ReproScore: Separating Readiness from Outcome in Research Software Reproducibility Assessment

Sheeba Samuel[1][0000−0002−7981−8504], Daniel Mietchen[2][0000−0001−9488−1870], Jungsan Kim[1], Waqas Ahmed[3][0000−0002−9354−3527], and Martin Gaedke[1][0000−0002−6729−2912]

[1] Chemnitz University of Technology sheeba.samuel, martin.gaedke@informatik.tu-chemnitz.de
[2] FIZ Karlsruhe — Leibniz Institute for Information Infrastructure, Germany daniel.mietchen@fiz-karlsruhe.de
[3] Friedrich Schiller University, Jena, Germany ahmed.waqas@uni-jena.de

Abstract. Digital libraries curate millions of research software artefacts yet lack scalable infrastructure for assessing whether those artefacts remain executable. Existing automated assessment tools treat static repository completeness—what a repository *contains*—as a proxy for execution success—whether it *runs*. We term this the *readiness–outcome conflation* and present **ReproScore**, a two-tier framework that explicitly separates reproducibility readiness (RRS) from reproducibility outcome (ROS), combining them into a coverage-adaptive Composite Score (RCS). RRS comprises 26 sub-metrics across five categories; ROS provides execution-based probes when sandbox infrastructure is available, and a community rubric externalises weighting priorities as versioned YAML profiles. Evaluated on 423 GitHub repositories from a large-scale ground-truth corpus spanning five failure modes, two complementary findings emerge: the environment category of RRS strongly discriminates failure mode, confirming static signals capture meaningful structural differences; yet RRS exhibits near-zero correlationwith binary execution success, which empirically quantifies the readiness–outcome gap at repository scale, as measured on a class-stratified sample (ca. 85 repositories per failure mode). Together, these findings validate the architectural separation as both necessary and non-trivial, positioning ReproScore as scalable infrastructure for reproducibility-aware curation in digital library workflows.

Keywords: Research software reproducibility · Digital libraries · Repository quality assessment · Open science · FAIR software

1 Introduction

Digital libraries and research repositories like Zenodo, Software Heritage, Figshare, and institutional data archives now store research software at a scale that challenges manual curation. The practical problem facing data librarians is concrete: when a deposit arrives, how can they efficiently assess whether it is executable, identify quality deficiencies, and prioritise remediation? This challenge sits within the Ingest and Archival Storage entities of the OAIS Reference

Model [21] and the appraisal stage of the DCC Curation Lifecycle [18], yet neither framework supplies computable metrics for research software executability.

The scale of the underlying reproducibility problem is well documented: fewer than one in four Jupyter notebooks can be re-executed without error [25], only 26% of computational articles in *Science* were reproducible under varying effort [35], and 90% of surveyed researchers acknowledged a reproducibility crisis [3]. At repository intake scale, these rates imply that the majority of deposits labelled as computational research cannot be re-executed by a curator who tries. This is a library infrastructure problem, not merely a research culture problem.

Existing automated tools address this gap incompletely and from opposing directions. Static analysis tools—SciScore [4], SOMEF [22], and README-based classifiers [2]—assess documentation completeness and metadata coverage: what a deposit *contains*. Execution-based approaches—CODECHECK [23], Whole Tale [6], and repo2docker [12]—assess whether code *runs*, but require expert human involvement or sandboxed infrastructure and cannot operate at collection intake scale. Automated notebook analysis [25,28] has characterised execution failure patterns in large corpora but stops short of providing a scoring framework for curatorial decision-making. The critical gap is that *static artefact quality and execution outcome are genuinely different quantities*: a repository with a pinned `requirements.txt`, a Zenodo data pointer, and a structured README is an excellent static deposit that may nonetheless fail at install time due to issues like a transitive dependency conflict—a failure no static tool can observe.

We call this the *readiness–outcome conflation* and present **ReproScore**, a repository curation framework that addresses it within digital library workflows. ReproScore functions as a *failure-mode characterisation* instrument: it tells a curator *what kind* of reproducibility problem a repository has (missing environment specification, inaccessible data, portability defects, absent determinism signals), not merely whether it will succeed. This diagnostic framing, combined with a transparent rubric governance mechanism, constitutes the primary contribution.

Contributions. This paper makes the following contributions: (1) A **conceptual and computational separation** between readiness (static, available at intake) and execution outcome (optional), realised as three formally distinct quantities: RRS (readiness, always computable), ROS (outcome, execution-dependent), and RCS (a coverage-weighted composite of the two). We term this pairing of an always-available static tier and an optional execution tier a *two-tier architecture*; RRS is the primary empirical contribution, while ROS and RCS constitute a partially validated architectural extension. (2) A **five-category, 26-sub-metric readiness model** (RRS) with expert-informed weights, a non-linear gate encoding curation policy, and file-level evidence provenance. (3) A **community rubric mechanism** externalising assessment priorities as a versioned YAML profile. (4) An **empirical evaluation on 423 GitHub repositories** from a large-scale ground-truth corpus [28] showing static readiness strongly discriminates failure mode ($H = 96.89$, $p < 0.001$).

2 Background and Related Work

Reproducibility in digital library curation. The OAIS Reference Model [21] identifies Ingest and Archival Storage as functional entities where quality assessment occurs, but specifies no executable quality metrics for research software. The DCC Curation Lifecycle [18] frames quality appraisal as a continuous obligation across the deposit lifecycle. Software Heritage [11] preserves source code at collection scale; Zenodo, Figshare, and institutional repositories accept heterogeneous deposits including Jupyter notebooks, R Markdown scripts, and containerised workflows. None of these platforms provides systematic, automated assessment of executability at intake—the gap thst ReproScore targets.

FAIR for software. The FAIR Guiding Principles [41] and FAIR4RS [9] articulate reusability desiderata but supply no computable executability scores. Tools such as `howfairis` [33], FAIRsFAIR [1], and FAIR-Checker [13] assess FAIR compliance at the metadata level. While CodeMeta [20] and RO-Crate [32] provide description vocabularies, neither links metadata to execution potential.

Static quality analysis. SciScore [4] analyses biomedical paper methods sections for reporting completeness, operating on paper text rather than code. Better Code Hub and SonarQube [7] assess maintainability and security without targeting reproducibility. Akdeniz et al. [2] used transformer models to assess section completeness of README files and generate reproducibility scores structurally identical to the conflation they aim to address. Unlike general-purpose software quality models (ISO/IEC 25010; CHAOSS [15]), ReproScore operationalises executability-specific criteria essential for reproducibility assessment.

