Original Article | Open Access

AI Agent

Lyu *et al. AI Agent* **2026**, *2*, 23 **DOI:**10.20517/aiagent.2026.31

# When code does not run: reproducibility challenges in materials machine learning benchmarks

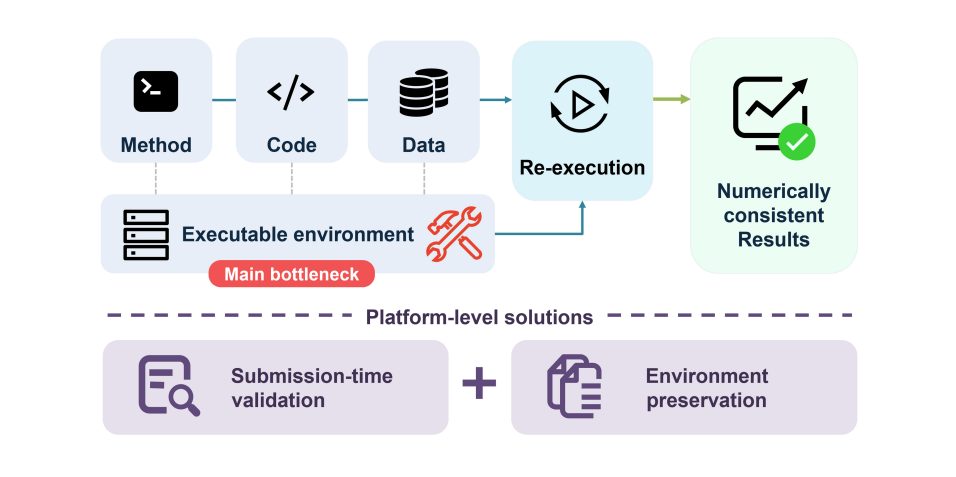
**Bohui Lyu[1], John Bonini[2], Mao Zhang[1], Amin Sadeghi[3], Ali Jaberi[3], Jason Hattrick-Simpers[3,4], Kamal Choudhary[2,5], Daniel Wines[2], Kangming Li[1]**

**Keywords:**
Machine learning benchmark, computational reproducibility, software environment, materials science

**Citation:** Lyu, B.; Bonini, J.; Zhang, M.; Sadeghi, A.; Jaberi, A.; Hattrick-Simpers, J.; Choudhary, K.; Wines, D.; Li, K. When code does not run: reproducibility challenges in materials machine learning benchmarks. *AI Agent* **2026**, *2*, 23. https://dx.doi.org/10.20517/aiagent.2026.31

**Received:** 22 Jun 2026 **First Decision:** 22 Jul 2026 **Revised:** 17 Aug 2026 **Accepted:** 24 Aug 2026 **Published:** 28 Sep 2026

**Academic Editor:** Hao Li
**Copy Editor:** Tong Wang
**Production Editor:** Tong Wang




## Abstract

Reproducible benchmarks are essential for advancing data-driven materials research by standardizing datasets, prediction tasks, and evaluation protocols. In machine learning benchmarks, reproducibility is often assumed to follow from releasing source code and documented software dependencies. However, dependency specifications must be sufficiently complete to reconstruct an executable software environment from a defined, isolated starting point. Because modern package ecosystems are complex and evolve over time, automated environment reconstruction and sandboxed executability checks are necessary prerequisites for assessing numerical reproducibility. Here, we systematically evaluate this aspect of reproducibility using MatBench, a benchmark platform for materials property prediction. Using the original dependency metadata without modification, only 2 of 28 submissions produced environments that could be installed and pass import checks. Rule-based and human-assisted remediation increased this number to 26 of 28. Among these, 13 submissions successfully completed a selected representative benchmark task



[1]Division of Physical Science and Engineering, King Abdullah University of Science and Technology (KAUST), Thuwal 23955-6900, Saudi Arabia.
[2]Material Measurement Laboratory, National Institute of Standards and Technology, Gaithersburg, MD 20899, USA.
[3]Department of Materials Science and Engineering, University of Toronto, Toronto M5S 3E4, Canada.
[4]Acceleration Consortium, University of Toronto, Toronto M5S 3E4, Canada.
[5]Department of Electrical and Computer Engineering, Whiting School of Engineering, Johns Hopkins University, Baltimore, MD 21218, USA.

**Correspondence to:** Prof. Kangming Li, Division of Physical Science and Engineering, King Abdullah University of Science and Technology (KAUST), Thuwal 23955-6900, Saudi Arabia. E-mail: kangming.li@kaust.edu.sa

and generated outputs, corresponding to 46.4% of all evaluated submissions. For these successfully re-executed submissions, the regenerated five-fold mean root mean square error (RMSE) values were close to the originally reported results. Similar executability issues were also observed on the JARVIS-Leaderboard. These findings demonstrate that reconstructing runnable environments from currently provided benchmark metadata remains challenging. We identify common causes of failure and propose platform-level practices for improving the long-term reproducibility and executability of machine learning benchmarks in materials informatics.

## INTRODUCTION

A description of the synthesis of a particular molecule, given as a set of instructions and reactants, which only reliably yields the specified products when performed with one beaker the chemist never washes, would not be recognized as complete or scientifically reproducible. A description of a computational workflow that only reliably runs on the compute resources where it was developed should be regarded similarly. Reproducibility is fundamental to the reliability of scientific conclusions, accelerates algorithmic iteration, and enables standardized head-to-head comparisons across methods[1,2].

Accordingly, ensuring the reproducibility of computational workflows and predictive results has become imperative, as underscored by the FAIR (Findability, Accessibility, Interoperability, and Reusability) principles for digital assets[3].

With the rapid development of machine learning, its applications in materials science have expanded markedly. Tasks such as rapid screening of *in silico*-generated materials[4-6], acceleration of molecular simulations[7,8], inverse design of novel materials[9-11], and autonomous experimentation[12,13] have substantially advanced the field, enabling the discovery and development of new materials. The reproducibility of machine learning workflows in materials science remains an ongoing challenge and has received increasing attention[14,15]. To support transparent comparison and algorithm reuse, benchmark platforms have been developed that standardize datasets and evaluation pipelines. Analogous to benchmark platforms in computer science[14-16], AI for materials science has produced a growing suite of benchmarking resources[17-19]. These platforms typically require source code together with software requirements in accompanying metadata. However, the executability of these resources is often overlooked, whereas reproducing results typically requires setting up the correct software environment to run the source code. If that environment cannot be established, the available data and code alone are insufficient to reproduce the reported results.

Among benchmark platforms for materials science, MatBench is a machine-learning benchmark that standardizes datasets, prediction tasks, evaluation splits, and leaderboards for comparing materials-property prediction models. MatBench comprises 13 supervised tasks covering composition- and structure-based prediction of key inorganic materials properties [Figure 1]. Its workflow uses predefined five-fold cross-validation, with model predictions and evaluation metrics recorded through the MatBench framework. The platform supports comparison across diverse models, from descriptor-based pipelines to advanced crystal graph neural networks.

