attempt to reproduce its results failed with errors when training the model. The errors are caused by a breaking change in XGBoost for target class inference: the "Cover_Type" labels in the original dataset range from 1 to 7, but newer versions of XGBoost require labels to be consecutive integers starting from 0.

In §2.1, we conduct a large-scale study to show that non-reproducibility (such as that caused by breaking changes in dependency libraries) is a pervasive problem in MLE notebooks. Then, in §2.2, we show that environment backporting, a naive solution that attempts to reproduce a notebook by downgrading the dependency libraries' versions to match the submission timestamp of the notebook, does not solve the reproducibility challenges.

[1]Link to the notebook: https://www.kaggle.com/code/sugamkhetrapal/tps-dec-2021-1-05-getting-started; Link to the competition: https://www.kaggle.com/competitions/tabular-playground-series-dec-2021/overview.

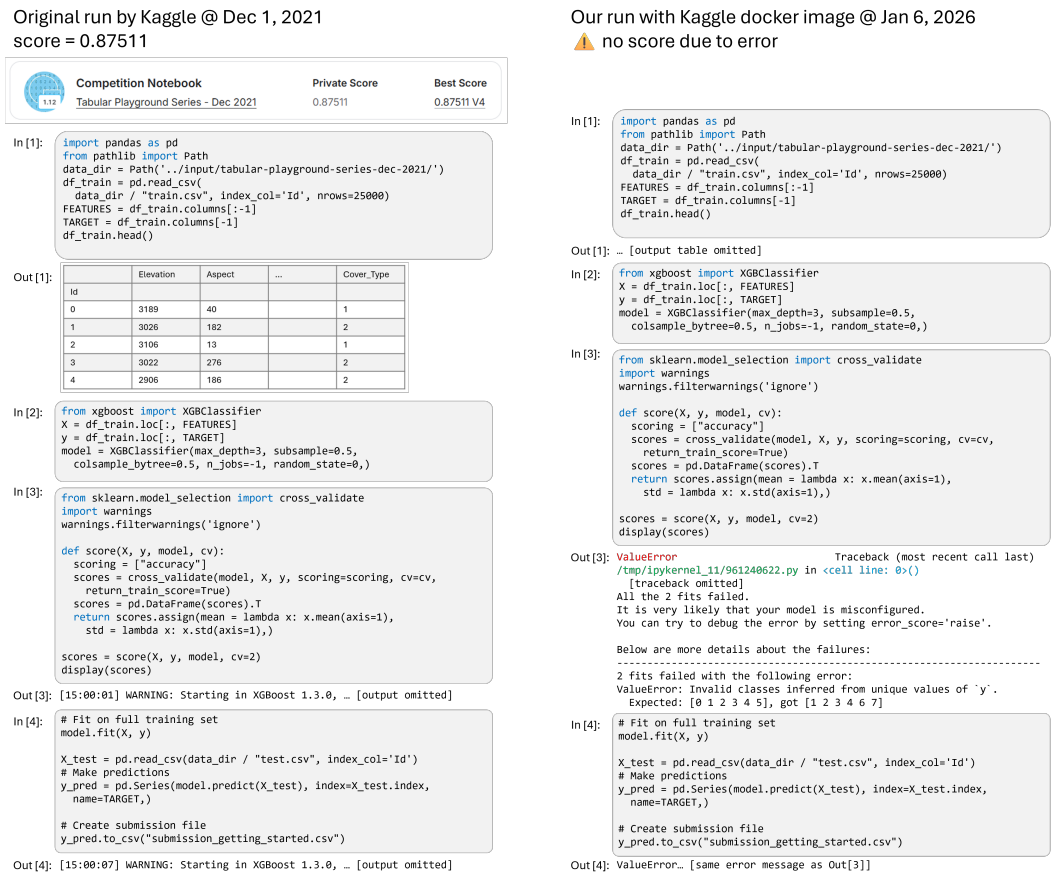


Fig. 1. An example notebook and our attempt to reproduce its results. Left: successful run on Kaggle; Right: our run failed due to a breaking change in XGBoost.

2.1 Reproducibility of MLE Notebooks

2.1.1 Dataset Collection. To systematically study the reproducibility challenges in MLE notebooks, we construct a large-scale dataset of real-world MLE notebooks from Kaggle. Our dataset collection process involves two main steps: (1) identifying relevant submissions from Kaggle competitions, and (2) extracting comprehensive metadata and code content for each submission version.

Competition Selection. We start from MLE-Bench [2], a curated benchmark built from Kaggle competitions and designed for evaluating end-to-end ML workflows. MLE-Bench provides a standardized offline grading setup, which makes it feasible to execute and grade notebooks at scale when Kaggle's held-out test sets are not publicly available. By anchoring our study on the competitions in MLE-Bench, we cover a broad range of high-quality MLE tasks including image classification, natural language processing, time series prediction, and tabular data analysis.

Notebook Mining. We leverage Kaggle's official data dump, Meta Kaggle [12] and Meta Kaggle Code [13], to identify all notebook submissions associated with the selected competitions, and we collect *all* available versions for each submission. Through this process, we mine 51,743 notebooks

from 75 Kaggle competitions. Our data cutoff is May 31, 2025; the oldest notebook in our corpus dates back to September 2016.

Filtering Criteria. We apply several filtering criteria to focus our study on notebooks where reproducibility can be meaningfully measured. First, we retain only notebooks that have been successfully executed on Kaggle and have a non-zero score, as these represent notebooks that successfully produced valid predictions and can be evaluated against ground truth. Second, we retain notebooks with a runtime of less than or equal to 600 seconds (decided by analyzing the distribution of runtime as shown in Figure 2), so that our study covers the majority of notebooks and remains tractable in terms of computational resource usage. Third, we restrict our analysis to Python notebooks (excluding R notebooks) to maintain consistency in dependency management and execution. Finally, we exclude notebooks that use external datasets beyond the competition's provided data. Such a filter eliminates external effects on reproducibility (e.g., datasets that may be inaccessible, or have changed over time) and allows us to focus on code modernization challenges. After applying these filters, we retain a total of 12,106 notebooks for our study.

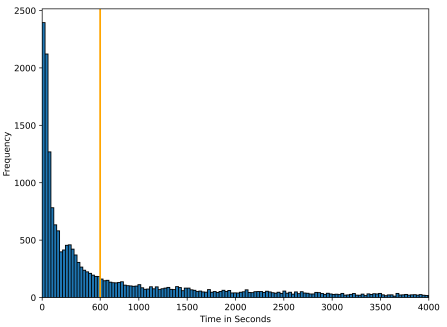


Fig. 2. Distribution of notebook runtime.

2.1.2 Execution Environment Setup. Kaggle notebooks are executed inside Docker containers [15], and each execution produces a versioned snapshot that is tied to a particular container version. To align with Kaggle's environment model and minimize environment-induced discrepancies, we execute each notebook inside a GPU-enabled container based on Kaggle's official Python image[2], mounting the competition training and test sets provided by MLE-Bench. The same execution environment is used for all experiments in this paper.

Environment and Hyper-Parameters. We conduct our experiments on a server with AMD EPYC 7343 CPU (16 cores @ 1.5GHz), NVIDIA RTX A6000 GPU (48G of RAM built with cuda_11.5.r11.5), running Ubuntu 22.04 LTS. To align with Kaggle's technical specifications [14], each run is provisioned with 4 CPU cores, 30 GB of memory, and one GPU.

Containerized Execution and Runtime Measurement. Notebooks are executed non-interactively via a notebook execution engine inside the container, with error-tolerant execution enabled so that exceptions do not immediately abort the run. Allowing errors ensures we can capture partial progress, logs, and all error locations; some errors (e.g., ones that arise during data visualization) may not affect the final execution outcome and may even be intentionally left unfixed. We impose a wall-clock cutoff of 600 seconds per notebook, consistent with our data filtering criteria in §2.1.1. Runtime is measured only during notebook execution, excluding the overhead of container initialization, filesystem mounting, and results grading.

Execution Outcome and Grading. Kaggle notebooks are expected to produce a competition "submission" file in CSV format. The required schema (e.g., column names) is specified by each competition. We treat successful CSV generation as the primary execution artifact and evaluate generated submissions using a grader adapted from MLE-Bench. Because Kaggle's full evaluation benchmark is not publicly released for many competitions, the MLE-Bench grader approximates Kaggle evaluation by splitting the original Kaggle training set into training and held-out test sets. For each

[2]https://github.com/Kaggle/docker-python

notebook, we report three metrics: (1) output status (whether the notebook has produced a valid CSV artifact), (2) reproduced score, and (3) runtime. Together, these metrics capture whether the notebook runs end-to-end and whether it produces a meaningful and competitive prediction file.