Execution-based evaluation. CODECHECK [23] provides an expert peer-review workflow for individual submissions. Trisovic et al. [38] execute Harvard Dataverse R scripts at scale, finding version conflicts as the dominant failure mode—motivating ReproScore's emphasis on environment specification. Costa et al. [10] compare eight reproduction tools, finding that design choices critically shape execution outcomes. Recent LLM-agent benchmarks—PaperBench [34], CORE-Bench [31], SUPER [5], REPRO-Bench [19]—evaluate automated reproduction on individual papers. These execution-based agents and static readiness tools are complementary: agents require functional code to orchestrate; static readiness scoring provides a pre-execution triage layer that identifies *why* code is unlikely to function before any execution resource is committed—and reduces the environmental cost of curation by screening out unexecutable deposits before compute-intensive container builds and sandbox runs are attempted. A consistent pattern is that static documentation quality and execution success are systematically conflated [8,25,38,39]. ReproScore addresses this directly.

3 The ReproScore Scoring Architecture

Figure 1 illustrates the ReproScore two-tier pipeline.

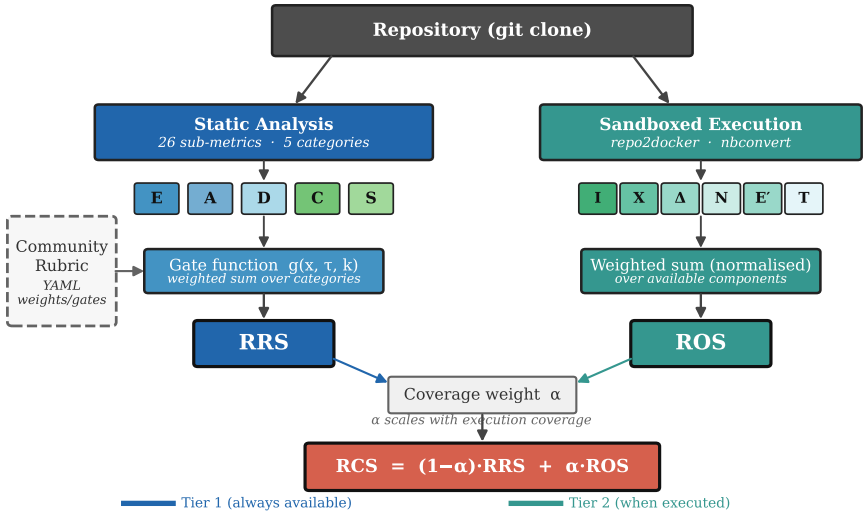

Fig. 1. ReproScore two-tier architecture. Tier 1 (static analysis, always available at intake) computes RRS (cf. Section 3.1) from 26 sub-metrics (cf. Table 2) in 5 categories (cf. Table 1) via a community-configurable rubric (cf. Section 3.4). Tier 2 (sandboxed execution, optional) computes ROS (cf. Section 3.2) from up to six execution probes. RCS (cf. Section 3.3) blends both tiers via a coverage weight α that scales with collected execution evidence, allowing the composite metric to degrade gracefully from RCS = RRS (no execution) to RCS $\approx$ ROS (full execution coverage).

3.1 Reproducibility Readiness Score (RRS)

Category structure and weights. RRS is computed over five top-level categories, each capturing a distinct dimension of reproducibility readiness. Table 1 lists the categories, default weights, gate parameters, and sub-metric counts.

Both inter-category weights w_i and within-category weights w_j are expert-informed default choices, not data-optimised parameters, and exposed through the community rubric (Section 3.4). Environment specification receives $w_E = 0.30$ because missing or conflicting dependencies are the dominant execution failure mode [38,28]. Data accessibility receives $w_A = 0.25$ because inaccessible or unresolvable input data is the second most common barrier to re-execution [35]: a pipeline that cannot obtain its inputs cannot be executed regardless of environment quality. Documentation receives $w_D = 0.20$ because the absence of execution-relevant instructions—entry points, install steps, expected outputs—forces a third party to reverse-engineer the workflow before any execution attempt can be made, compounding other failure modes. Code portability receives $w_C = 0.15$ because source-code properties that cause failure when code is moved across machines—absolute paths, undeclared imports, hardcoded credentials—are detectable statically and directly predictive of execution failure, independent

Table 1. RRS categories, default weights, gate parameters, and sub-metric counts. Weights sum to 1.0. Core categories (E, A) use steeper gates; quality categories (D, C, S) use lenient gates.

Symbol	Category and guiding question	Weight w_i	τ_i	k_i	Sub-metrics
E	**E**nvironment specification *Can the computational environment be reconstructed?*	0.30	40	1.5	4
A	Data **a**ccessibility *Can the required data be obtained and used?*	0.25	30	1.5	4
D	**D**ocumentation *Can a third party understand how to execute this?*	0.20	20	1.2	7
C	**C**ode portability *Will the code run on a machine other than the author's?*	0.15	25	1.2	4
S	Reproducibility **s**ignals *Will repeated executions produce the same verifiable result?*	0.10	30	1.2	7
	Total	1.00			26 atomic sub-metrics

of environment quality [38]. Reproducibility signals receive $w_S = 0.10$ because workflow practices and determinism commitments—seed management, execution order, output declarations—are necessary for result verification but presuppose that execution itself succeeds; they are therefore downstream of the four primary categories.

Gate function. A power-law gate function penalises sub-threshold failures non-linearly: $E = 2$ (no specification) should not score comparably to $E = 35$ (partial specification) under a linear scale. The gate encodes a curation policy claim rather than a performance mechanism. ReproScore defines $g : [0, 100] \times \mathbb{R}_{>0} \times \mathbb{R}_{>1} \to [0, 1]$ as:

$$g(x,\, \tau,\, k) \;=\; \begin{cases} \dfrac{x}{100} & \text{if } x \geq \tau \\[2ex] \left(\dfrac{x}{\tau}\right)^{k} \cdot \dfrac{\tau}{100} & \text{if } x < \tau \end{cases} \tag{1}$$

where $x \in [0, 100]$ is the raw sub-score, τ is the threshold below which compression begins, and $k > 1$ controls steepness. Above τ, g is linear. Below τ, g applies a power-law contraction: at $x = \tau/2$, the contribution is $2^{-k} \cdot \tau/100$, i.e. less than half the linear value. Core categories use $k = 1.5$ (steep); quality categories use $k = 1.2$ (lenient). Robustness analysis (Section 4.3) shows the gate has negligible predictive impact (AUC range ≈ 0.008); its value is normative.