As a case study, we attempted to reproduce the reported results on MatBench by re-executing the original evaluation pipeline of 28 algorithms using the provided source code and documented environment specifications. Unexpectedly, the process was far from straightforward: we encountered numerous obstacles, most of which were related to the inability to instantiate Python environments from the specifications provided by the platforms. While we managed to set up most environments by relaxing software version constraints, the resulting environments often differed from those originally used to generate the uploaded results, which in turn prevented the successful execution of the source code or reproduction of the uploaded

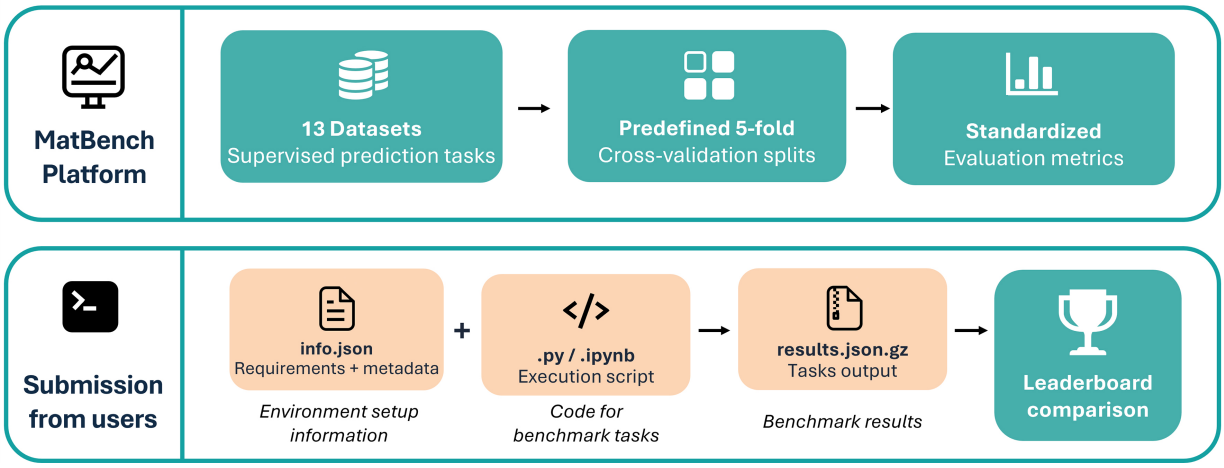



**Figure 1.** MatBench infrastructure and submission protocol.

results. We therefore categorized the algorithms according to the difficulty of environment reconstruction and summarized the problems encountered at different stages of the reproduction process. Accordingly, we identify common obstacles to re-execution and propose platform-level recommendations for improving benchmark reproducibility.

## METHOD

### Assessment framework

Each MatBench submission contains three principal components: a metadata file (info.json) describing the algorithm and its software requirements, source code for training and evaluating the model, and a compressed result file (results.json.gz) containing the submitted predictions and benchmark metrics [Figure 1]. We evaluated 28 submitted algorithms using their publicly available metadata and source code, while treating the submitted result files as the reference for numerical comparison.

The assessment comprised three sequential levels. First, environment reconstruction tested whether the dependency information supplied with a submission was sufficient to establish an import-valid Python environment. An environment was considered successfully reconstructed when, under at least one tested Python version and installation route, dependency installation completed and all imports identified in the submitted execution scripts could be loaded without error. This criterion evaluates environment availability and import compatibility but does not establish that the complete benchmark workflow is executable.

Second, code executability tested whether the submitted source code could complete a selected representative MatBench task and generate benchmark outputs in the reconstructed environment. Third, numerical result reproducibility quantified the difference between the regenerated benchmark metric and the value recorded in the original submission. These three levels were evaluated separately because successful environment reconstruction does not necessarily guarantee successful code execution, and successful execution does not necessarily guarantee numerical agreement with the reported result. The overall workflow is illustrated in Figure 2.

### Environment reconstruction

Environment reconstruction was conducted in clean, isolated Linux runners using GitHub Actions. For each submission, the dependency specifications in info.json were tested across Python versions 3.6-3.12 using pip and conda-based installation routes. After installation, the pipeline automatically scanned the source code to collect all import statements and performed an import-based sanity check. The reconstruction procedure comprised three stages of progressively increasing intervention.

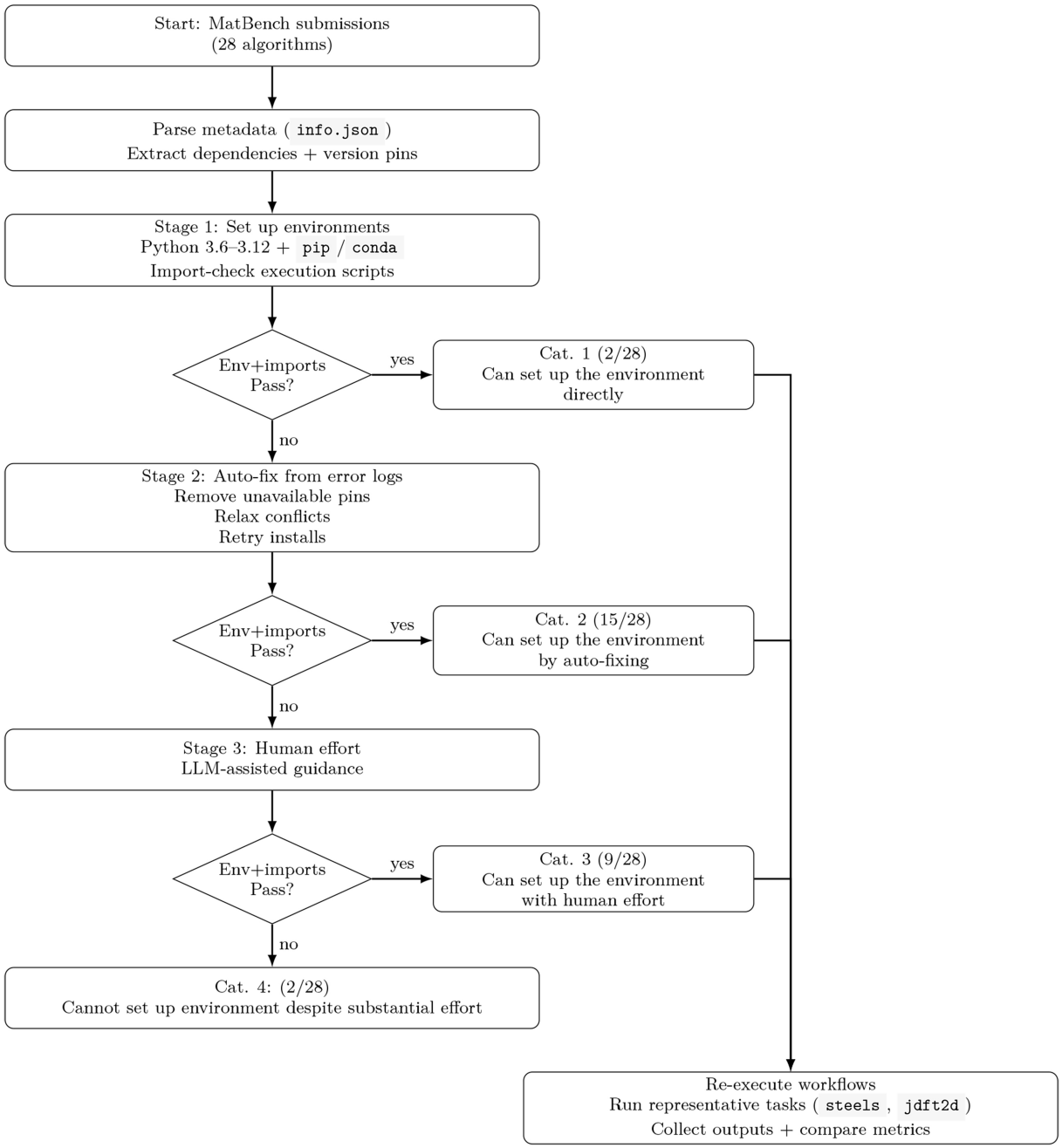



**Figure 2.** Operational workflow used in this study to assess practical reproducibility on MatBench. LLM: Large language model.

In Stage 1, dependencies were installed using the author-provided package names and version constraints without modification. Submissions that completed installation and passed the import check were assigned to Category 1 (Cat. 1).