Reproduction aims to replicate the historical performance that each notebook originally achieved. Because MLE-Bench uses an offline grader (with train/test splits that may differ from Kaggle's private leaderboard evaluation), reproduced scores may not exactly match the target scores reported on Kaggle even when the pipeline is correct. We apply a two-sided one-sample t-test to assess whether repeated reproduced scores provide evidence of a mean difference from the target score, using up to ten executions per notebook. We operationally classify a notebook as reproducible when the test does not detect such a difference at $\alpha = 0.05$ ($p \geq 0.05$). When the reproduced scores are deterministic, making the t-test uninformative, we instead apply a heuristic threshold of $\tau = 10\%$ to classify the notebook as reproducible if the reproduced score differs from the target score by no more than this threshold.

Results. Table 1 shows baseline outcomes by execution status and reproducibility, with output status (whether the notebook has produced a valid CSV artifact) indicated for non-reproducible notebooks. Among the 12,106 notebooks in the study, 3,171 (26%) are classified as reproducible. Among error-free notebooks, 1,271 are Error-Free Reproducible, 636 are Error-Free Non-Reproducible, of which 56 do not generate a CSV result. The latter can occur because some submissions produce artifacts that do not match the grader's expected CSV format or naming conventions, or because the collected version does not contain the full submission-generation code. Among notebooks that raise errors, 1,900 are Error Reproducible, 8,002 are Error Non-Reproducible (where 6,536 have no CSV). An additional 297 notebooks are classified as failed (all ten execution attempts fail): 275 time out before completion and 22 correspond to notebook versions whose outputs were not written back to the notebook.

Table 1. Notebook reproducibility results in the baseline environment.

Error Status	Reproducibility	Count
Error-free	Reproducible	1,271
	Non-reproducible (w/ CSV)	580
	Non-reproducible (w/o CSV)	56
Error	Reproducible	1,900
	Non-reproducible (w/ CSV)	1,466
	Non-reproducible (w/o CSV)	6,536
Failed (timeout)		275
Failed (notebook not saved)		22

Summary: *Under the contemporary baseline environment, only 26% of the studied Kaggle notebooks reproduce; the majority are non-reproducible due to execution errors, missing CSV output, or score deviation, and a small fraction fail due to timeouts or missing saved outputs.*

2.2 Environment Backporting

Backporting aims to reconstruct the historical software environment that originally executed each notebook on Kaggle. Since Kaggle's Docker images older than two years are no longer available for download, we approximate historical environments by downgrading dependencies to versions consistent with each notebook's submission timestamp.

2.2.1 *Backporting Algorithm.*

We design a rule-based procedure for selecting historical versions of Python and third-party libraries based on submission metadata and code characteristics.

Dependency Analysis. To identify the dependencies used by each notebook, we apply an AST-based dependency extraction tool (pigar [4]) to the notebook code and generate dependency lists with correct PyPI package names. These dependency lists, together with the submission timestamp, form the inputs to our rule-based backporting procedure.

Inferring Python Version. First, we infer the Python major version (2 vs. 3) and minor version by cross-referencing the notebook's submission timestamp with the release dates of historical Python versions. The code is also analyzed to detect syntax patterns characteristic of Python 2 (for example, bare `print` statements, `xrange`, `raw_input`) or Python 3 (for example, `print()` calls, f-strings, `async`/`await`). We choose the latest `major.minor` version series that predates the submission but always use the latest patch release available within that series, because it minimizes dependency conflicts, and newer patch versions surpass older ones without breaking changes. For example, for a notebook submitted on 2021-10-01, we will first select Python 3.9 released on 2020-10-05 instead of Python 3.10 released on 2021-10-04, then select Python 3.9.25 that is released on 2025-10-31.

Selecting Dependency Versions. Given a notebook and its inferred Python version, we select historical versions of all third-party dependencies used by that notebook. For each imported package, we query the corresponding package index to retrieve the full release history, along with per-release metadata, required Python versions, and yanked-status flags. We filter out yanked releases and, for each remaining version, record both the release time and the declared Python compatibility. For a given notebook, we then choose the newest version released before the submission timestamp; if no such version exists, we fall back first to the oldest version that is compatible with the inferred Python version and, as a last resort, to the oldest available version. The resulting per-notebook dependency lists are written in a requirements-style format that constrains each package to be less than or equal to the selected version (e.g., `package_name<=v_hist`). Our rule-based selection strategy is designed to be robust, as it systematically respects submission dates, enforces Python-version compatibility, and promotes the use of shared minimal versions.

Environment Materialization. We materialize the inferred environments based on the baseline Docker container (§2.1.2), using `virtualenv` to install the inferred Python version and the selected dependency versions. If the backported environment cannot be installed (e.g., due to dependency conflicts or unavailable historical wheels), we mark the notebook as Failed.

2.2.2 *Reproducibility Analysis.*

Results. Table 2 shows backporting outcomes by execution status and reproducibility. Among error-free notebooks, only 972 are Error-Free Reproducible, and 879 are Error-Free Non-Reproducible, of which 62 do not generate a CSV. Among notebooks that raise errors, merely 515 are reproducible (Error Reproducible); 9,151 are Error Non-Reproducible (where 7,251 have no CSV). An additional 589 notebooks are classified as failed due to timeouts, unsaved notebook outputs, or failures to materialize a viable backporting environment (e.g., incompatible transitive dependencies, unavailable historical wheels, or exceeding the 30-minute setup limit).

Table 2. Notebook reproducibility results in the backported environment.

Error Status	Reproducibility	Count
Error-free	Reproducible	972
	Non-reproducible (w/ CSV)	817
	Non-reproducible (w/o CSV)	62
Error	Reproducible	515
	Non-reproducible (w/ CSV)	1,900
	Non-reproducible (w/o CSV)	7,251
Failed (timeout)		215
Failed (notebook not saved)		9
Failed (backporting)		365

Comparison with Baseline. Figure 3 illustrates the flow of notebooks from Baseline to Backporting: at Baseline, 1,271 are Error-Free Reproducible, 1,900 are Error Reproducible, 636 are Error-Free Non-Reproducible, 8,002 are Error Non-Reproducible, and 297 are failed. Backporting reduces the number of Error-Free Reproducible notebooks (1,271 to 972) and nearly doubles the failed set (297 to 589), while Error Non-Reproducible jumps (8,002 to 9,151); many notebooks that were reproducible

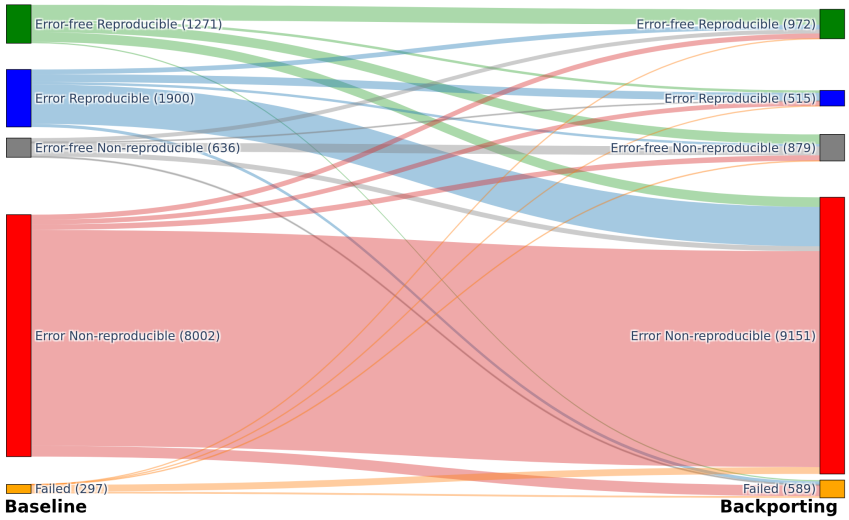


Fig. 3. Notebook reproducibility flow from Baseline to Backporting.

at Baseline become non-reproducible or fail under the backported environment, and a substantial flow from Error Non-Reproducible enters the failed category.

Backporting, therefore, does not improve reproducibility and introduces more failure modes. The results imply that simply downgrading dependencies to match submission timestamps is not a reliable remedy when migrating notebooks or reusing scripts from other sources in one's own environment: many notebooks remain non-reproducible or become outright failures due to environment reconstruction issues, so backporting is not recommended as a general-purpose fix.

Summary: *Under the backported environment, 1,487 notebooks (12%) are reproducible, fewer than the baseline (3,171, i.e., 26%). Environment backporting does not improve reproducibility and introduces additional failure modes; downgrading dependencies to match submission timestamps is not a reliable fix when migrating or reusing MLE notebooks.*

3 MLEModernizer Technique

Because environment backporting fails to address the reproducibility gap, and MLE execution environments evolve rapidly, we *modernize* notebook code while keeping the shared base container and hardware configuration fixed. We design MLEModernizer as an LLM-driven agentic framework that iteratively fixes a notebook toward reproducibility. LLMs have shown promising capabilities for program repair and code migration, but unlike prior work, the LLM fixes in MLEModernizer need to be performance-aware: MLEModernizer targets not only successful execution, but also reproducing the originally reported score. Thus, MLEModernizer grounds LLM fixes with rich execution feedback (runtime, error tracebacks, and score deviation) and uses specific prompts (runtime-reduction, error-repair, score-calibration) to guide each LLM fix. An overview of MLEModernizer's workflow is shown in Figure 4.