RRS formula. The Reproducibility Readiness Score is computed as:

$$\text{RRS} \;=\; 100 \cdot \sum_{i \in \{E,A,D,C,S\}} w_i \cdot g(x_i,\, \tau_i,\, k_i) \;-\; P_{\text{hard}}(E, A) \;-\; P_{\text{seed}} \tag{2}$$

where $x_i \in [0, 100]$ is the raw category score, w_i and (τ_i, k_i) are the weights and gate parameters from Table 1, and the penalty terms are:

$$P_{\text{hard}}(E, A) \;=\; 20 \cdot \mathbf{1}[E < 10] \;+\; 15 \cdot \mathbf{1}[A < 10] \tag{3}$$

$$P_{\text{seed}} \;=\; 10 \cdot \mathbf{1}[\sigma < 50] \tag{4}$$

where σ is the seed management score (sub-metric of S). P_{hard} penalises complete absence of environment specification or data artefacts—conditions under which reproduction is virtually impossible. P_{seed} penalises repositories with stochastic operations but no seed-setting calls. Penalty magnitudes are calibrated so that each penalty approximates the maximum weight contribution of the penalised category: -20 pts for $E < 10$ corresponds to $w_E \times 100 \times 0.67$; -15 pts for $A < 10$ to $w_A \times 100 \times 0.60$. RRS $\in [0, 100]$ after clamping.

Sub-metric definitions. Table 2 presents the full 26-sub-metric taxonomy. Each sub-metric is computed by deterministic, rule-based analysis of the locally cloned repository—no execution required. Sub-metrics are grounded in execution failure patterns from large-scale reproduction studies [25,28,38], FAIR4RS criteria [9], and static signals from the FAIR Jupyter knowledge graph [29] (e.g. `notebook_exec_order`, `markdown_code_ratio`).

3.2 Reproducibility Outcome Score (ROS)

When a repository is executed in a sandboxed environment, ReproScore computes an execution-based outcome score comprising six components: install success I (0.30), execution success X (0.25), output determinism Δ (0.20), notebook execution rate N (0.10), import success rate E' (0.10, directly measuring dependency resolution), and test pass rate T (0.05). All components are optional; normalisation adjusts for missing components via a weighted sum over available probes only:

$$\text{ROS} \;=\; \frac{\sum_j v_j \cdot y_j \cdot \mathbf{1}[y_j \text{ available}]}{\sum_j v_j \cdot \mathbf{1}[y_j \text{ available}]} \tag{5}$$

where $y_j \in [0, 100]$ is the score for ROS component j and v_j is its weight. When no components are available, ROS is undefined ($\perp$) and the system falls back to RCS = RRS.

I and E' are *causally independent*: I tests package-manager resolution; E' tests runtime import success—both can diverge (confirmed empirically in Section 4.5). X is binary (overall outcome); N is continuous (fraction of notebooks completing without error). Δ compares output cells across two sandboxed runs via nbdime (exact match for non-numeric; tolerance 10^{-6} for numeric).

Table 2. ReproScore 26 atomic sub-metrics with within-category weights w_j (sum to 1.0 within each category). Type: Bin = binary (0/100); Cont = continuous; Tier = tiered discrete levels. Full measurement heuristics are available in the source repository.

Cat.	Sub-metric	w_j	Rationale	Type
E	Dep. pinning	0.25	Lockfile > exact pins > partial > absent; tiered by reproducibility guarantee	Tier
	Container spec	0.30	Container file presence and quality; pinned base + RUN is strongest	Tier
	Env. bootstrap	0.25	One-command environment creation (install script, Makefile target)	Bin
	Runtime version	0.20	Explicit Python/R version declaration (.python-version, runtime.txt, etc.)	Bin
A	Data description	0.20	Dedicated data documentation; tiered by word count / section richness	Tier
	Data pointer	0.30	Tiered by archival permanence: DOI > institutional > platform URL > local file	Tier
	Workflow orch.	0.20	End-to-end workflow tool (Snakemake, DVC, Nextflow) > Makefile > pipeline script	Tier
	Data acquisition	0.30	Automated download present (DVC tracking, wget/curl/API calls to data archives)	Bin
D	Doc. structure	0.25	Fraction of 4 execution-relevant README sections (install, run, expected output, requirements)	Cont
	Install instructions	0.20	Completeness of install guidance; one-command > multi-step > vague	Tier
	Usage examples	0.20	Runnable command in code fence > any code block > examples directory	Tier
	Inline explanation	0.15	Notebook md/code ratio and script comment density; both target ≥ 0.5 / $\geq 20\%$	Cont
	Entry point	0.10	Clear first command to execute (run.sh, main.py, Makefile run target)	Bin
	Docstring coverage	0.05	Inline docstrings on public functions/classes	Cont
	Reuse metadata	0.05	LICENSE + CITATION.cff + codemeta.json; tiered by count (transparency, not execution)	Tier
C	No absolute paths	0.40	Machine-specific paths (/home/user/, C:\Users\) absent from source files & notebooks	Cont
	Import resolvability	0.35	Third-party imports cross-referenced against declared dependencies	Cont
	No hardcoded creds	0.15	Credential assignment patterns (API keys, tokens) absent from source	Cont
	No silent failures	0.10	Bare `except:pass` (hides execution errors) absent from source	Cont
S	Seed management	0.30	$\lvert\mathcal{F}_{\text{seed}}\rvert/\lvert\mathcal{F}_{\text{rand}}\rvert$: stochastic files with seed-setting calls	Cont
	Notebook exec. order	0.20	Notebooks have monotonically increasing execution counts (run top-to-bottom)	Cont
	Test file presence	0.18	$\min(\lvert\mathcal{T}\rvert/2, 1)$; test suite existence	Cont
	Expected outputs	0.12	Reference output directory or committed figures present	Tier
	CI presence	0.10	Any CI configuration file present	Bin
	Config externalised	0.06	Experimental parameters in config files or CLI args (not hardcoded)	Tier
	Hardware requirements	0.04	If GPU packages detected: CUDA/hardware requirements declared; otherwise N/A	Bin