Submissions that failed Stage 1 proceeded to Stage 2, in which predefined error-guided rules were used to relax unavailable or conflicting version constraints. The rules operated only on dependency specifications and did not modify the submitted model or evaluation code. Submissions that passed the import check after this automated remediation were assigned to Cat. 2. The error-classification rules, retry procedure, and workflow pseudocode are provided in Supplementary Method 1.

Submissions that remained unresolved proceeded to Stage 3. At this stage, the metadata and execution scripts were reviewed manually to identify omitted dependencies and potential incompatibilities. Package versions, installation order, and the use of pip, conda, or a combination of the two were adjusted where necessary. A large language model (LLM) was used as a human-supervised troubleshooting aid for selected difficult cases, but all suggested changes were reviewed, implemented, and validated by the authors. No submitted model or evaluation code was modified during environment reconstruction. The complete intervention protocol, stopping criteria, and LLM-assisted procedure are described in Supplementary Method 2.

The four environment categories were assigned according to the minimum level of intervention required to satisfy the environment-reconstruction criterion:

These categories describe the reconstruction pathway and the minimum intervention required; they do not indicate whether the source code subsequently executed successfully or whether the regenerated numerical results agreed with the original submission. Cat. 4 likewise denotes failure under the predefined testing protocol and should not be interpreted as proof that reconstruction is impossible under all configurations.

### Code re-execution and numerical comparison

Submissions in Cat. 1-3 were advanced to code re-execution. Their reconstructed environments were installed on an High-Performance Computing (HPC) system running Rocky Linux 9.4 (Blue Onyx), equipped with an NVIDIA A100-SXM4-80GB Graphics Processing Unit (GPU) and CUDA 12.8. Environments that passed the GitHub Actions validation could also be installed on this system.

To limit the computational cost while covering both MatBench input classes, we selected the smallest representative task from each class: steels for composition-based models and jdft2d for structure-based models. Each submission was evaluated on the applicable task according to its input type. Any submission-specific exceptions and the task selected for each algorithm are provided in Supplementary Table 1, available as a separate Excel file. Code re-execution was considered successful when the submitted workflow completed the selected task and generated predictions and benchmark metrics in the expected MatBench output format. Runtime failures and their causes were recorded separately from failures of environment reconstruction.

For submissions that completed re-execution, numerical agreement was quantified using the relative difference between the regenerated and originally reported five-fold mean root mean square error (RMSE):

$$\Delta_{\mathrm{RMSE}}\,(\%) = \frac{\mathrm{RMSE}_{\text{re-executed}} - \mathrm{RMSE}_{\text{reported}}}{\mathrm{RMSE}_{\text{reported}}} \times 100. \tag{1}$$

Here, $\mathrm{RMSE}_{\text{reported}}$ is the mean RMSE across the five predefined cross-validation folds recorded in the original submission, and $\mathrm{RMSE}_{\text{re-executed}}$ is the corresponding value obtained through re-execution. Positive values indicate higher error in the re-executed result, whereas negative values indicate lower error. We report the signed and absolute differences rather than imposing an arbitrary binary threshold for numerical reproducibility.

## RESULTS AND DISCUSSION

### MatBench

#### *Environment reconstruction*

##### Stage 1

We first attempted to build the environments by strictly following the dependency specifications provided by the authors. This approach represents the most straightforward workflow from an end user's perspective. Among 28 distinct algorithms, only two of them could be installed and all functions required by the execution scripts could be successfully imported. The Auto-sklearn submission provides an environment backup file (environment.yml), enabling the environment to be installed directly, whereas the Ax_10_90_CrabNet_v1.2.7 implementation installs cleanly on Python 3.7. For the remaining cases, neither conda nor pip could satisfy all version constraints. The failures mainly fall into three categories [Table 1]: (i) For CrabNet the metadata was not machine-readable in practice, preventing our scripts from extracting the necessary information; for this case, we attempted manual environment reconstruction in Stage 3; (ii) Unavailability of specific pinned package versions. The most frequent error arose from a hard requirement on matbench v0.1.0; this version is no longer available in standard package repositories, leading to installation failures in 11 environments; (iii) Version conflicts among dependencies. These errors commonly occur when the version constraints for packages such as matbench, scikit-learn, pymatgen, and matminer cannot be satisfied simultaneously, resulting in environment reconstruction failures. The latter two types of errors may be attributable to the age of the submissions, all of which were uploaded two to three years ago. Over this period, the underlying package ecosystem has evolved substantially: some older versions have been deprecated or removed from distribution channels, and newer releases have introduced incompatibilities and dependency conflicts. We therefore infer that these changes collectively contributed to the installation and runtime errors observed in Stage 1. These unresolved issues motivated a second stage focused on error-guided automatic remediation.

##### Stage 2

Based on the error messages from the algorithms that failed in Stage 1, we hypothesized that appropriately relaxing the version constraints of the implicated packages could alleviate many installation failures. Therefore, we extracted the error logs and attempted to fix the environment reconstruction process automatically, which we refer to as Stage 2. Following these error-guided fixes, 15 of the remaining 26 algorithms installed successfully and completed the import checks without errors. The detailed procedures and rationale are provided in the Methods section. For the remaining 11 algorithms, we observed two principal issues [Table 2]. (i) Several packages invoked by the code were not fully enumerated in the metadata files, resulting in import errors; (ii) Even after removing all version pins, some environments still exhibited dependency conflicts that prevented installation. These issues are difficult to resolve reliably through a fully automated workflow. Therefore, for the remaining algorithms, we adopted an iterative troubleshooting strategy that combined manual intervention with LLM-assisted guidance to provision the environments.

##### Stage 3

As shown in Table 2, incomplete dependency information caused import failures for several algorithms. We therefore identified and manually installed the missing packages before repeating the import checks. For submissions whose metadata could not be reliably parsed by our automated workflow [Table 1], the environments were reconstructed manually following the authors instructions. Where necessary, this process was supported by LLM-assisted troubleshooting, particularly to identify compatible package versions and

**Table 1. Representative package installation errors**

| Error type | Example output |
|---|---|
| Metadata is not machine-readable | CrabNet<br>Extracting packages from info.json...<br>AttributeError: 'str' object has no attribute 'get' |
| Version not found | automatminer_expressv2020, darwin, cgcnnv2019, GN-OA, *etc.*<br>ERROR: Could not find a version that satisfies the requirement matbench==0.1.0 (from versions: 0.2, 0.3, 0.4, 0.5, 0.6)<br>modnet_v0.1.10<br>ERROR: Could not find a version that satisfies the requirement modnet==0.1.10 (from versions: 0.1.1, 0.1.2, 0.1.3, 0.1.4, 0.1.5, 0.1.6, 0.1.7, 0.1.8, 0.1.9, 0.1.10.dev0, 0.1.11.dev0, 0.1.11, 0.1.12.dev0, 0.1.12, 0.1.13, 0.2.0, 0.2.1, 0.3.0, 0.4.0, 0.4.1, 0.4.2, 0.4.3, 0.4.4, 0.4.5)<br>ERROR: No matching distribution found for modnet==0.1.10 |
| Dependency conflict | modnet_v0.1.12<br>The conflict is caused by:<br>modnet 0.1.12 depends on pymatgen <2020.9 and >= 2020<br>matbench 0.2 depends on pymatgen==2021.2.16<br>TPOT<br>The conflict is caused by:<br>The user requested scikit-learn==1.2.2<br>tpot 0.11.7 depends on scikit-learn>=0.22.0<br>matbench 0.6 depends on scikit-learn==1.0.1<br>matbench 0.5 depends on scikit-learn==1.0<br>matbench 0.4 depends on scikit-learn==1.0<br>matbench 0.3 depends on scikit-learn==0.24.2<br>matbench 0.2 depends on scikit-learn==0.24.1<br>Ax_CrabNet_v1.2.1<br>ERROR: Cannot install ax-platform==0.2.3, crabnet==1.2.1, matbench==0.5 and scikit_learn==1.0.2 because these package versions have conflicting dependencies.<br>The conflict is caused by:<br>The user requested scikit_learn==1.0.2<br>ax-platform 0.2.3 depends on scikit-learn<br>crabnet 1.2.1 depends on scikit-learn<br>matbench 0.5 depends on scikit-learn==1.0 |