3.1 Agentic Workflow

MLEModernizer takes as inputs a notebook and a target scores_t to be reproduced. Then, MLEModernizer adopts an agentic workflow that iteratively collects execution feedback and applies LLM-driven fixes.

At each iteration, MLEModernizer performs the following steps:

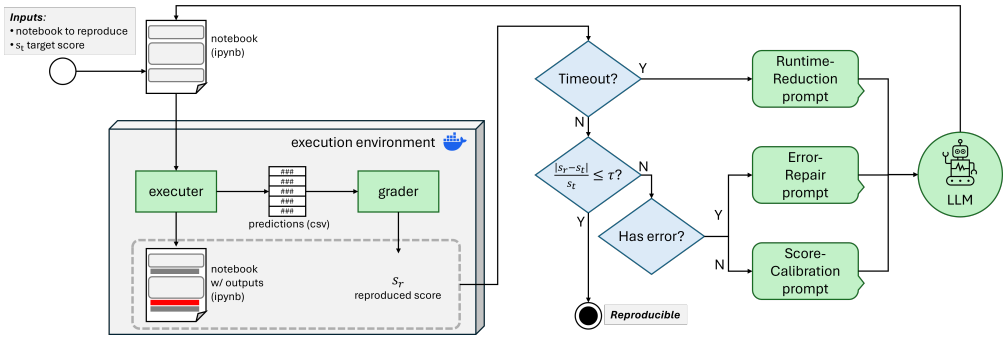


Fig. 4. MLEModernizer's workflow.

(1) executes the notebook and collects execution feedback, namely the outputs, error tracebacks (if any error occurs), reproduced score s_r, and runtime (detailed in §3.2);

(2) analyzes the execution feedback to detect one of three non-reproducibility issues:
- if the execution times out, which can be caused by API misuse or not enabling GPU acceleration, MLEModernizer first needs to reduce the runtime;
- if the reproduced score deviates from the target score, and the notebook has errors, MLEModernizer attempts to fix the errors as they are likely to be the root causes of non-reproducibility;
- if the score deviates, and the notebook does not have errors, a hidden API behavior change may be responsible, and MLEModernizer needs to calibrate the score to the target.

(3) triggers a targeted LLM fix based on the detected issue (detailed in §3.3).

MLEModernizer terminates when the notebook becomes reproducible (i.e., the score deviation is within the heuristically chosen replication threshold $\tau = 10\%$). We also set a maximum number of iterations $N = 16$; if the notebook does not become reproducible within the cap, it is assigned a terminal outcome (e.g., Error Non-Reproducible, Error-Free Non-Reproducible, or Failed due to timeout/LLM failure).

3.2 Execution Feedback Collection

Each notebook is executed inside the containerized environment as described in §2.1.2. We also mount the MLE-Bench competition dataset into the container so notebooks can access inputs and emit the submission CSV to standardized paths.

The base container is shared across all notebooks, while notebook-authored dependency setup remains part of the notebook code. The executor therefore applies any `!pipinstall` commands before executing the remaining cells; these commands can change packages for that notebook's run without changing the shared starting container. Then, the notebook is executed in an error-tolerant mode that continues past failing cells; the outputs and all error tracebacks are collected and saved to the notebook file. If the ML pipeline runs successfully, a submission CSV should be produced. The executor has a wall-clock time limit of 600 seconds. The grader, which is adapted from MLE-Bench [2], grades the submission CSV to obtain the reproduced scores s_r.

3.3 LLM Fixing

The input to the LLM is a structured prompt that includes a context-specific part and a shared part. The context-specific part depends on the detected non-reproducibility issue and instructs the LLM to follow certain requirements when performing the fix. There are three types of fixes:

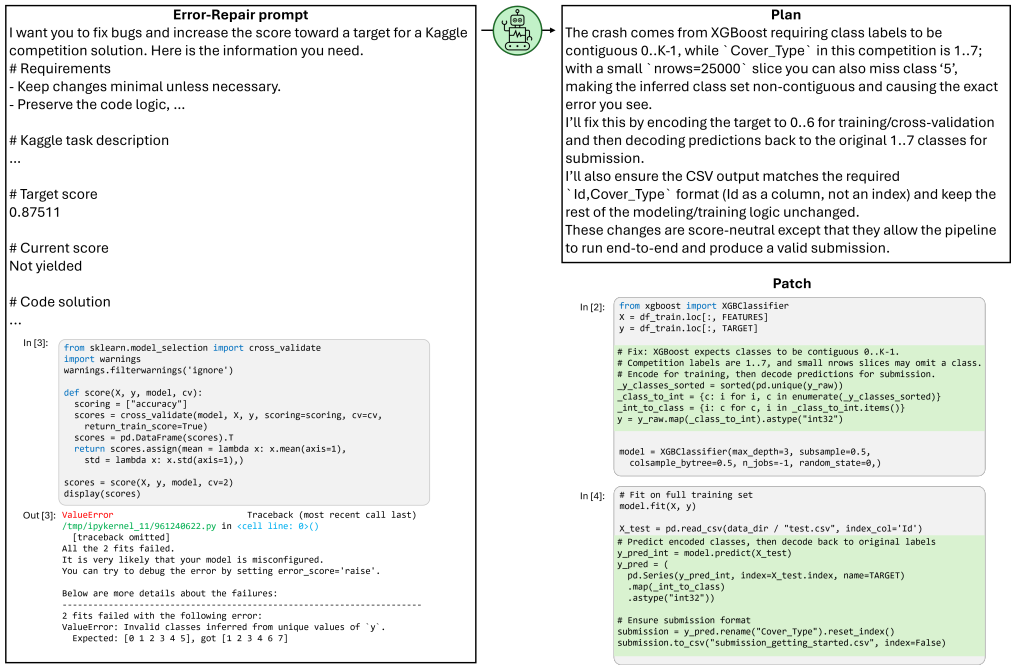


Fig. 5. MLEModernizer successfully modernizes the notebook in Figure 1 (s_r = 0.93778 compared to s_t = 0.87511), by re-encoding the dataset's classes (1..7) as the labels required by XGBoost (0..6).

Runtime-Reduction when the execution times out, Error-Repair when the notebook has errors, and Score-Calibration when the notebook does not have errors.

The shared part contains the following information:

- **Kaggle task description**: describing the competition's task, evaluation metric, and dataset schema.
- **Execution environment**: describing the Python version and dependency libraries' versions.
- **I/O contract**: specifying the paths for input data (`/kaggle/input/`) and the expected submission CSV location (`/kaggle/working/`).
- **Scores**: target s_t, current s_r, and the metric directionality (higher/lower is better).
- **Notebook**: the current notebook rendered in a cell-delimited format; when there are errors, the traceback is appended to the corresponding cell.

The LLM is also instructed to return a response in a fixed plan → patch format: (1) a short plan that enumerates the intended changes and their expected effect on the observed failure mode, followed by (2) a single markdown code block containing the full updated notebook in the same cell-delimited representation. MLEModernizer parses this output deterministically and applies the patch to the notebook for the next iteration. This protocol both enforces a concrete, executable patch and preserves sufficient structure for automated application and auditing.

3.4 Running Example

In the motivating example in Figure 1, the execution fails under a newer XGBoost version because the classifier expects labels in $\{0, \ldots, K-1\}$ but the dataset provides labels in $\{1, \ldots, 7\}$. MLEModernizer detects this failure from the traceback, issues an Error-Repair prompt, and applies a patch

that inserts an explicit label re-encoding step (e.g., shifting/encoding labels to start at 0) while preserving the original training/evaluation logic and submission format. As shown in Figure 5, the patched notebook executes successfully and achieves a score of 0.93778, slightly above the target score of 0.87511.

4 Evaluation

Subjects. MLEModernizer focuses on non-reproducible notebooks: these notebooks either fail to produce a valid submission artifact or fail the statistical reproducibility criterion (one-sample t-test, §2.1.2). We exclude reproducible notebooks because they already run and reproduce under the baseline environment and offer little headroom for improvement. We also exclude notebooks that hard-fail due to timeouts or pathological resource usage, as such cases are dominated by system-level constraints rather than code-level modernization.

From the 8,638 non-reproducible notebooks identified in the baseline (§2.1), we further filter candidates based on prompt length. For each notebook, MLEModernizer constructs prompts that include the source code (organized into cells) and, when applicable, error tracebacks from previous executions. To avoid exceeding the context window of the LLM and to prevent repeated failures due to overlong prompts, we empirically select a token cutoff based on the joint distribution of tokenized source and error text. Specifically, we tokenize all notebooks and associated error messages using an OpenAI model-compatible tokenizer [10], compute the 95th percentile of the resulting token counts, and obtain a cutoff of 16,337 tokens. Notebooks whose combined source and error text exceed this cutoff are excluded from the pool. The final subject set consists of 8,210 notebooks (out of 8,638 non-reproducible notebooks); we remove 428 extremely large notebooks to keep LLM calls within context limits and avoid disproportionate allocation of the compute budget.