3.3 Composite Score and Coverage Weighting (RCS)

The coverage weight α measures what fraction of maximum execution evidence has been collected, capped at $\alpha_{\max} = 0.70$:

$$\alpha \;=\; \min\left(\sum_{j} v_j \cdot \mathbf{1}[y_j \text{ available}],\;\; 1\right) \cdot \alpha_{\max} \tag{6}$$

with floor $\alpha \geq \alpha_{\min} = 0.10$ whenever any ROS component is available. The ceiling $\alpha_{\max} = 0.70$ encodes the principle that static readiness is never entirely dominated by execution outcomes: even perfect execution does not measure data documentation, README completeness, or portability. The composite score is:

$$\text{RCS} \;=\; \begin{cases} \text{RRS} & \text{if ROS} = \perp \\ (1-\alpha)\cdot \text{RRS} \;+\; \alpha \cdot \text{ROS} & \text{otherwise} \end{cases} \tag{7}$$

For partial empirical validation of the RCS blending formula, see Section 4.5.

3.4 Community Rubric Mechanism

Assessment priorities differ across communities. A bioinformatics data archive may weigh data accessibility more heavily than CI configuration; a software-centric repository may prioritise portability. ReproScore externalises these priorities through a *community rubric*: a named, versioned YAML file overriding default category weights w_i and gate thresholds. The `RubricEngine` validates $\sum w_i = 1.0 \pm 0.01$ and recomputes the score using the community-specified profile. An illustrative FAIR-aligned bioinformatics override could look as follows:

```
name: bioinformatics-v1
version: "1.0"
categories:
  E: {weight: 0.35, tau: 40, k: 1.5}
  A: {weight: 0.40, tau: 30, k: 1.5}  # FAIR data priority
  D: {weight: 0.10, tau: 20, k: 1.2}
  C: {weight: 0.05, tau: 25, k: 1.2}
  S: {weight: 0.10, tau: 30, k: 1.2}
```

A platform assigning $w_A = 0.40$ expresses a versioned, auditable curation policy, in contrast to scoring heuristics embedded opaquely in source code. Rubric files are portable across institutions: archives can publish their weighting rationale as a citable artefact, enabling policy comparison and reuse.

Rubric governance and federation. Community rubric files function as governance artefacts, not merely configuration. An institution publishing its rubric declares a curation policy independently citable by peer archives, enabling cross-institutional score comparability wherever the same rubric version was applied. Archives can federate: a consortium of biomedical repositories might adopt a

shared baseline rubric, then document domain-specific overrides with additional rationale. When thresholds or weights are revised, affected scores can be recomputed from stored records without re-cloning, and the version history makes policy evolution auditable. The rubric schema is designed for serialisation as a CodeMeta [20] property, RO-Crate [32] contextual entity or nanopublication [16], enabling score-with-policy packaging within standard research object formats.

4 Evaluation

4.1 Corpus and Protocol

We evaluate the RRS scoring model on a stratified sample of Python repositories from a large-scale execution ground-truth corpus [27,28], which records the Python traceback of each notebook's first failing cell (or NULL for success). Repository-level failure modes aggregate notebook records via priority ordering (install errors $\succ$ import errors $\succ$ file-not-found errors $\succ$ code errors $\succ$ success); a repository is labelled *success* only when all its notebooks yield NULL. We stratify the computational results by failure mode, targeting **86 repositories per class**: *success* (all execution failure reasons = NULL), *install_dep* (`<Install Dependency Error>`), *missing_module* (`ModuleNotFoundError`/`ImportError`) and *missing_data* (network errors/`FileNotFoundError`), with the *code_error* class (`TypeError`/`NameError`/`SyntaxError` and related runtime exceptions) completing the set. Of 430 sampled repositories, 423 were scored (7 excluded due to clone or computation errors), yielding 84–85 per class. Each repository was shallow-cloned and scored by static RRS analysis only; no execution was performed. Per-repository provenance JSON and cloned SHA are archived with the dataset [30]. The sample is restricted to Python Jupyter notebook repositories to match the execution environment of the original dataset. The five-class failure mode is the primary ground truth for failure-mode discriminability (Kruskal-Wallis H, pairwise Cohen's d); binary success/failure is a secondary non-collapse check only. All statistical tests were computed using SciPy [40] (Python 3).

4.2 Failure-Mode Characterisation

Evaluation goals. The primary criterion is whether each category structurally separates repositories by failure type—a curator needs to know *what kind* of problem a repository has, not its failure probability. AUC-ROC is used solely as a non-collapse probe in Section 4.3. Table 3 reports category-level statistics.

Finding 1: E discriminates failure mode, not failure probability. The environment category achieves the highest Kruskal-Wallis H ($H = 96.89$, $p < 0.001$) yet near-zero binary correlation ($r_{pb} = -0.014$, $p = 0.767$). The mechanism is an *E detection paradox*: *install_dep* repositories score highest on E (mean 22.8)—they specify environments explicitly, but with version conflicts—while *missing_module* (5.2) and *missing_data* (4.4) score near zero. Critically, *success* repositories score lower ($E = 10.3$) than *install_dep*, so directional cancellation across the binary label produces near-zero (slightly negative) correlation. Figure 2 makes this concrete. Pairwise effect sizes confirm failure-mode separability: E separates *install_dep* from *missing_module* with Cohen's $d = 1.16$

Table 3. Category-level discriminability on the 423-repository sample. H: Kruskal-Wallis statistic, $df = 4$ (primary criterion). r_{pb}: point-biserial with binary success label (secondary; near-zero is expected for a failure-mode characterisation instrument).