**Table 2. Representative package installation errors**

| Error type | Example output |
|---|---|
| Missing packages in the metadata files | Matformer<br>from torch_scatter import gather_csr, scatter, segment_csr ModuleNotFoundError: No module named 'torch_scatter<br>lattice_xgboost<br>import google<br>ModuleNotFoundError: No module named 'google' |
| Without all version pins, some environments still exhibited dependency conflicts | automatminer_expressv2020<br>The conflict is caused by:<br>automatminer 1.0.0.20191110 depends on matminer==0.6.2<br>matbench 0.6 depends on matminer==0.7.4<br>automatminer 1.0.0.20191110 depends on matminer==0.6.2<br>matbench 0.5 depends on matminer==0.7.4<br>automatminer 1.0.0.20191110 depends on matminer==0.6.2<br>matbench 0.4 depends on matminer==0.7.4<br>automatminer 1.0.0.20191110 depends on matminer==0.6.2<br>matbench 0.3 depends on matminer==0.7.3<br>automatminer 1.0.0.20191110 depends on matminer==0.6.2<br>matbench 0.2 depends on matminer==0.6.5<br>coGN<br>ERROR: pip's dependency resolver does not currently take into account all the packages that are installed. This behavior is the source of the following dependency conflicts<br>kgcnn 3.0.0 requires numpy>=1.23.0, but you have numpy 1.22.4 which is incompatible. kgcnn 3.0.0 requires scikit-learn>=1.1.3, but you have scikit-learn 1.0.1 which is incompatible<br>kgcnn 3.0.0 requires scipy>=1.9.3, but you have scipy 1.7.3 which is incompatible<br>pyxtal 1.1.3 requires pymatgen>=2024.3.1, but you have pymatgen 2023.9.25 which is incompatible<br>tensorflow 2.20.0 requires numpy>=1.26.0, but you have numpy 1.22.4 which is incompatible<br>from numpy._typing import ArrayLike<br>ModuleNotFoundError: No module named 'numpy._typing' |

**Table 3. Categories of MatBench algorithms based on reproducibility of the Python environment**

| Categories | Algorithms |
|---|---|
| Cat. 1: Can set up the environment directly (2/28) | Auto-sklearn, Ax_10_90_CrabNet_v1.2.7 |
| Cat. 2: Can set up the environment by auto-fixing (15/28) | alignn, Ax_CrabNet_v1.2.1, Ax_SAASBO_CrabNet_v1.2.7, DimeNetPP_kgcnn_v2.1.0, dummy, Finder_v1.2_composition, Finder_v1.2_structure, GN-OA, gptchem, MegNet_kgcnn_v2.1.0, modnet_v0.1.10, modnet_v0.1.12, RFLR, SchNet_kgcnn_v2.1.0, TPOT |
| Cat. 3: Can set up the environment with human effort (9/28) | automatminer_expressv2020, CrabNet, cgcnnv2019, CrabNet_v1.2.1, darwin, lattice_xgboost, matformer, coGN, coNGN |
| Cat. 4: Cannot set up environment despite substantial effort (2/28) | DeeperGATGNN, rf |

adjust the installation order. Stage 3 successfully recovered environments for 9 additional algorithms, while 2 remained unresolved after all permitted troubleshooting attempts. The final classification is summarized in Table 3.

### *Reproducing the benchmark results*

To set up the Python environments successfully, we adjusted version specifications for selected packages, either automatically through our scripts or manually, which resolved many installation errors. However, such modifications may introduce potential risks: the execution scripts may become incompatible, or replacing author-specified algorithm versions may affect benchmark outcomes.

Execution of the provided code showed that only half of the algorithms with successfully reconstructed environments produced benchmark outputs. For the remaining algorithms, runtime failures occurred because the required version of a key dependency could not be installed. As shown in Figure 3, these failures resulted from renamed Application Programming Interfaces (APIs), changed entry points after package updates, or relocated modules after code refactoring. For example, the training script in alignn was previously invoked as train_folder, whereas in newer releases it has been renamed to train_alignn. Similarly, modnet imports Structure directly from pymatgen, whereas, in the reconstructed environment, the installed pymatgen version exposes Structure only through pymatgen.core.structure. In addition, gptchem attempts to access an OpenAI token resource; however, the referenced API endpoint is no longer available, resulting in an HTTP 404 ("Not Found") error. In principle, one could modify the execution scripts or patch the corresponding package source code to restore compatibility; however, such interventions are beyond the scope of this manuscript. We also identified a special case that led to irreproducibility for reasons unrelated to dependency resolution: the run.py file in the cgcnnv2019 submission is empty. Thus, even though the environment can be installed, the benchmark tasks cannot be executed. This issue could be addressed by correcting the submission files. The names of the 26 algorithms and their corresponding code executability outcomes are summarized in Table 4.

The comparison between the results obtained in our reproduction runs and those reported in the submissions is shown in Figure 4. For the 13 submissions that completed re-execution on the selected representative tasks, the regenerated five-fold mean RMSE values were close to the originally reported values, with the maximum absolute difference being 7.2%. This numerical agreement suggests that, for these successfully re-executed submissions and the specific tasks examined, the benchmark outcomes were relatively stable once a runnable environment had been established.

## JARVIS-Leaderboard

Environment instantiation was also identified as a nontrivial issue in a parallel reproducibility effort we conducted on the JARVIS-Leaderboard benchmark set[19]. Submissions to this platform contain model predictions on the platform benchmarks, an executable run.sh script (which in most cases executes a python

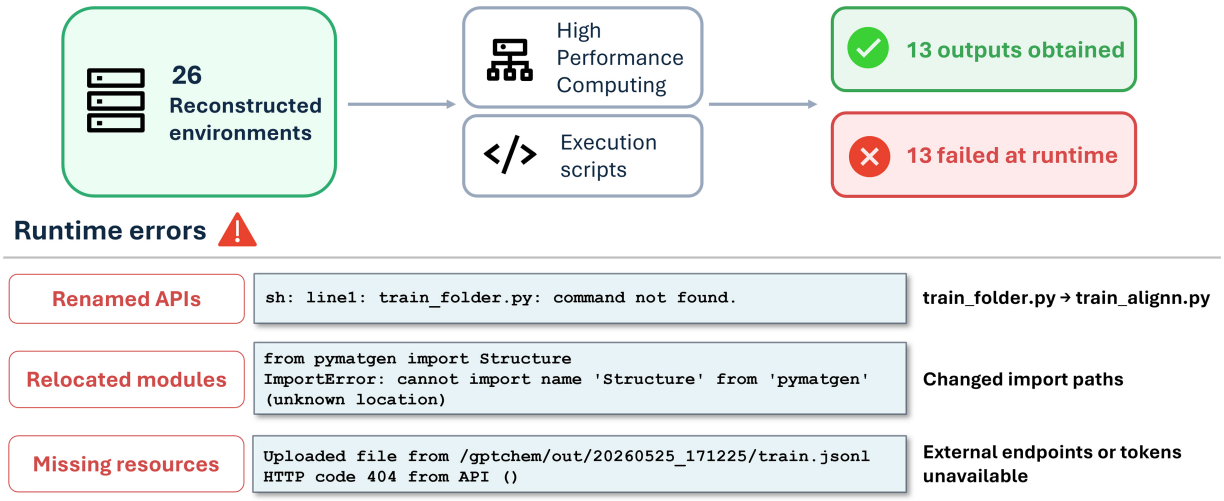



**Figure 3.** Code execution results and runtime error messages. API: Application Programming Interface.

**Table 4. Executability of the submitted code in the 26 reconstructed environments**

| Executability | Algorithms |
|---|---|
| Can reproduce the result (13/26) | Auto-sklearn, Ax_10_90_CrabNet_v1.2.7, Ax_CrabNet_v1.2.1, CrabNet, CrabNet_v1.2.1, coGN, coNGN, DimeNetPP_kgcnn_v2.1.0, dummy, Finder_v1.2_composition, Finder_v1.2_structure, RFLR, lattice_xgboost |
| Cannot obtain the benchmark result (13/26) | alignn, automatminer_expressv2020, Ax_SAASBO_CrabNet_v1.2.7, cgcnnv2019, darwin, GN-OA, gptchem, matformer, MegNet_kgcnn_v2.1.0, modnet_v0.1.10, modnet_v0.1.12, SchNet_kgcnn_v2.1.0, TPOT |

script), and a metadata.json file which plays a similar role as the info.json file used in MatBench submissions where the most relevant information for the environment is in the software_used field.