Experimental Setup. We use the same software and hardware setup as the motivation study (§2.1.2). We use OpenAI's GPT-5.2 (the 2025-12-11 version) as the LLM in MLEModernizer and the open-weight GPT-OSS-120b for comparison.

Manual Inspection. We statistically sample a subset (368 {before, after} notebook pairs modernized by MLEModernizer, ensuring a 95% confidence level and 5% margin of error) and manually verify crash types and crash phases within the ML pipeline, as outlined in Wang et al. [26]'s work. In addition, functional equivalence of the notebook pairs before and after upgrade is also inspected. Specifically, two annotators (the first author and a Master's student in CS) independently label the notebook pairs, following a closed coding procedure [23]. Then, the two annotators discuss the labels for each notebook pair on which they disagree and establish consensus. If any conflicts persist, a third annotator (who is a computer science Professor in MLE and also an author of this paper) arbitrates remaining conflicts. Inter-rater reliability (IRR) is measured by Cohen's kappa [3].

Research Questions. We study the following research questions:
RQ1. How many notebooks become reproducible after applying MLEModernizer?
RQ2. How many and what kinds of fixes does the LLM perform?
RQ3. How much code modification is needed to make the notebooks reproducible?
RQ4. How much does it cost to apply MLEModernizer?
RQ5. Which types of errors can and cannot be fixed by MLEModernizer?

4.1 RQ1: MLEModernizer Reproducibility

Table 3 reports file-level modernization outcomes for the proprietary GPT-5.2 and open-weight GPT-OSS-120b models, categorized by execution status and reproducibility. With MLEModernizer supported by GPT-5.2, 3,292 of the 8,210 upgraded notebooks (40.1%) become reproducible: 2,397

Table 3. Reproducibility after applying MLEModernizer to the 8,210 non-reproducible notebooks.

(a) Upgrade with GPT-5.2.

Error Status	Reproducibility	Count
Error-free	Reproducible	2,397
	Non-reproducible (w/ CSV)	2,578
	Non-reproducible (w/o CSV)	1
Error	Reproducible	895
	Non-reproducible (w/ CSV)	633
	Non-reproducible (w/o CSV)	243
Failed (timeout)		1,023
Failed (notebook not saved)		424
Failed (LLM failure)		16

(b) Upgrade with GPT-OSS-120b.

Error Status	Reproducibility	Count
Error-free	Reproducible	2,877
	Non-reproducible (w/ CSV)	2,348
	Non-reproducible (w/o CSV)	22
Error	Reproducible	806
	Non-reproducible (w/ CSV)	303
	Non-reproducible (w/o CSV)	371
Failed (timeout)		1,060
Failed (notebook not saved)		416
Failed (LLM failure)		7

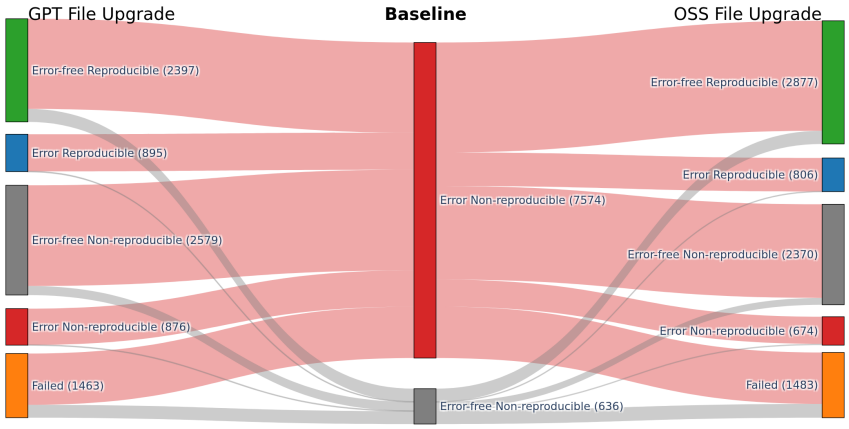


Fig. 6. Notebook reproducibility transitions from Baseline to MLEModernizer modernized notebook.

end-to-end runs complete without errors and pass the one-sample t-test (Error-Free Reproducible), and 895 produce a reproducible outcome despite residual execution errors (Error Reproducible).

Among 2,579 Error-Free Non-Reproducible notebooks, 2,578 are w/ CSV, and 1 is w/o CSV generated. Among notebooks that raise errors, 876 are Error Non-Reproducible (where 243 have no CSV). An additional 1,463 notebooks are classified as failed. Specifically, Figure 6 (left) illustrates the flow from Baseline to file-level upgrade: at Baseline, 636 Error-Free Non-Reproducible and 7,574 Error Non-Reproducible notebooks are fed into the modernization pool (8,210 in total); after upgrading, 2,397 become Error-Free Reproducible, 895 become Error Reproducible, 2,579 remain Error-Free Non-Reproducible, 876 remain Error Non-Reproducible, and 1,463 fail.

Using the open-weight GPT-OSS-120b, 3,683 of the 8,210 upgraded notebooks (44.9%) become reproducible (Table 3b, Figure 6 (right)): 2,877 end-to-end runs complete without errors and pass the one-sample t-test (Error-Free Reproducible), and 806 produce a reproducible outcome despite residual execution errors (Error Reproducible). Among 2,370 Error-Free Non-Reproducible notebooks, 2,348 have CSV generated, and 22 are w/o CSV generated; among notebooks that raise errors, 674 are Error Non-Reproducible (371 w/o CSV). An additional 1,483 notebooks fail (1,060 timeouts, 416 unsaved outputs, and 7 LLM failures). Figure 6 (right) shows a qualitatively similar transition pattern from Baseline to file-level upgrade: after modernization, 2,877 become Error-Free Reproducible, 806 become Error Reproducible, 2,370 remain Error-Free Non-Reproducible, 674 remain Error Non-Reproducible, and 1,483 fail.

The upgraded cohort is drawn from Non-reproducible notebooks in Baseline (§2.1); in that cohort, zero notebooks are reproducible at Baseline. MLEModernizer raises the share of reproducible notebooks from 0% to 40.1% (3,292 of 8,210) with GPT-5.2, including 2,397 notebooks that are also error-free and 895 that remain reproducible despite residual errors. With GPT-OSS-120b, MLEModernizer achieves a higher reproducibility rate of 44.9%, reproducing 3,683 notebooks, including 2,877 notebooks that are error-free and 806 that remain Error Reproducible.

Beyond statistical reproducibility, we also assess functional equivalence between each baseline notebook and its modernized version (manual inspection protocol above). Among pairs upgraded by GPT-5.2, 71.5% are judged functionally equivalent, with an IRR of 0.70 (substantial consensus). Among the non-equivalent pairs, 47.6% of mismatches are attributed to score-calibration, which is allowed to perform small parameter/model changes to reach the target score under the new environment. With GPT-OSS-120b, only 52.7% of pairs are functionally equivalent (IRR = 0.78, substantial consensus); among its non-equivalent pairs, 46.9% of mismatches are due to score-calibration.

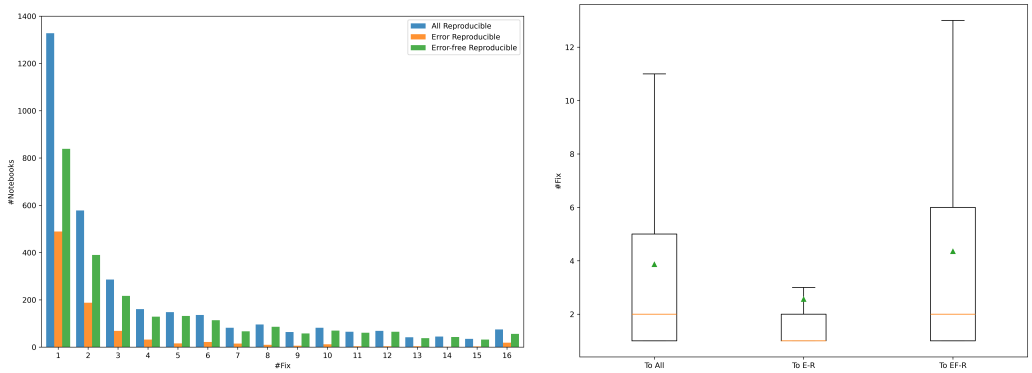
GPT-OSS-120b achieves a reproducibility rate 4.8 percentage points higher than GPT-5.2 (44.9% vs. 40.1%), but upon inspection, GPT-5.2 yields substantially more functionally equivalent modernizations (71.5% vs. 52.7%). This shows a dilemma that achieving score-based reproducibility and preserving semantic parity are not always compatible. We therefore focus the RQ2 and RQ3 analyses on GPT-5.2 to examine fix statistics and code-edit patterns under the backend that better preserves functional equivalence, which is more important in our goal.