Cat.	Name	w_i	r_{pb}	p_{pb}	H (p_{KW})
E	Environment spec.	0.30	−0.014	0.767	96.89 (<0.001)
A	Data accessibility	0.25	+0.016	0.745	16.12 (0.003)
D	Documentation	0.20	+0.028	0.572	53.09 (<0.001)
C	Code portability	0.15	+0.060	0.215	56.55 (<0.001)
S	Repro. signals	0.10	+0.153**	0.002	40.19 (<0.001)

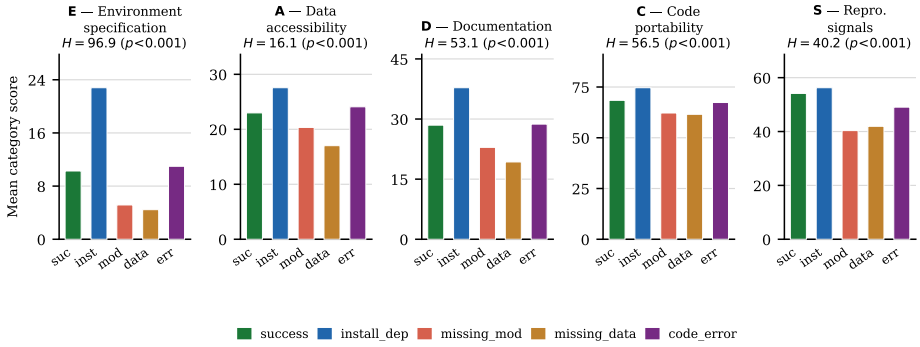

Fig. 2. Mean category score by failure mode (423 repositories; 84–85 per class). The E panel illustrates the detection paradox: *install_dep* scores highest on environment specification, yet fails at install time; *success* scores lower. Kruskal-Wallis H and p-values per panel.

($p < 0.001$, KS = 0.68) and from *missing_data* with $d = 1.25$ ($p < 0.001$, KS = 0.68). For a curator, this means a low-E score points to underspecification (most likely a missing-module or missing-data failure), while a high-E score with execution failure points to a version conflict—actionable information that binary prediction cannot provide.

Finding 2: C and D discriminate failure mode through complementary mechanisms. C achieves the second-strongest discriminability ($H = 56.55$, $p < 0.001$), driven by `import_resolvability`: repositories without dependency specification files receive score = 0, correctly penalising *missing_module* and *missing_data* repositories (which predominantly lack dependency files), while *install_dep* repositories (which specify dependencies, if incorrectly) score highest ($\bar{C} = 74.6$). D achieves $H = 53.09$ ($p < 0.001$), reflecting that well-documented repositories cluster in the *install_dep* and *code_error* modes—repositories that at least partially executed—rather than in *missing_module* and *missing_data* modes where documentation is sparse.

Finding 3: Weak binary association confirms failure-mode characterisation as the appropriate evaluation criterion. All $|r_{pb}| \leq 0.153$; only S achieves binary significance ($p = 0.002$). At the sub-metric level, r_{pb} ranges from -0.079 to $+0.145$; 21 of 26 sub-metrics have $|r_{pb}| < 0.08$. Five sub-metrics are nominally significant ($p < 0.05$): `notebook_exec_order` ($r_{pb} = +0.145$, $p = 0.003$), `seed_management` ($+0.113$, $p = 0.020$), `no_absolute_paths` ($+0.107$, $p = 0.027$), `ci_presence` ($+0.103$, $p = 0.035$), and `silent_failure_masking` ($+0.099$, $p = 0.041$). Applying Benjamini-Hochberg correction (FDR = 0.05, $m = 26$ tests), no sub-metric survives correction (minimum $q = 0.078$ for `notebook_exec_order`); all five nominally significant results are treated as exploratory. These weak associations are consistent with the readiness–outcome gap in this corpus. A post-hoc contingency analysis of $A = 0$ rates by failure mode shows that *install_ dep* repositories have the lowest $A = 0$ rate (8%) while *missing_ data* and *success* repositories have the highest (both 24%) ($\chi^2 = 9.89$, $df = 4$, $p = 0.042$).

Worked example. A representative *install_ dep* repository scores RRS = 54 (pinned requirements, Zenodo data pointer, structured README), yet fails at install due to a transitive version conflict; a *success* repository with RRS = 14 (no pins, no container) executes successfully because implicit dependencies happen to resolve. Together, these two repositories illustrate the readiness–outcome gap.

4.3 Robustness Diagnostics

The following diagnostics use AUC-ROC as a non-collapse probe—verifying the composite does not implicitly behave as a binary classifier—rather than as an optimisation target.

Weight stability. Rank-stability analysis (each weight varied ±50% with proportional redistribution; 20 steps per category) yields Kendall's $\tau \geq 0.911$ across all five categories, confirming stable repository ordering under substantial weight perturbation. A weight grid search over the simplex (126 configurations) finds maximum AUC = 0.581—only +0.045 above the default 0.536; the AUC-optimal configuration concentrates weight on S, the category with highest binary correlation. The community rubric makes any such policy adjustment explicit and auditable.

Gate robustness. Sweeping τ from 10 to 70 yields AUC 0.528–0.536 (span 0.008); linear ($k = 1$) and default ($k = 1.5$) exponents achieve identical AUC (0.536). The gate is empirically insensitive because many sub-metrics return binary values; its significance is normative, encoding a curation policy claim about sub-threshold failures.

Leave-one-category-out (Table 4). Removing E or A *improves* binary AUC (+0.028, +0.020 respectively)—both categories add noise to binary discrimination precisely because they characterise failure mode rather than predict it. Removing C or S degrades AUC (-0.022 each), reflecting their modest correlation with binary success. The LOCO span is 0.050 units; all deltas fall within the 95% bootstrap CI $[0.471, 0.602]$, confirming failure-mode characterisation and binary prediction are structurally distinct properties.

Table 4. Leave-one-category-out AUC. Baseline = 0.536 [0.471, 0.602] (95% CI). Positive Δ: removal *increases* binary AUC.

Removed	**AUC**	**ΔAUC**
None (full model)	0.536	—
$-S$ (Repro. signals)	0.514	−0.022
$-C$ (Code portability)	0.513	−0.022
$-D$ (Documentation)	0.536	+0.001
$-A$ (Data accessibility)	0.556	+0.020
$-E$ (Env. spec.)	0.564	+0.028

4.4 Rubric Comparison: Default versus Bioinformatics Profile

We contrast the default profile with the FAIR-aligned bioinformatics profile ($w_A = 0.40$, $w_E = 0.35$, $w_D = 0.10$, $w_C = 0.05$, $w_S = 0.10$) on the 423-repository sample. The most consequential change is w_A: $0.25 \rightarrow 0.40$. Since *install_ dep* repositories have the lowest $A = 0$ rate (8%) and *missing_ data*/*success* the highest (both 24%), increasing w_A systematically downranks *missing_ data* repositories and upranks *install_ dep*, reflecting the stated FAIR data priority. Reducing w_C ($0.15 \rightarrow 0.05$) diminishes the portability penalty on *missing_ module* repositories; reducing w_D ($0.20 \rightarrow 0.10$) down-weighs the signal separating *code_ error* from lower-documentation modes. Per-repository Spearman analysis ([30]) confirms strong overall rank agreement with policy-relevant reorderings in the lower-scoring quartile. The reordering is auditable, attributable (stated FAIR priority), and reproducible (same YAML profile = same result)—properties generally absent when weights are embedded opaquely in source code.