The reproducibility effort for JARVIS-Leaderboard models aimed not only to set up a valid environment in which to run the models but to do so using a purely functional Nix packaging ecosystem[20]. The functional packaging paradigm models the software environment as the output of a pure function whose inputs are the pure functions that similarly output the environment's direct dependencies[21]. This functional relation continues recursively such that an environment closure precisely specifies dependencies as a directed acyclic graph with nodes ranging from the model benchmark script, through the Python interpreter and modules, to low-level transitive dependencies such as the C standard library. The functional approach then makes full or partial updates to the environment straightforward, while only rebuilding portions of the dependency graph downstream of any changes. This is in contrast with the more common imperative model of instantiating software environments, where a series of steps are performed, each mutating the filesystem of some OS image to add, remove, or update packages to arrive at a given environment. Stronger reproducibility guarantees can then be made in the functional model without the dependence on the details of an entire OS image[22-24]. See Figure 5 for an example graph of the package dependencies which can readily be constructed from the Nix function describing an environment.

To narrow the initial scope of this work, we focus on packaging just the models which provide predictions for the exfoliation energy benchmark using Nix. This was chosen to be generally cheaper to test as the prediction output is a single scalar, and the training dataset is smaller than for other benchmarks. This allows efforts to be directed toward reproducing usable environments for the model to run rather than other factors. The general approach was to start by writing a Nix function for each model's environment containing only the software in JARVIS-Leaderboard entry. This function is modified by adding packages, constraining

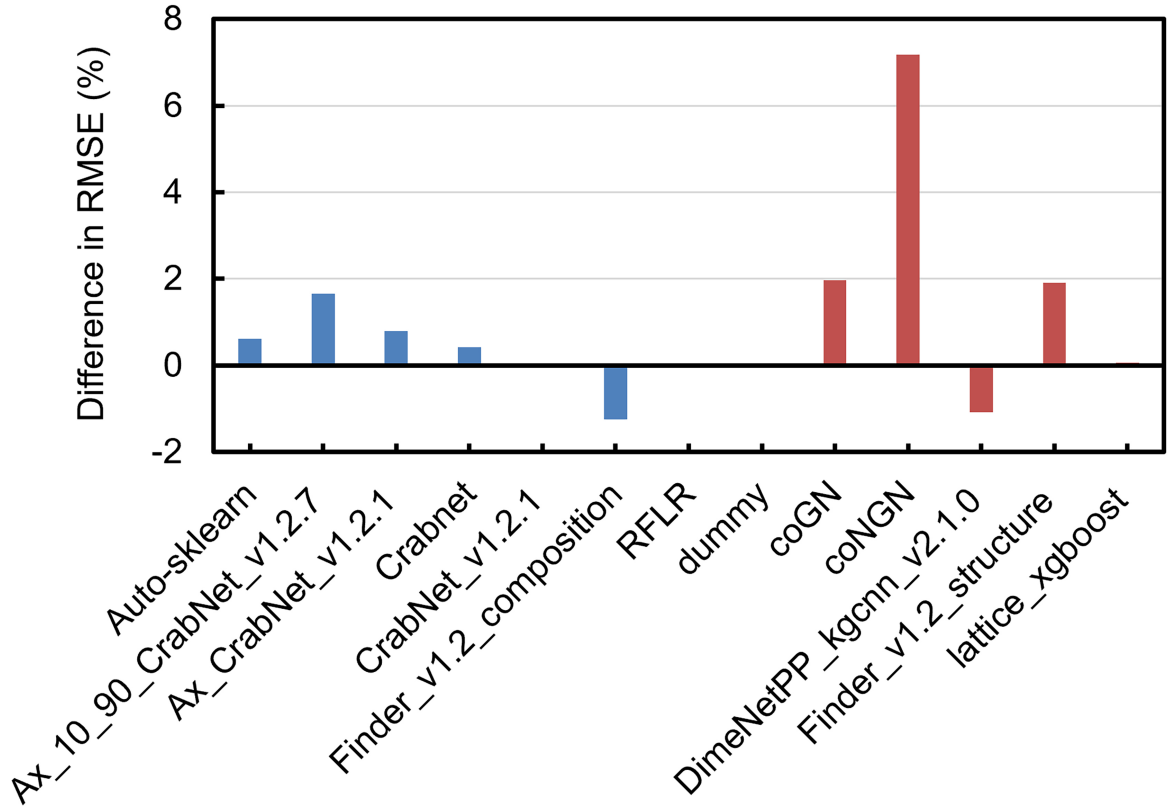



**Figure 4.** Percentage difference between the five-fold mean RMSE obtained from re-execution and that recorded in the original MatBench submission, calculated using Eq. (1). Positive values indicate that the re-executed RMSE was higher than the reported value. In contrast, negative values indicate that it was lower. Values of 0 indicate that the reported and re-executed five-fold mean RMSE values were identical within the numerical precision of the stored results. Blue and red bars represent composition-based and structure-based tasks, respectively. RMSE: Root mean square error; RFLR: Random Forest by Lorenz Romaner; coGN: Connectivity optimized Graph Network; coNGN: Connectivity optimized Nested Graph Network.

package versions, and/or adding patches to the source script until a trial training and inference run succeeds. Source scripts were also patched to enable Command-Line Interface (CLI) flags for running only the desired benchmark (some scripts ran more than just exfoliation energy) and running in a "test mode" to reduce the number of training iterations.

Nix packages for non-Python dependencies and the Python interpreter itself were obtained from the nixpkgs repository[25]. While nixpkgs also contains a number of Python packages, it does not contain all of the Python Package Index (PyPI) or some other packages that are more easily handled by Python-specific tooling. For Python-specific environments we prepared a pyproject.toml file[26] for each module compatible with the PDM[27] Python dependency management tool. This was then incorporated into the Nix function via dream2nix[28]. No attempt was made to constrain dependency versions to those used by the original authors. The JARVIS-Leaderboard project only required the software_used field of metadata.json to contain a string of comma-separated software projects, so such constraints were rarely provided. For models where version constraints were provided, they were utilized only on select packages when unconstrained dependency versions resulted in errors.