Summary: *MLEModernizer restores statistical reproducibility for 40.1% of notebooks with GPT-5.2 and 44.9% with GPT-OSS-120b; the GPT-OSS-120b rate is 4.8 percentage points higher. Manual inspection finds higher functional consistency with GPT-5.2 (71.5%, IRR = 0.70) than GPT-OSS-120b (52.7%, IRR = 0.78); most non-equivalent mismatches stem from score-calibration. Overall, MLEModernizer restores reproducibility for a large fraction of previously non-reproducible MLE notebooks.*

4.2 RQ2: Statistics of LLM Fixes

Figure 7a shows how many notebooks become reproducible at each fix count. The distribution is heavily skewed toward few fixes, with the majority of notebooks becoming reproducible after 1–3 fixes; at 1 fix, which can be seen as a zero-shot LLM repair baseline, the modernization success rate is 16.2%, comparing to 40.1% success rate for MLEModernizer achieved with up to 16 iterative repair rounds. The absolute count of Error-Free Reproducible notebooks peaks at 2 fixes, while Error-Free Reproducible notebooks become a larger *share* of reproducible outcomes from 3–4 fixes onward (and dominate at higher fix counts despite smaller totals). Eliminating all errors thus often requires additional fixes beyond the first reproducibility milestone.

Figure 7b compares the distribution of fix counts by outcome: "To All" (all reproducible notebooks), "To E-R" (Error Reproducible), and "To EF-R" (Error-Free Reproducible). In file-level upgrade over 8,210 notebooks, 3,292 notebooks become reproducible; the mean number of fixes per notebook across all upgraded notebooks is 10.1, and among reproducible notebooks the mean is 3.87. Among reproducible notebooks, the mean number of fixes that achieve reproducibility but leave error cells (Error Reproducible) is 2.56, and the mean number that achieve reproducibility without error cells (Error-Free Reproducible) is 4.35. "To EF-R" has a higher median and mean (and wider spread up to 16 fixes), indicating that achieving error-free reproducibility typically requires more fixes than achieving reproducibility with residual errors. Most notebooks are made reproducible with few fixes: at least half require no more than 2 fixes, while the higher mean of 3.87 reflects the right-skewed tail; achieving Error-Free Reproducible often requires more fixes than Error Reproducible.



(a) Number of notebooks that become reproducible at each fix count.

(b) Box plots of number of LLM fixes. "To All" represents all reproducible notebooks (E-R and EF-R) after modernization.

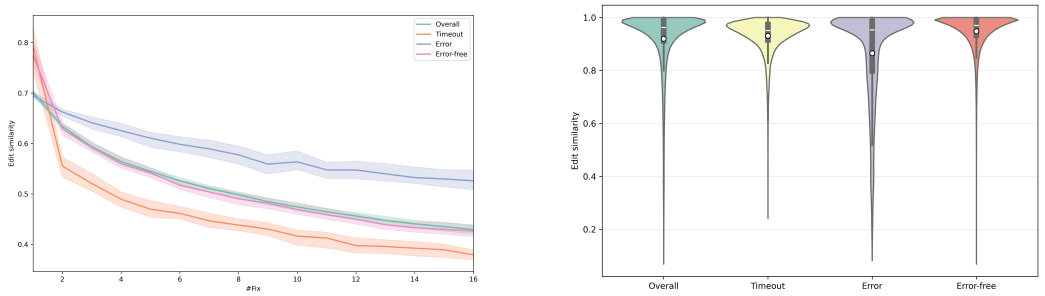
Fig. 7. Distribution of the number of LLM (GPT-5.2) fixes performed by MLEModernizer.

To characterize the kind of fixes the LLM applies and the resulting outcomes, Table 4 reports state-change transitions by fix type. Rows identify the fix types selected for the next iteration: the LLM chooses Runtime-Reduction when the script exceeds the runtime limit, Error-Repair when the script has execution errors, and Score-Calibration when the script has no errors and does not need runtime reduction, but its output must be aligned toward the reported Kaggle score. Columns correspond to the status of the notebook after the fix is applied and the notebook is run: Timeout (execution exceeds the time limit), Error Non-reproducible (Error Non-Reproducible), Error-free Non-reproducible (Error-Free Non-Reproducible), Error Reproducible (Error Reproducible), and Error-free Reproducible (Error-Free Reproducible). Score-Calibration is the most frequent fix type (39,226 applications), and the vast majority of those applications leave the notebook Error-Free Non-Reproducible (35,339); 1,261 reach Error-Free Reproducible. Error-Repair is the next most frequent (27,765); 1,059 reach Error-Free Reproducible and 834 Error Reproducible, but 17,046 remain Error Non-Reproducible and 5,176 become Error-Free Non-Reproducible, so error repair often improves state but does not always achieve reproducibility in one step. Runtime-Reduction is applied 15,718 times; 11,838 applications still result in a timeout, and only 127 reach a reproducible state (41 Error Reproducible, 86 Error-Free Reproducible), so reducing runtime is the hardest type of fix to resolve. Overall, the table shows that Score-Calibration and Error-Repair account for most fixes. Error-Repair produces the largest number of error-containing reproducible outcomes, whereas Score-Calibration contributes the most Error-Free Reproducible outcomes. By contrast, Runtime-Reduction rarely leads directly to reproducibility.

Summary: *In file-level upgrade (GPT-5.2), MLEModernizer makes 3,292 of 8,210 notebooks (40.1%) reproducible; a single-fix zero-shot repair succeeds on only 16.2% of the cohort, versus 40.1% with up to 16 iterative fixes. Among reproducible notebooks, the mean is 3.87 LLM fixes (median 2); achieving Error-Free Reproducible requires more fixes on average (4.35) than Error Reproducible (2.56). By fix type, Score-Calibration and Error-Repair dominate; Score-Calibration contributes the most Error-Free Reproducible outcomes (1,261), while Runtime-Reduction rarely yields immediate reproducibility (127 of 15,718).*

Table 4. Number of LLM fixes by type (rows) and resulting notebook reproducibility outcomes (columns).

Fix Type	#Total fixes	Timeout	Error Non-reproducible	Error-free Non-reproducible	Error Reproducible	Error-free Reproducible
Runtime-Reduction	15,718	11,838	2,797	956	41	86
Error-Repair	27,765	3,650	17,046	5,176	834	1,059
Score-Calibration	39,226	1,136	1,475	35,339	15	1,261



(a) Edit similarity to the original notebook as #fix increases.

(b) Distribution of the edit similarity between the notebook before and after each LLM fix.

Fig. 8. Cumulative (left) and per-fix (right) code modification scale by MLEModernizer (GPT-5.2).

4.3 RQ3: Code Modification Scale

Making notebooks reproducible often requires substantial cumulative code modification. Figure 8a displays how much the notebook changes as #fix grows at the file level: edit similarity is measured between the current version and the *baseline* version. Edit similarity to the baseline drops as the number of fixes grows (per file, from ~0.7 to ~0.43 by 16 fixes). Per fix, however, the LLM usually makes small, incremental edits (median edit similarity to the previous version is close to 1.0); a minority of fixes involve large changes. Specifically, four series (Overall, Timeout, Error, Error-free) all trend downward: more fixes lead to greater divergence from the baseline. The Timeout series has the lowest similarity from the second fix onwards (down to ~0.38 by 16 fixes), so notebooks that initially time out undergo the largest cumulative modification; the Error series remains higher but still declines to ~0.53. Thus, we argue that achieving reproducibility requires more than minor tweaks—cumulative changes can substantially alter the code; the extent of modification depends on the initial problem (timeout notebooks change the most).

The LLM primarily performs incremental edits per step, but cumulative file-level divergence from the baseline is substantial when many fixes are applied. Figure 8b reveals edit similarity per fix, comparing each version to the previous version (one fix step). Four series (Overall, Timeout, Error, Error-free) are concentrated at high similarity (median near 1.0), implying that most individual fixes are small and incremental. All categories exhibit a long tail to low similarity, so some fixes involve large edits; the overall shape is similar across categories, meaning that the amount of modification per fix does not differ drastically by fix type.

Summary: *Making notebooks reproducible often requires substantial cumulative code modification: edit similarity to the baseline drops as #fix grows (from ~0.7 to ~0.43 by 16 fixes), and notebooks that initially time out change the most. Per fix, the LLM usually makes small, incremental edits (median edit similarity to the previous version close to 1.0); a minority of fixes involve large edits. The LLM thus performs incremental edits per step, but cumulative divergence from the baseline becomes substantial when many fixes are applied.*

Table 5. Cost and token usage by MLEModernizer.