4.5 Partial ROS Computation and RCS Formula Validation

We proxy four ROS components from the ablation corpus failure-mode labels: $I = 100$ iff failure mode $\neq$ *install_ dep*; $X = 100$ iff failure mode = *success*; $N = 100 \times$ `success_nb_count/total_exec_count`; $E' = 100$ iff failure mode $\neq$ *missing_ module*. Output determinism Δ and test pass rate T require independent execution and are unavailable. Available component weight = 0.75, giving $\alpha = 0.525$ and $\text{RCS} = 0.475 \times \text{RRS} + 0.525 \times \text{ROS}_{\text{partial}}$. Table 5 reports component means and aggregate scores by failure mode.

Note on proxy validation. Since X derives directly from the ground-truth label, ROS AUC = 1.000 and RCS AUC = 0.993 are artefacts of construction, not independent findings; the substantive signals are: (i) *I and E' are independent*: *install_ dep* repositories have $I = 0$ yet $E' = 100$, confirming package-resolution and runtime import failure are distinct phenomena. (ii) *N grades partial execution*: among non-success repositories, N yields a diagnostic gradient—*code_ error* (13.3%) > *missing_ data* (8.6%) > *missing_ module* (3.8%)—absent from binary X. (iii) *RCS correctly re-orders*: *install_ dep* ranks second on RRS (26.8) but falls to the lowest RCS (19.8) via near-zero ROS (13.5), validating the coverage-weighting mechanism.

Table 5. ROS component means and aggregate scores by failure mode (partial ROS; Δ and T unavailable; $\alpha = 0.525$). All values on $[0, 100]$ scale.

Failure mode	I	X	N	E'	ROS	RCS	RRS
success	100	100	100.0	100	100.0	59.4	14.6
install_dep	0	0	1.3	100	13.5	19.8	26.8
missing_module	100	0	3.8	0	40.5	24.6	7.0
missing_data	100	0	8.6	100	54.5	31.7	6.4
code_error	100	0	13.3	100	55.1	35.7	14.3

Code and Data Availability: The ReproScore implementation, rubric profiles, and per-repository provenance records are publicly available at https://github.com/myVSR/reproscore [30] and archived on Software Heritage [26].

5 Discussion

The Readiness–Outcome Gap as a Repository Curation Problem

When a new software deposit is received, a curator relying on a static score labelled "reproducibility" acts on a quantity that measures artefact presence—not whether the software executes. RRS is designed as a *diagnostic* instrument: it identifies which dimension of readiness is deficient (environment, data, documentation, portability, or determinism signals), enabling targeted remediation rather than predicting whether execution will succeed. The per-category profile is the primary curatorial instrument; the composite RRS serves as a structural completeness summary only. Statistical separability between failure modes (Kruskal-Wallis H) is therefore the appropriate criterion; binary AUC appears in Section 4 only as a non-collapse probe. A repository with a pinned `requirements.txt`, a Zenodo data pointer, and a structured README scores well by any static metric; if it fails at install time due to a transitive version conflict, the score misrepresents the deposit's reuse potential. ReproScore resolves this by making the distinction explicit: RRS, ROS and their combination as RCS are formally distinct quantities; the coverage weight α communicates how much execution evidence underpins the composite. In OAIS terms, RRS fits within the Ingest functional entity as a pre-acceptance quality gate; ROS/RCS applies during Archival Storage for ongoing usability monitoring. The DCC Curation Lifecycle similarly positions quality appraisal as a continuous obligation—ReproScore's coverage-adaptive composite (α rising as execution evidence accumulates) supports this incrementally rather than requiring a one-time full execution audit. We stress that RCS is a design proposal: the blending formula has been validated partially in Section 4.5 but not at scale; $\alpha_{\max} = 0.70$ and per-component ROS weights are design choices awaiting empirical calibration.

Category Weights as Curation Priorities

The weight assignment encodes a principled curation priority: $w_E = 0.30$ is empirically supported by $H = 96.89$ ($p < 0.001$), separating install-dependency

failures (mean $E = 22.8$) from missing-module failures ($E = 5.2$). The E detection paradox—success repositories score *lower* on E than install-dep—confirms E characterises failure *type*, not failure *probability*. A ($w = 0.25$) achieves $H = 16.12$ ($p = 0.003$); post-hoc contingency analysis confirms the mechanism: *install_dep* repositories have the lowest $A = 0$ rate (8%) versus *missing_data* (24%) ($\chi^2 = 9.89$, $df = 4$, $p = 0.042$). C ($w = 0.15$) achieves $H = 56.55$ ($p < 0.001$), driven by `import_resolvability`: repositories lacking dependency files score 0, correctly penalising *missing_module* and *missing_data*, while *install_dep* scores highest ($\bar{C} = 74.6$); LOCO $\Delta = -0.022$ confirms binary discrimination. S ($w = 0.10$) yields the highest binary predictive power (AUC = 0.611); `notebook_exec_order` is the strongest single predictor (AUC = 0.638, $r_{pb} = +0.145$, $p = 0.003$). $w_S = 0.10$ is retained because S captures workflow practices rather than primary execution artefacts. The $E < 10$ hard penalty fires for 92% of *missing_data* and 80% of *missing_module* repositories, validating the threshold. In practice: high E with failure suggests a dependency conflict; low E suggests underspecification; low A suggests missing data pointers; low C suggests portability defects.

Implications for Practice

Repository curators and data librarians gain an automated pre-execution triage layer: the five-category profile flags repositories with $E < \tau_E$ or $A = 0$ without code execution. *Researchers* should act on the category profile: low E calls for dependency pinning or containerisation; a high-RRS, low-ROS gap signals a version conflict that static analysis cannot detect. *Platform operators* can adapt the model to domain curation policies via the community rubric mechanism (e.g. $w_A = 0.40$ for a FAIR-aligned bioinformatics archive [41]), producing a versioned, auditable profile rather than an undocumented scoring heuristic. The static-first design also reduces the environmental footprint (cf. [24]) of large-scale curation: screening deposits for readiness before committing to container builds and sandbox execution avoids wasting CPU cycles on non-executable artefacts.