In addition to just identifying dependencies and versions, it also became necessary to patch the source files of some benchmark scripts. Some patches were needed due to API changes in dependencies (similar to those

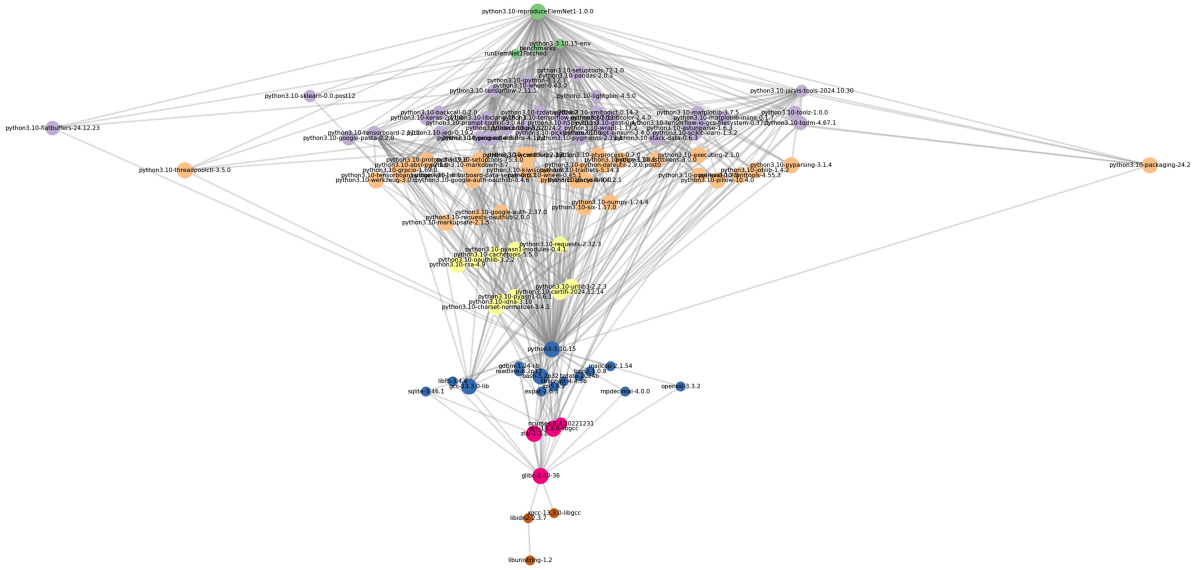


**Figure 5.** Runtime dependencies of the "Elemnet" model environment constructed from the Nix packaging done for JARVIS-Leaderboard. The environment is the node at the very top while deeper levels of transitive dependencies are shown lower in the graph. While package names and versions are present for those who wish to zoom in, the purpose of the image is to demonstrate the complexity of the environment needed for even a simple model, and the ability of the Nix approach to capture these requirements.

mentioned in Section "Reproducing the benchmark results"). Other patches were needed for Nix-specific reasons, mostly to avoid hardcoding filesystem paths for output data. In particular, mutating the source directory at runtime breaks the purely functional model. Patches were also added to create a uniform CLI interface to select the benchmark set to run and a-test flag which runs a computationally cheaper version of the model to test the environment (e.g., with fewer training iterations). Making edits across multiple contribution scripts from different authors to achieve similar behavior modification was facilitated by LLM models, though the patches were sufficiently contained in scope to be reviewed and judged to be very unlikely to have an impact on results.

Some models (namely, those from the kgcnn group) contained a conda environment.yaml file with specific versions; while not used directly (as conda was not used in our packaging approach), this was the most complete of the environments documented (indeed it seems it may be overcomplete, likely containing some packages which are not used). Most models had incomplete descriptions of the software used. In this case, Python reflection capabilities were used to obtain a list of required modules which could then be used to identify packages needed in the environment. One unfortunate aspect of the Python ecosystem is that module names and package names need not be the same; indeed, the appropriate "jarvis" module itself is provided by the package "jarvis-tools". Even with reflection tools, this task could only be partially automated. In addition to identifying the required packages, the package versions were also constrained where needed. Additional patching (not related to the Nix filepath constraints or added CLI flags) was then also applied where needed. Some remaining errors were a result of changes in the PyPI, such as the package "sklearn" being renamed to "scikit-learn". The matminer-lgbm model also had an error related to the fact that the JARVIS-Leaderboard repo had changed a directory name where benchmark data was stored and the script was not kept in sync with this change.

After these modifications, still 3 out of the 14 models require a more substantial effort to properly package their environments. The alignn model depends on the deep graph library (DGL) which is not present in the PyPI and requires additional packaging work to be amenable for use with the Nix approach. The provided

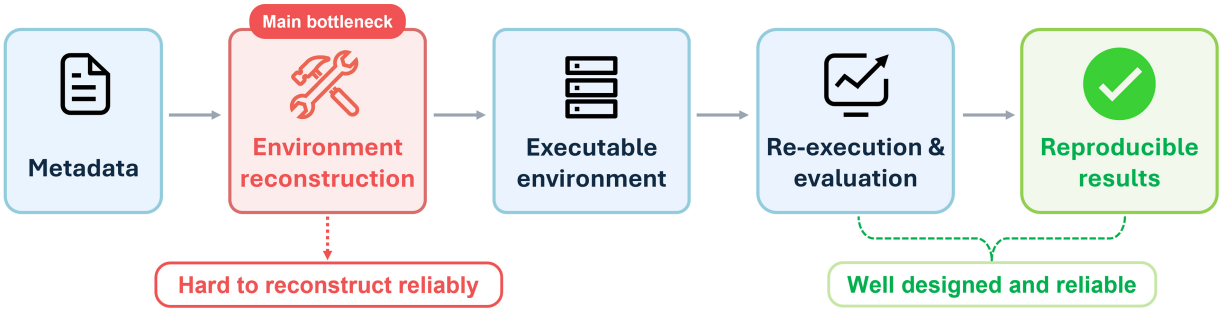



**Figure 6.** The real bottleneck in reproducing the results is the difficulty in reconstructing the Python environment used by the author.

scripts for the cgcnn model download data and code which are then executed all at runtime breaking the separation between environment instantiation and model training/inference execution, the untangling of which was not perused. Finally, the matformer model run script appears to activate a conda environment presumably present on the contributor's machine, but which is not present in the leaderboard contribution.

The central issue of insufficiently specified environments is just as present in JARVIS-Leaderboard as in MatBench and the Nix approach to defining and testing environments taken comes with a distinct set of tradeoffs. Writing Nix functions for model environments is likely less familiar to most materials data science researchers. However, this barrier can likely be lowered significantly by LLMs, especially since Nix environments are fully represented in text. The Nix approach enables sandboxed environments to be built and run locally while iterating to debug issues without needing to rebuild portions of the environment which remain unchanged between iterations. GPU acceleration is crucial to the field, but was not fully addressed at this stage of the work. This "hardware dependent software" has some necessary implications on reproducibility and requires some special treatment within the functional packing model. Multiple, evolving approaches currently exist within the ecosystem which will be utilized in subsequent work[29,30].

## Discussion

Based on the discussion above, we conclude that the main reproducibility challenges on the MatBench and JARVIS-Leaderboard platform stem from the difficulty of accurately reconstructing the Python environments originally used by the authors. Once a runnable environment can be successfully established, the reported results are generally highly reproducible [Figure 6]. The limitations of the currently provided metadata can be summarized as follows: (i) the list of required packages is sometimes incomplete, leading to import errors before execution can begin; (ii) specific package versions may exhibit dependency conflicts, whereas substituting alternative versions can render the code incompatible; and (iii) as the package ecosystem evolves, some previously specified dependencies may become unavailable, causing environment reconstruction to fail. Therefore, additional measures are needed to improve the success rate of environment reconstruction and the executability of the submitted code, thereby strengthening the reproducibility of benchmark results.