(a) GPT-5.2

		Cost (USD)	Average #Tokens		
			Input (not cached)	Input cached	Output
Per notebook		0.6453	52,197.54	35,817.71	39,118.81
Per Fix	Error-Repair	0.0612	5,507.29	3,244.12	3,641.50
	Score-Calibration	0.0611	4,928.20	3,821.76	3,703.09
	Runtime-Reduction	0.0590	3,829.15	2,474.31	3,703.73

(b) GPT-OSS-120b

		Cost (USD)	Average #Tokens		
			Input (not cached)	Input cached	Output
Per notebook		0.0074	63,697.77	1,135.59	26,962.14
Per Fix	Error-Repair	0.0008	6,952.89	100.78	2,763.56
	Score-Calibration	0.0007	6,132.29	146.10	2,384.63
	Runtime-Reduction	0.0007	4,624.97	28.12	2,847.35

4.4 RQ4: Cost

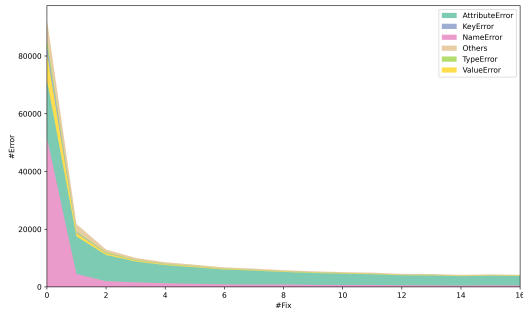
With GPT-5.2, applying MLEModernizer remains practical at scale: Table 5 reports the cost (USD) and average token counts at the per-notebook and per-fix level. Per notebook, the average cost is $0.65, with average token usage of 52,198 input (not cached), 35,818 input (cached), and 39,119 output. At the per-fix level, Error-Repair costs $0.061 on average (5,507 input not cached, 3,244 cached, 3,642 output), Score-Calibration $0.061 (4,928 not cached, 3,822 cached, 3,703 output), and Runtime-Reduction $0.059 (3,829 not cached, 2,474 cached, 3,704 output); the three fix types have similar per-fix cost, with Runtime-Reduction using fewer cached tokens. GPT-OSS-120b costs are estimated using OpenRouter rates in the same table: only $0.0074 per notebook on average, with higher uncached input volume (~64K) but minimal prompt caching (~1.1K cached). Although GPT-OSS is less functionally consistent than GPT-5.2 in RQ1, it offers a much cheaper alternative when cost is the primary concern, and some loss in modernization consistency is acceptable.

Summary: *With GPT-5.2, upgrading one notebook costs $0.65 on average (~52K uncached input, ~36K cached, ~39K output); each fix costs ~$0.06. GPT-OSS-120b (OpenRouter pricing) averages $0.0074 per notebook with far less caching but lower functional consistency, offering a low-cost alternative when budget is the primary constraint.*

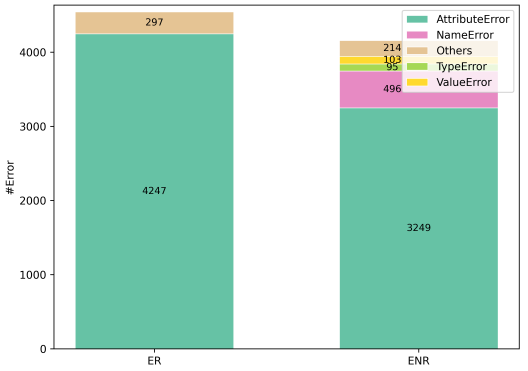
4.5 RQ5: Error Types

We analyze error types in each fix (including Error Reproducible notebooks) following the approach from prior work [26]. In Python, *NameError* is raised when a name is not defined (e.g., `print(x)` with `x` undefined or a missing import); *AttributeError* when an object has no requested attribute (e.g., `obj.foo` when `foo` does not exist); *ValueError* when an operation receives an invalid value of the expected type (e.g., `int('hello')`); *KeyError* when a dictionary key is missing (e.g., `d['key']`); *TypeError* when an operation is applied to an inappropriate type (e.g., `len(5)`). Such errors often arise from API or dependency changes, incorrect variable scope, or data shape mismatches.

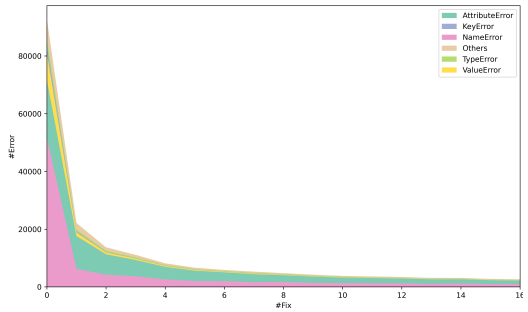
With GPT-5.2, we find that **MLEModernizer is highly effective at fixing *NameError*** (undefined name or missing import; baseline 51,772, reduced to 498 after 16 fixes) and substantially reduces *KeyError* and *ValueError*; *AttributeError* (missing or invalid attribute on an object) is the most stubborn—it remains dominant among persistent errors in both Error Reproducible and Error



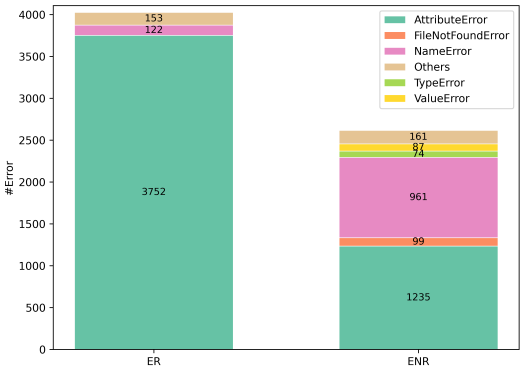
(a) Error types by number of LLM fixes (GPT-5.2).



(b) Error types at end of modernization (GPT-5.2).



(c) Error types by number of LLM fixes (GPT-OSS).



(d) Error types at end of modernization (GPT-OSS).

Fig. 9. Error-type evolution (left) and distribution at end of fixes (right) for both models.

Non-Reproducible notebooks after modernization. Figure 9a shows how error counts by type evolve as #fix increases. At baseline (#fix = 0), the top five types are *NameError* (51,772), *AttributeError* (19,923), *ValueError* (9,316), *KeyError* (2,773), and *TypeError* (2,407). Total errors drop sharply in the first few fixes (from >92K to ~20K by #fix = 1, ~10K by #fix = 2) and then decline more gradually. After 16 fixes, 4,183 crashes remain; *NameError* (51,772 → 498) falls sharply in early fixes but is not fully eliminated. *AttributeError* (19,923 → 3,265) becomes the dominant remaining type at higher #fix. MLEModernizer resolves many *NameErrors* effectively, but *AttributeErrors* are harder to fix. We argue that missing-import or undefined-name issues are amenable to automated repair; attribute and API-usage issues often require deeper or multi-step fixes.

Figure 9b shows the distribution of error types at each notebook's final fix for Error Reproducible and Error Non-Reproducible notebooks (only types ≥2% shown). *AttributeError* is the main stubborn type in both outcomes; *NameError* persists chiefly in Error Non-Reproducible notebooks, so fixing *NameError* is often necessary to reach reproducibility, while *AttributeErrors* frequently remain even when the notebook is already reproducible. In Error Reproducible notebooks, *AttributeError* dominates (4,247, 93.5%), followed by Others (297, 6.5%). In Error Non-Reproducible notebooks, *AttributeError* is again largest (3,249, 78.2%), followed by *NameError* (496, 11.9%).

Figures 9c and 9d show the same analysis for GPT-OSS-120b. After 16 fixes, 2,652 crashes remain—fewer total persistent crashes than GPT-5.2 but with a larger residual *NameError* count (498 → 965). In Error Reproducible notebooks, *AttributeError* overwhelmingly dominates (3,752,

93.2%). In Error Non-Reproducible notebooks, *AttributeError* (1,235, 47.2%) and *NameError* (961, 36.7%) remain the two most prevalent error types. GPT-OSS leaves substantially more *NameError* in Error Non-Reproducible notebooks than GPT-5.2 (36.7% vs. 11.9%), consistent with its lower functional-equivalence rate in RQ1.

Complementing the automatic error-type taxonomy, we manually label *crash causes* and *crash phases* on the inspected statistically sampled {before, after} notebook pairs. For GPT-5.2, *library issues* (LIB) constitute the most frequent crash root cause (55% before, 88% after), followed by *environment issues* (ENV; 39% before, 2% after); IRR=0.90. By crash phase, failures concentrate at *environment setup* (ENVS; 57% before, 85% after), then *data preparation* (DATAP; 22% before, 5% after); IRR=0.90. Thus, modernization primarily removes ENV-related failures and DATAP-stage crashes, while residual sampled crashes are increasingly LIB-dominated and ENVS-localized—consistent with persistent *AttributeError*s that reflect library/API drift under the new stack.

For GPT-OSS-120b, LIB again dominates (56% before, 74% after), but ENV remains more visible after upgrade (37% before, 9% after) than under GPT-5.2; IRR=0.84. By crash phase, ENVS is still primary (57% before, 73% after), while DATAP retains a larger post-upgrade share (23% before, 14% after) than GPT-5.2 (5%), indicating that **the open model less consistently clears early pipeline/setup failures**; IRR=0.91. Overall, the manual crash analysis shows that MLEModernizer fixes environment and data-preparation blockers first; the hardest remaining failures are library- and setup-stage issues, which align with stubborn *AttributeError*s and with GPT-OSS's higher residual *NameError* burden in Error Non-Reproducible notebooks.