Limitations and Threats to Validity

Domain bias: The evaluation corpus comprises exclusively of Python/Jupyter repositories drawn from biomedical publications indexed in PubMed Central [28], and two sub-metrics (`notebook_exec_order`, `inline_explanation_density`) are notebook-specific in their current operationalisation. Generalisability to script-only, R, or compiled-language repositories remains untested, and sub-metric applicability varies by research domain and programming paradigm. Discriminability and metric coverage will differ for R workflows, non-notebook packages, or repositories with complex build systems. Generalisation claims should be understood with these scope limits. *Static coverage*: sub-metrics detect artefact presence, not semantic correctness—a pinned `requirements.txt` with conflicting constraints scores well on E yet fails at install. The two-tier architecture

addresses this structurally rather than improving the static approximation. *Label quality*: ground-truth labels aggregate notebook-level records to repositories via priority ordering; 77% of notebooks in multi-notebook repositories share the dominant failure mode (95% for *install_dep*). *Feature expressivity*: 10 of 26 sub-metrics return binary values for $> 90\%$ of repositories, limiting the gate's effect; the `import_resolvability` boundary condition (score = 0 when no dependency files are present) drives much of C's discriminability. *RCS validation*: $\alpha_{\max} = 0.70$ and per-component ROS weights are design choices awaiting calibration against an independent execution dataset. *Multiple comparisons*: sub-metric r_{pb} tests apply Benjamini-Hochberg correction (FDR = 0.05, $m = 26$); no sub-metric survives correction (minimum $q = 0.078$), confirming the exploratory nature of the reported associations.

6 Conclusion

ReproScore separates *reproducibility readiness* (RRS: static artefact analysis) from *reproducibility outcome* (ROS: execution-based probes), combining them into a coverage-adaptive composite (RCS). The community rubric externalises assessment priorities as a versioned YAML profile, making weight choices explicit and auditable for digital library curation workflows. Evaluated on 423 repositories, the ablation confirms stable category weights ($\tau \geq 0.911$ under $\pm 50\%$ perturbation), an insensitive gate function (AUC range 0.008), and an environment category that strongly discriminates failure mode ($H = 96.9$, $p < 0.001$; $r_{pb} = -0.014$). The new `notebook_exec_order` sub-metric is the strongest single predictor (AUC = 0.638); S alone achieves AUC = 0.611, suggesting that archives can prioritise notebook execution-order remediation as a high-yield, low-effort curatorial intervention. Partial RCS validation confirms the blending formula re-orders correctly: *install_dep* repositories (RRS = 26.8) are pulled to the lowest RCS (19.8)—below all partially-executing failure modes—by their near-zero execution score. A repository can be well-specified yet fail to execute; another can be poorly specified yet succeed. A framework that names this gap is more informative than one that collapses readiness and outcome into a single score. Future work includes: validating ROS and RCS at scale with independently run execution probes; aligning the rubric schema with CodeMeta and FAIR4RS, such that scores travel with artefact metadata; and longitudinal tracking of RRS/ROS/RCS over repository snapshots.

Acknowledgments. This work was supported in part by the German Research Foundation (DFG) through the following projects: Jupyter4NFDI (DFG 521453681 [17]), find.software (DFG 567156310 [14]), MaRDI (DFG 460135501 [37] as well as SeDOA (DFG 556323977 [36]) and HYP*MOL (DFG 514664767).

The text of this manuscript was improved with the following AI tools: ChatGPT and Claude.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. FAIRsFAIR data object assessment metrics. Zenodo **4081213** (2020). https://doi.org/10.5281/zenodo.4081213
2. Akdeniz, E.K., Tekir, S., Hinnawi, M.N.A.A.: An end-to-end system for reproducibility assessment of source code repositories via their readmes. arXiv preprint arXiv:2310.09634 (2023)
3. Baker, M.: 1,500 scientists lift the lid on reproducibility. Nature **533**, 452–454 (2016). https://doi.org/10.1038/533452a
4. Bandrowski, A., Roelandse, M.: SciScore, a tool that can measure rigor criteria presence or absence in a biomedical study. In: RExPO22. ScienceOpen (2022)
5. Bogin, B., Yang, K., Gupta, S., Richardson, K., Bransom, E., Clark, P., Sabharwal, A., Khot, T.: Super: Evaluating agents on setting up and executing tasks from research repositories. In: Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing. pp. 12622–12645 (2024)
6. Brinckman, A., Chard, K., Gaffney, N., Hategan, M., Jones, M.B., Kowalik, K., Kulasekaran, S., Ludäscher, B., Mecum, B.D., Nabrzyski, J., et al.: Computing environments for reproducibility: Capturing the "whole tale". Future Generation Computer Systems **94**, 854–867 (2019)
7. Campbell, G.A., Papapetrou, P.P.: SonarQube in action. Manning Publications Co. (2013)
8. Carlson, D.E., Chavarriaga, R., Liu, Y., Lotte, F., Lu, B.L.: The nerve-ml (neural engineering reproducibility and validity essentials for machine learning) checklist: ensuring machine learning advances neural engineering. Journal of Neural Engineering **22**(2), 021002 (2025)
9. Chue Hong, N.P., Katz, D.S., Barker, M., Lamprecht, A.L., Martinez, C., Psomopoulos, F.E., Harrow, J., Castro, L.J., Gruenpeter, M., Martinez, P.A., et al.: FAIR principles for research software (FAIR4RS principles). Zenodo (2022). https://doi.org/10.15497/RDA00068
10. Costa, L., Barbosa, S., Cunha, J.: Evaluating tools for enhancing reproducibility in computational scientific experiments. In: Proceedings of the 2nd ACM Conference on Reproducibility and Replicability. p. 46–51. ACM REP '24, Association for Computing Machinery, New York, NY, USA (2024). https://doi.org/10.1145/3641525.3663623, https://doi.org/10.1145/3641525.3663623
11. Di Cosmo, R., Zacchiroli, S.: Software heritage: Why and how to preserve software source code. iPRES 2017 – 14th International Conference on Digital Preservation (2017). https://doi.org/10.1145/3238147.3241985
12. Forde, J., Head, T., Holdgraf, C., Panda, Y., Nalvarete, G., Ragan-Kelley, B., Sundell, E.: Reproducible research environments with repo2docker (2018), https://openreview.net/forum?id=B1lYOwuoxm
13. Gaignard, A., Rosnet, T., De Lamotte, F., Lefort, V., Devignes, M.D.: Fair-checker: supporting digital resource findability and reuse with knowledge graphs and semantic web standards. Journal of Biomedical Semantics **14**(1), 7 (2023). https://doi.org/10.1186/s13326-023-00289-5
14. Gey, R., Mietchen, D., Karras, O., Wittenborg, T., Schubotz, M., Bumberger, J.: find.software: Foundations for interdisciplinary discovery of (research) software. Research Ideas and Outcomes **11**, e179253 (2025). https://doi.org/10.3897/rio.11.e179253
15. Goggins, S., Lumbard, K., Germonprez, M.: Open source community health: Analytical metrics and their corresponding narratives. In: 2021 IEEE/ACM 4th In-