Based on these observations, we propose several recommendations for benchmark platform maintainers to improve the reproducibility of submitted algorithms as shown in Figure 7. These recommendations can be broadly divided into two categories: submission-time validation and preservation of validated environments. First, when a new algorithm is submitted to a benchmark platform, a continuous integration workflow could be used to automatically validate the corresponding Python environment. Submissions could be accepted only after they successfully pass environment reconstruction and import checks. To further strengthen this validation process, the workflow could include a lightweight smoke test that executes a minimal benchmark

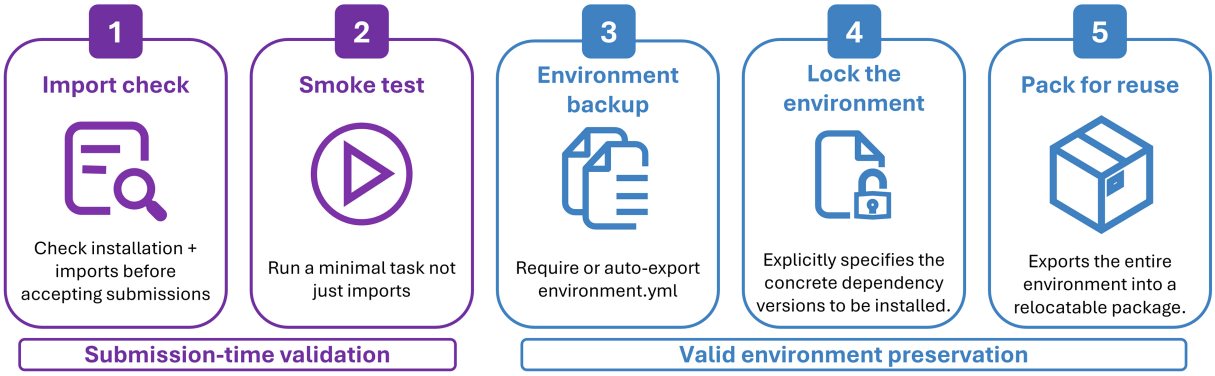



**Figure 7.** Platform-level recommendations for better environment reproducibility.

run within the same gated submission pipeline, ensuring that the submitted code is not only installable but also executable. Once a valid environment has been established, it should be preserved to support future re-execution. For example, authors or platform maintainers could export an environment backup file, such as environment.yml, which provides a user-friendly starting point for environment reconstruction. Third, as package ecosystems evolve, an environment backup file may become insufficiently robust for resolving the intended dependency set. In such cases, tools such as conda-lock can post-process environment.yml file by generating a fully resolved lockfile that explicitly specifies the concrete dependency versions to be installed. As a result, even if upstream package repositories change in the future, environment resolution remains unaffected. Fourth, in some cases, the target package is no longer available (e.g., matbench v0.1.0), making it impossible for even a lockfile to retrieve the required artifacts. In such situations, another practical option is to use a containerization tool (e.g., conda-pack) to package the entire environment into a relocatable archive that users can activate and run directly.

Notably, these improvements can be achieved without changing the current submission format. An environment backup file can be automatically exported, and the installed environment can be post-processed using package locker or containerization tools by configuring a continuous integration workflow. As a result, maintainers would not need to invest substantial additional manual effort for future submissions, because continuous integration workflows can perform validation checks while improving environment reproducibility. A working example of the code and end-to-end execution workflow has been uploaded to our repository.

Our recommendations primarily emphasize the reconstruction of executable software environments. However, the reproducibility of benchmark results may also be affected by differences in operating systems, hardware driver versions, hardware configurations, and nondeterministic behavior in numerical computation. Because it is impractical to require all training runs to be performed on identical machines, several additional strategies could be considered to improve the executability, accessibility, and robustness of algorithms hosted on benchmark platforms. For instance, environment reconstruction and execution workflows could be provided through a hosted cloud service, allowing users to test both environment configuration and code execution on a widely accessible managed cloud environment. Alternatively, callable model interfaces, associated metadata, and execution specifications could be published through Machine Learning Operations (MLOps) platforms such as Garden-AI, which may improve model discoverability and simplify remote invocation, particularly for standardized machine-learning workflows. Another option is to distribute each algorithm through a container (e.g., as a Docker image or Dockerfile), which can preserve much of the user-space software stack and reduce dependency drift caused by changes in the Python package ecosystem. Nevertheless, containerized execution does not eliminate all sources of variation, as host drivers,

**Table 5. Comparison of common strategies for preserving and providing executable machine-learning environments**

| Approach | Main capability | Remaining limitations | Maintenance burden |
|---|---|---|---|
| requirements.txt or environment.yml | Records direct dependencies and provides an editable starting point for environment reconstruction | Does not necessarily preserve exact transitive dependencies or package builds; reconstruction may fail if packages or repositories change | Low |
| conda-lock | Generates a fully resolved lockfile containing exact package versions and builds for specified platforms | Still depends on the continued availability of the referenced packages and channels; does not preserve the operating system, drivers, or hardware | Low to moderate |
| conda-pack | Archives an already installed Conda environment for direct relocation and reuse | Portability may be limited across operating systems, architectures, system libraries, and hardware configurations | Moderate |
| Docker or Apptainer image | Preserves most of the user space software stack and reduces dependency drift | Does not preserve the host kernel, GPU drivers, hardware, or external services; image registries and security updates require maintenance | Moderate to high |
| Nix | Describes the environment through a declarative dependency graph and supports traceable, reproducible builds | Requires specialized packaging knowledge; GPU support, proprietary software, and external resources may require additional configuration | Moderate to high |
| Hosted cloud or notebook service | Provides an accessible and partially standardized execution platform | Base images, available hardware, service policies, and package versions may change over time; continued availability depends on the service provider | Continuous |
| Hosted MLOps or model-serving platform | Provides standardized model interfaces, remote execution, metadata management, and improved discoverability | May restrict modification of the underlying environment and depends on long-term platform hosting, infrastructure, and service compatibility | High |

GPU: Graphics Processing Unit; MLOps: Machine Learning Operations.

GPU hardware, system architecture, and registry maintenance may still affect execution. These approaches can improve different aspects of reproducibility and usability, but they also impose additional burdens on method developers, algorithm contributors, and platform maintainers in terms of packaging, validation, infrastructure, storage, and long-term maintenance. These approaches address different aspects of environment preservation and workflow accessibility, and no single strategy eliminates all sources of computational variation. The trade-off between these operational costs and the resulting gains in reproducibility should be carefully considered. Their respective capabilities, limitations, and maintenance requirements are summarized in Table 5.

## CONCLUSION

In this work, we assessed the practical reproducibility of 28 MatBench submissions using the publicly available source code and dependency metadata. Only 2 of the 28 submissions could be installed from the original specifications and pass the import-based environment validation without modification. Using a three-stage workflow consisting of strict environment reconstruction, automated dependency remediation, and targeted human-supervised intervention, we established runnable environments for 26 submissions. These 26 submissions subsequently entered the code re-execution stage, of which 13 completed the selected representative benchmark task and generated benchmark outputs. For these 13 cases, the regenerated five-fold mean RMSE values obtained for the selected representative tasks were generally close to the values reported in the original submissions. Similar issues have been found on JARVIS-Leaderboard, a recent materials benchmark platform. Based on this analysis, we recommend that benchmark platforms adopt continuous integration-based validation of submissions and, within the workflow, automatically generate an environment backup file for each submission while employing stronger environment-preservation mechanisms and tools. Together, these measures can reduce the maintenance burden on users and maintainers while improving long-term reusability and trust.

## DECLARATIONS

### Acknowledgments

For computer time, this research used Shaheen III managed by the Supercomputing Core Laboratory at King

Abdullah University of Science and Technology (KAUST) in Thuwal, Saudi Arabia.

### Authors' contributions

Conceptualization: Lyu, B.; Bonini, J.; Wines, D.; Li, K.
Methodology: Lyu, B.; Bonini, J.; Wines, D.; Li, K.
Investigation and data curation: Lyu, B.; Bonini, J.; Zhang, M.
Formal analysis and visualization: Lyu, B.; Bonini, J.
Writing - original draft: Lyu, B.; Bonini, J.
Writing - review and editing: Lyu, B.; Bonini, J.; Zhang, M.; Sadeghi, A.; Jaberi, A.; Hattrick-Simpers, J.; Choudhary, K.; Wines, D.; Li, K.
Supervision: Li, K.