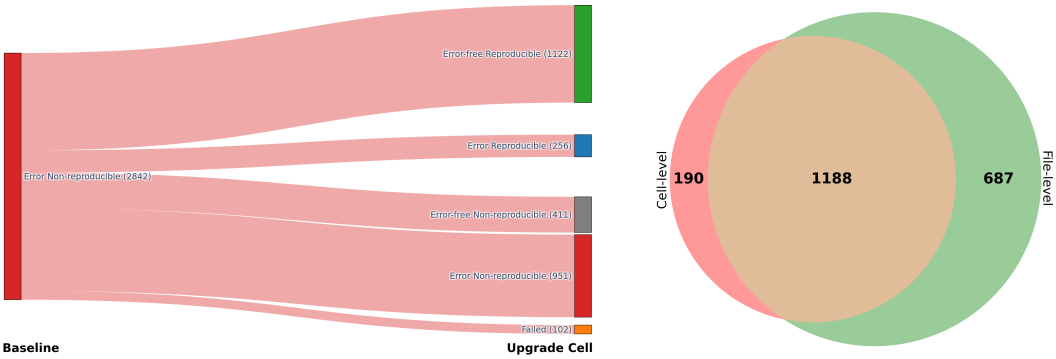
Summary: *MLEModernizer substantially reduces crash counts from >92K at baseline; after 16 fixes, GPT-5.2 leaves 4,183 persistent crashes and GPT-OSS-120b leaves 2,652. Both backends eliminate most* NameError/KeyError/ValueError *early, but* AttributeError *dominates remaining errors in Error Reproducible (>93%) and Error Non-Reproducible (>78% for GPT-5.2). Manual crash-cause/phase labeling shows modernization removes most ENV and DATAP failures; residual crashes are mainly LIB-related at ENVS. GPT-OSS retains more* NameError *in Error Non-Reproducible and more post-upgrade ENV/DATAP crashes than GPT-5.2, consistent with its weaker functional consistency in RQ1.*

5 Discussion

5.1 Alternative Repair Strategy: MLEModernizer with Cell-Level Error-Repair

MLEModernizer performs file-level error-repair, i.e., attempting to fix all errors at once. As a preliminary comparison on a subset of our dataset, we also explore an alternative strategy: cell-level error-repair, i.e., attempting to fix one error at a time. Specifically, in the error-repair prompt, we only provide the notebook cells up to the first cell with error(s), plus its traceback and the following cell, and instruct the LLM to focus on fixing the error. Because cell-level repair only diverges from file-level MLEModernizer when errors persist, we compare both repair strategies on the same subset: 2,842 Error Non-Reproducible notebooks that fail on baseline evaluation in a previous version of our experiment.

Figure 10a shows the results of cell-level error-repair. With cell-level repair, 1,378 notebooks become reproducible, while 1,362 remain non-reproducible, and 102 fail due to timeout/LLM failure. Figure 10b compares which notebooks can be modernized by file-level and cell-level repair. We find that file-level error-repair is slightly more effective, uniquely modernizing 687 notebooks versus 190 for cell-level error-repair. Given the stronger coverage of file-level error-repair, we therefore focus on file-level repair in this work.



(a) Notebook reproducibility transitions from Baseline to cell-level error-repair.

(b) Successfully modernized notebooks by cell- and file-level upgrades.

Fig. 10. Effectiveness of using cell-level error-repair in MLEModernizer.

5.2 Limitations

Offline Grading and Reproducibility Criteria. Since most Kaggle competitions do not publicly release their test sets, we use the offline-grading setup from MLE-Bench [2] to approximate the Kaggle evaluation. This leads to a mismatch between the training and test sets. To mitigate this, we classify reproducibility using a one-sample t-test (§2.1.2). Future work can explore other sources of notebooks where the test set is publicly available to enable a stricter reproducibility criterion.

Stochasticity of LLM Modernization. Each file-level modernization trajectory is executed once per notebook because re-running thousands of agentic LLM repair loops is prohibitively expensive (total cost > $5k USD). For reproducibility *measurement*, we repeat notebook execution up to 10 times and apply one-sample t-tests to account for score variance from stochastic training.

Generalizability. Our evaluation demonstrates generalizability to both a proprietary LLM (GPT-5.2) and an open-weight LLM (GPT-OSS-120b). We also observe that successful modernizations share many recurring library/API evolution patterns. Future work can study extracting generalizable patterns in the form of deterministic refactoring recipes, e.g., OpenRewrite scripts [18].

Flexible Error-Repair and Semantic Preservation. Some errors do not need to be fixed for notebooks to produce valid and reproducible predictions. As observed in §4.1, around 1/4 of the reproducible notebooks (GPT-5.2) still contain errors. Additionally, score-based reproducibility does not guarantee functional equivalence but is acceptable, as MLEModernizer instructs the model to preserve the original semantics while also enabling score calibration. As discussed in §4.1, these objectives may cause confusion or lead to repairs that restore the target score at the expense of semantic consistency. Future work will prioritize functionality-critical errors, separate structure-preserving repair from score calibration, and strengthen semantic validation to distinguish reproducibility from functional equivalence.

6 Related Work

Reproducibility of ML research and notebooks. Pimentel et al. [19] study the reproducibility of Jupyter notebooks on GitHub; their follow-up work introduces Julynter [20] for identifying potential reproducibility issues in Jupyter notebooks. Pineau et al. [21] analyze reproducibility in ML research papers through the NeurIPS reproducibility program, and Gundersen and Kjensmo [8], Henderson et al. [9] study reproducibility challenges in AI and deep RL. Wang et al. [26] characterize notebook

crash types by analyzing error outputs from notebooks mined from GitHub and Kaggle, without re-executing the notebooks. Yao et al. [28] introduce NBTest, a regression-testing framework that automatically generates cell-level assertions for machine learning notebooks. Elhashemy et al. [7] propose a multi-agent system for transforming notebooks from prototypes to production settings; their focus is workflow migration rather than code modernization under environment erosion. In contrast, our study focuses on large-scale re-execution in a containerized environment and evaluates not only execution success but also score deviation against the reported target.

Benchmarks and datasets for MLE notebooks. Yin et al. [29] build ARCADE, a benchmark for natural-language-to-code generation in data science notebooks, and introduce PACH-INCO, a notebook-centric code LM that reasons over cells and execution context. Mostafavi Ghahfarokhi et al. [17] release DistilKaggle, a distilled dataset of Kaggle notebooks and code-quality signals, and Chan et al. [2] introduce MLE-Bench, an offline benchmark derived from Kaggle competitions for evaluating ML engineering agents. Jin et al. [11] study learning to edit ML pipeline code in notebooks using LLMs and find low accuracy even after fine-tuning. Our work leverages MLE-Bench for large-scale, offline grading and studies how to modernize real-world Kaggle notebooks to restore reproducibility under contemporary environments. Dilhara et al. [6] mine and transplant evolution patterns in Python ML systems, complementing our code-modernization perspective.

LLM-based program repair and code evolution. Bouzenia et al. [1] present RepairAgent, an autonomous LLM-based agent for program repair that invokes tools and validation. Xie et al. [27] propose PReMM, an LLM-based repair technique for multi-method bugs, and Deligiannis et al. [5], Zhang et al. [30] study LLM-driven repair for Python and Rust, respectively. Wang et al. [25] analyze deprecated API usage in LLM-based code completion and propose lightweight mitigation strategies. Compared with prior program-repair work, our setting emphasizes notebook-specific execution context (cells and state), a fixed modern environment, and *score-aware* repair: a patch is only acceptable if it restores end-to-end execution and brings the reproduced score within the reproducible band of the reported target score.

7 Conclusions

We investigate the reproducibility challenges of MLE notebooks. In a large-scale study of 12,106 Kaggle notebooks, only 26% are reproducible, and environment backporting hurts instead of helping. We then develop MLEModernizer, an LLM-driven agentic framework for modernizing notebooks toward reproducibility as an initial exploration of this approach. MLEModernizer iteratively executes notebooks and applies three types of targeted fixes: error-repair, runtime-reduction, and score-calibration. Our evaluation shows that MLEModernizer successfully modernizes 40.1%–44.9% of the notebooks that are non-reproducible in the baseline environment. These results suggest that, although code modernization is a best-effort recovery technique, it offers a viable path to restoring reproducibility for legacy MLE notebooks, enabling more reliable validation and reuse of ML pipelines in a rapidly evolving hardware and software environment.

Acknowledgments

We thank Ian Chen, Yuntian Deng, Yinxi Li, Yu Liu, Chengnian Sun, Shirley Xiao, and the anonymous reviewers for their comments and feedback. This work is partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) under funding reference RGPIN2024-04909. Bihui Jin is additionally supported by the NSERC Canada Graduate Research Scholarship–Doctoral Program (CGRS D).