### Availability of data and materials

The GitHub Action workflows for MatBench are available at https://github.com/BohuiLyu/matbench_reproduce.git. The Nix packaging for JARVIS-Leaderboard is available at https://github.com/usnistgov/reproducible-models.

### AI and AI-assisted tools statement

During the preparation of this manuscript, the AI tool GPT-5.4 Thinking (version 5.4, released 2026-03-05) was accessed through the ChatGPT web interface and used as a human-supervised troubleshooting assistant for software-environment reconstruction. Package requirements, installation commands, and associated error messages were provided to the model to obtain possible remediation suggestions. The model did not execute commands or autonomously modify any code or environment. The tool did not influence the study design, data collection, analysis, interpretation, or the scientific content of the work. All authors take full responsibility for the accuracy, integrity, and final content of the manuscript.

### Financial support and sponsorship

Li, K. acknowledges funding support from King Abdullah University of Science and Technology (KAUST). NIST work was funded solely by the United States Government.

Certain commercial equipment, instruments, software, or materials are identified in this paper in order to specify the experimental procedure adequately. Such identifications are not intended to imply recommendation or endorsement by NIST, nor it is intended to imply that the materials or equipment identified are necessarily the best available for the purpose.

### Conflicts of interest

All authors declared that there are no conflicts of interest.

### Ethical approval and consent to participate

Not applicable.

### Consent for publication

Not applicable.



### Supplementary Materials

Supplementary Materials

## REFERENCES

1. Lejaeghere, K.; Bihlmayer, G.; Björkman, T.; et al. Reproducibility in density functional theory calculations of solids. *Science* **2016**, *351*, aad3000. DOI PubMed

2. Bosoni, E.; Beal, L.; Bercx, M.; et al. How to verify the precision of density-functional-theory implementations via reproducible and universal workflows. *Nat. Rev. Phys.* **2023**, *6*, 45-58. DOI
3. Wilkinson, M. D.; Dumontier, M.; Aalbersberg, I. J.; et al. The FAIR Guiding Principles for scientific data management and stewardship. *Sci. Data.* **2016**, *3*, 160018. DOI PubMed PMC
4. Boyd, P. G.; Chidambaram, A.; García-Díez, E.; et al. Data-driven design of metal-organic frameworks for wet flue gas $CO_2$ capture. *Nature* **2019**, *576*, 253-6. DOI PubMed
5. Wang, M.; Jiang, J. Accelerating discovery of polyimides with intrinsic microporosity for membrane-based gas separation: synergizing physics-informed performance metrics and active learning. *Adv. Funct. Mater.* **2024**, *34*, 2314683. DOI
6. Tang, H.; Jiang, J. In silico screening and design strategies of ethane-selective metal-organic frameworks for ethane/ethylene separation. *AIChE. J.* **2020**, *67*, e17025. DOI
7. Wang, T.; He, X.; Li, M.; et al. Ab initio characterization of protein molecular dynamics with AI²BMD. *Nature* **2024**, *635*, 1019-27. DOI PubMed PMC
8. Unke, O. T.; Chmiela, S.; Sauceda, H. E.; et al. Machine learning force fields. *Chem. Rev.* **2021**, *121*, 10142-86. DOI PubMed PMC
9. Chung, Y. G.; Gómez-Gualdrón, D. A.; Li, P.; et al. In silico discovery of metal-organic frameworks for precombustion $CO_2$ capture using a genetic algorithm. *Sci. Adv.* **2016**, *2*, e1600909. DOI PubMed PMC
10. Nandy, A.; Yue, S.; Oh, C.; et al. A database of ultrastable MOFs reassembled from stable fragments with machine learning models. *Matter* **2023**, *6*, 1585-603. DOI
11. Terrones, G. G.; Huang, S. P.; Rivera, M. P.; Yue, S.; Hernandez, A.; Kulik, H. J. Metal-organic framework stability in water and harsh environments from data-driven models trained on the diverse WS24 data set. *J. Am. Chem. Soc.* **2024**, *146*, 20333-48. DOI PubMed
12. Stach, E.; Decost, B.; Kusne, A. G.; et al. Autonomous experimentation systems for materials development: a community perspective. *Matter* **2021**, *4*, 2702-26. DOI
13. Dai, T.; Vijayakrishnan, S.; Szczypiński, F. T.; et al. Autonomous mobile robots for exploratory synthetic chemistry. *Nature* **2024**, *635*, 890-7. DOI PubMed PMC
14. Krizhevsky, A.; Sutskever, I.; Hinton, G. E. ImageNet classification with deep convolutional neural networks. *Commun. ACM.* **2017**, *60*, 84-90. DOI
15. Wang, A.; Singh, A.; Michael, J.; Hill, F.; Levy, O.; Bowman, S. R. GLUE: a multi-task benchmark and analysis platform for natural language understanding. In *Proceedings of the 2018 EMNLP Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*; Linzen, T.; Chrupała, G.; Alishahi, A., Eds.; Association for Computational Linguistics: Brussels, Belgium, 2018; pp 353-5. DOI
16. Reddi, V. J.; Cheng, C. Kanter, D., et al. MLPerf inference benchmark. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*; 2020; pp 446-59. DOI
17. Dunn, A.; Wang, Q.; Ganose, A.; Dopp, D.; Jain, A. Benchmarking materials property prediction methods: the Matbench test set and Automatminer reference algorithm. *npj Comput. Mater.* **2020**, *6*, 138. DOI
18. Riebesell, J.; Goodall, R. E. A.; Benner, P.; et al. A framework to evaluate machine learning crystal stability predictions. *Nat. Mach. Intell.* **2025**, *7*, 836-47. DOI
19. Choudhary, K.; Wines, D.; Li, K.; et al. JARVIS-Leaderboard: a large scale benchmark of materials design methods. *npj. Comput. Mater.* **2024**, *10*, 93. DOI
20. NixOS. Declarative builds and deployments. https://nixos.org/. (accessed 2026-09-11).
21. Dolstra, E. The purely functional software deployment model. Ph.D. Thesis, Utrecht, The Netherlands: Utrecht University, 2006. https://edolstra.github.io/pubs/phd-thesis.pdf. (accessed 2026-09-11).
22. Kowalewski, M.; Seeber, P. Sustainable packaging of quantum chemistry software with the Nix package manager. *Int. J. Quantum. Chem.* **2022**, *122*, e26872. DOI
23. Hausch, M.; Hauser, S.; Uekermann, B. Improving reproducibility of scientific software using Nix/NixOS: a case study on the preCICE ecosystem. *Electron. Commun. EASST.* **2025**, *83*. DOI
24. Bandt, C. Assessment of reproducibility in Nix. https://doi.org/10.5281/zenodo.17372811. (accessed 2026-09-11).
25. *Nixpkgs*. https://github.com/NixOS/nixpkgs. (accessed 2026-09-11).
26. Cannon, B.; Ingram, D. Ganssle, P., et al. PEP 621 - Storing project metadata in pyproject.toml. https://peps.python.org/pep-0621/. (accessed 2026-09-11).
27. *PDM - A Modern Python Package and Dependency Manager*. https://pdm-project.org/. (accessed 2026-09-11).
28. *DREAM2NIX - Automate Reproducible Packaging for Various Language Ecosystems*. https://github.com/nix-community/dream2nix. (accessed 2026-09-11).
29. *NixGL - A Wrapper Tool for Nix OpenGL Application*. https://github.com/nix-community/nixGL. (accessed 2026-09-11).

30. *Nix System Graphics - Run Graphics Accelerated Nix Applications on Any Linux Distribution*. https://github.com/soupglasses/nix-system-graphics. (accessed 2026-09-11).