Data Availability

We provide an artifact package containing our MLEModernizer implementation, replication scripts, the MLE-Bench grader components used for evaluation, and the curated list of notebook identifiers and metadata from Meta Kaggle and Meta Kaggle Code [12, 13]. Notebook content and metadata were collected from Kaggle's official Meta Kaggle data dumps, as described in §2.1.1. We do not redistribute the competition datasets themselves, which can be downloaded on demand using our provided scripts. Our artifact is available at https://github.com/Bihui-Jin/MLEModernizer.

References

[1] Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. 2025. RepairAgent: An Autonomous, LLM-Based Agent for Program Repair. In *International Conference on Software Engineering* (Ottawa, Ontario, Canada) *(ICSE '25)*. 2188–2200. doi:10.1109/ICSE55347.2025.00157

[2] Jun Shern Chan, Neil Chowdhury, Oliver Jaffe, James Aung, Dane Sherburn, Evan Mays, Giulio Starace, Kevin Liu, Leon Maksin, Tejal Patwardhan, Lilian Weng, and Aleksander Mądry. 2025. MLE-bench: Evaluating Machine Learning Agents on Machine Learning Engineering. In *International Conference on Learning Representations*. 22 pages. https://proceedings.iclr.cc/paper_files/paper/2025/hash/7e3767db483c942b883eb4f8cfb74e31-Abstract-Conference.html

[3] Jacob Cohen. 1960. A Coefficient of Agreement for Nominal Scales. *Educational and Psychological Measurement* 20, 1 (1960), 37–46. doi:10.1177/001316446002000104

[4] damnever. 2025. *pigar 2.2.0*. https://pypi.org/project/pigar/2.2.0/

[5] Pantazis Deligiannis, Akash Lal, Nikita Mehrotra, Rishi Poddar, and Aseem Rastogi. 2025. RustAssistant: Using LLMs to Fix Compilation Errors in Rust Code. In *International Conference on Software Engineering*. 3097–3109. doi:10.1109/ICSE55347.2025.00022

[6] Malinda Dilhara, Danny Dig, and Ameya Ketkar. 2023. PyEvolve: Automating Frequent Code Changes in Python ML Systems. In *International Conference on Software Engineering*. 995–1007. doi:10.1109/ICSE48619.2023.00091

[7] Hanya Elhashemy, Youssef Lotfy, and Yongjian Tang. 2025. Bridging the Prototype-Production Gap: A Multi-Agent System for Notebooks Transformation. In *International Conference on Automated Software Engineering Workshops*. 299–302. doi:10.1109/ASEW67777.2025.00061

[8] Odd Erik Gundersen and Sigbjørn Kjensmo. 2018. State of the Art: Reproducibility in Artificial Intelligence. In *AAAI Conference on Artificial Intelligence*. 1644–1651. doi:10.1609/aaai.v32i1.11503

[9] Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. 2018. Deep Reinforcement Learning That Matters. In *AAAI Conference on Artificial Intelligence*. 3207–3214. doi:10.1609/aaai.v32i1.11694

[10] Shantanu Jain. 2025. *tiktoken 0.12.0*. https://pypi.org/project/tiktoken/0.12.0/

[11] Bihui Jin, Jiayue Wang, and Pengyu Nie. 2025. Learning to Edit Interactive Machine Learning Notebooks. In *Companion Proceedings of the International Conference on the Foundations of Software Engineering*. 681–685. doi:10.1145/3696630.3728523

[12] Kaggle. 2025. *Meta Kaggle*. https://www.kaggle.com/datasets/kaggle/meta-kaggle/data

[13] Kaggle. 2025. *Meta Kaggle Code*. https://www.kaggle.com/datasets/kaggle/meta-kaggle-code/data

[14] Kaggle. 2025. *Notebooks Documentation: Technical Specifications*. https://www.kaggle.com/docs/notebooks#technical-specifications

[15] Kaggle. 2025. *Notebooks Documentation: The Notebook Environment*. https://www.kaggle.com/docs/notebooks#the-notebooks-environment

[16] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, Carol Willing, and Jupyter Development Team. 2016. Jupyter Notebooks—a publishing format for reproducible computational workflows. In *Positioning and Power in Academic Publishing: Players, Agents and Agendas*. 87–90. doi:10.3233/978-1-61499-649-1-87

[17] Mojtaba Mostafavi Ghahfarokhi, Arash Asgari, Mohammad Abolnejadian, and Abbas Heydarnoori. 2024. DistilKaggle: A Distilled Dataset of Kaggle Jupyter Notebooks. In *International Working Conference on Mining Software Repositories*. 647–651. doi:10.1145/3643991.3644882

[18] OpenRewrite. 2026. *OpenRewrite by Moderne: Large Scale Automated Refactoring*. https://docs.openrewrite.org/

[19] João Felipe Pimentel, Leonardo Murta, Vanessa Braganholo, and Juliana Freire. 2019. A Large-scale Study about Quality and Reproducibility of Jupyter Notebooks. In *International Working Conference on Mining Software Repositories*. 507–517. doi:10.1109/MSR.2019.00077

[20] João Felipe Pimentel, Leonardo Murta, Vanessa Braganholo, and Juliana Freire. 2021. Understanding and Improving the Quality and Reproducibility of Jupyter Notebooks. *Empirical Software Engineering* 26, 4, Article 65 (2021), 41 pages. doi:10.1007/s10664-021-09961-9

[21] Joelle Pineau, Philippe Vincent-Lamarre, Koustuv Sinha, Vincent Larivière, Alina Beygelzimer, Florence d'Alché-Buc, Emily Fox, and Hugo Larochelle. 2021. Improving Reproducibility in Machine Learning Research (A Report from the NeurIPS 2019 Reproducibility Program). *Journal of Machine Learning Research* 22, 164 (2021), 1–20. https://www.jmlr.org/papers/v22/20-303.html

[22] Adam Rule, Aurélien Tabard, and James D. Hollan. 2018. Exploration and Explanation in Computational Notebooks. In *CHI Conference on Human Factors in Computing Systems*. Article 32, 12 pages. doi:10.1145/3173574.3173606

[23] Carolyn B. Seaman. 1999. Qualitative Methods in Empirical Studies of Software Engineering. *Transactions on Software Engineering* 25, 4 (1999), 557–572. doi:10.1109/32.799955

[24] April Yi Wang, Dakuo Wang, Jaimie Drozdal, Michael Muller, Soya Park, Justin D. Weisz, Xuye Liu, Lingfei Wu, and Casey Dugan. 2022. Documentation Matters: Human-Centered AI System to Assist Data Science Code Documentation in Computational Notebooks. *Transactions on Computer-Human Interaction* 29, 2, Article 17 (2022), 33 pages. doi:10.1145/3489465

[25] Chong Wang, Kaifeng Huang, Jian Zhang, Yebo Feng, Lyuye Zhang, Yang Liu, and Xin Peng. 2025. LLMs Meet Library Evolution: Evaluating Deprecated API Usage in LLM-Based Code Completion. In *International Conference on Software Engineering*. 885–897. doi:10.1109/ICSE55347.2025.00245

[26] Yiran Wang, Willem Meijer, José Antonio Hernández López, Ulf Nilsson, and Dániel Varró. 2025. Why do Machine Learning Notebooks Crash? An Empirical Study on Public Python Jupyter Notebooks. *Transactions on Software Engineering* 51, 7 (2025), 2181–2196. doi:10.1109/TSE.2025.3574500

[27] Linna Xie, Zhong Li, Yu Pei, Zhongzhen Wen, Kui Liu, Tian Zhang, and Xuandong Li. 2025. PReMM: LLM-Based Program Repair for Multi-method Bugs via Divide and Conquer. *Proceedings of the ACM on Programming Languages* 9, OOPSLA2, Article 319 (Oct. 2025), 29 pages. doi:10.1145/3763097

[28] Yingao (Elaine) Yao, Vedant Nimje, Varun Viswanath, and Saikat Dutta. 2026. Automated Assertion Generation and Regression Testing for Machine Learning Notebooks. In *International Conference on Automated Software Engineering*. to appear. https://arxiv.org/abs/2509.13656

[29] Pengcheng Yin, Wen-Ding Li, Kefan Xiao, Abhishek Rao, Yeming Wen, Kensen Shi, Joshua Howland, Paige Bailey, Michele Catasta, Henryk Michalewski, Oleksandr Polozov, and Charles Sutton. 2023. Natural Language to Code Generation in Interactive Data Science Notebooks. In *Annual Meeting of the Association for Computational Linguistics*. 126–173. doi:10.18653/v1/2023.acl-long.9

[30] Jialu Zhang, José Pablo Cambronero, Sumit Gulwani, Vu Le, Ruzica Piskac, Gustavo Soares, and Gust Verbruggen. 2024. PyDex: Repairing Bugs in Introductory Python Assignments using LLMs. *Proceedings of the ACM on Programming Languages* 8, OOPSLA1, Article 133 (April 2024), 25 pages. doi:10.1145/3649850

Received 2026-01-30; accepted 2026-06-25