ternational Workshop on Software Health in Projects, Ecosystems and Communities (SoHeal). pp. 25–33. IEEE (2021). https://doi.org/10.1109/SoHeal52568.2021.00010

16. González-Beltrán, A., Li, P., Zhao, J., Avila-Garcia, M.S., Roos, M., Thompson, M., van der Horst, E., Kaliyaperumal, R., Luo, R., Lee, T.L., Lam, T.W., Edmunds, S., Sansone, S.A., Rocca-Serra, P.: From Peer-Reviewed to Peer-Reproduced in Scholarly Publishing: The Complementary Roles of Data Models and Workflows in Bioinformatics **10**(7), e0127612 (2015). https://doi.org/10.1371/JOURNAL.PONE.0127612
17. Hagemeier, B., Bleier, A., Flemisch, B., Reuter, K., Dogaru, G., Mietchen, D., Lieber, M.: Jupyter4NFDI - Proposal for the Integration Phase of Base4NFDI (Dec 2025). https://doi.org/10.5281/zenodo.17867919
18. Higgins, S.: The DCC curation lifecycle model. In: Proceedings of the 8th ACM/IEEE-CS Joint Conference on Digital Libraries. p. 453. JCDL '08, Association for Computing Machinery, New York, NY, USA (2008). https://doi.org/10.1145/1378889.1378998, https://doi.org/10.1145/1378889.1378998
19. Hu, C., Zhang, L., Lim, Y., Wadhwani, A., Peters, A., Kang, D.: REPRO-BENCH: Can Agentic AI Systems Assess the Reproducibility of Social Science Research? In: Findings of the Association for Computational Linguistics: ACL 2025. pp. 23616–23626 (2025). https://doi.org/10.48550/arXiv.2507.18901
20. Jones, M.B., Boettiger, C., Mayes, A.C., Smith, A., Slaughter, P., Niemeyer, K., Gil, Y., Fenner, M., Schulte, K., Chamberlin, L., et al.: CodeMeta: an exchange schema for software metadata. Knowledge Exchange (2017), version 2.0, https://codemeta.github.io
21. Lavoie, B.: The Open Archival Information System (OAIS) Reference Model: Introductory Guide. DPC Technology Watch Report 14-02, Digital Preservation Coalition, York, UK (2014). https://doi.org/10.7207/twr14-02
22. Mao, A., Garijo, D., Fakhraei, S.: Somef: A framework for capturing scientific software metadata from its documentation. In: 2019 IEEE International Conference on Big Data (Big Data). pp. 3032–3037 (2019). https://doi.org/10.1109/BigData47090.2019.9006447, http://dgarijo.com/papers/SoMEF.pdf
23. Nüst, D., Eglen, S.J.: CODECHECK: an Open Science initiative for the independent execution of computations underlying research articles during peer review to improve reproducibility. F1000Research **10**, 253 (2021). https://doi.org/10.12688/f1000research.51738.2
24. Pazienza, A., Baselli, G., Vinci, D.C., Trussoni, M.V.: A holistic approach to environmentally sustainable computing. Innovations in Systems and Software Engineering (February 2024). https://doi.org/10.1007/S11334-023-00548-9
25. Pimentel, J.F., Murta, L., Braganholo, V., Freire, J.: A large-scale study about quality and reproducibility of jupyter notebooks. In: 2019 IEEE/ACM 16th international conference on mining software repositories (MSR). pp. 507–517. IEEE (2019). https://doi.org/10.1109/MSR.2019.00077
26. Samuel, S.: Reproscore: Separating readiness from outcome in research software reproducibility assessment (May 2026), https://archive.softwareheritage.org/swh:1:rev:5b4de0124a482d6266c39b0538a8e8da75cd68fa;origin=https://github.com/myVSR/reproscore;visit=swh:1:snp:ba15617a05c38cb074a8e49a0074a475f272724e
27. Samuel, S., Mietchen, D.: Dataset of a Study of Computational reproducibility of Jupyter notebooks from biomedical publications (Aug 2023). https://doi.org/10.5281/zenodo.8226725

28. Samuel, S., Mietchen, D.: Computational reproducibility of Jupyter notebooks from biomedical publications. GigaScience **13**, giad113 (2024). https://doi.org/10.1093/gigascience/giad113
29. Samuel, S., Mietchen, D.: FAIR Jupyter: A Knowledge Graph Approach to Semantic Sharing and Granular Exploration of a Computational Notebook Reproducibility Dataset. Transactions on Graph Data and Knowledge **2**(2), 4:1–4:24 (2024). https://doi.org/10.4230/TGDK.2.2.4
30. Samuel, S., Mietchen, D.: ReproScore (2026), https://doi.org/10.5281/zenodo.20154206, The ReproScore implementation, rubric profiles, and per-repository provenance record
31. Siegel, Z.S., Kapoor, S., Nagdir, N., Stroebl, B., Narayanan, A.: Core-bench: Fostering the credibility of published research through a computational reproducibility agent benchmark. arXiv preprint arXiv:2409.11363 (2024). https://doi.org/10.48550/arXiv.2409.11363
32. Soiland-Reyes, S., Sefton, P., Crosas, M., Castro, L.J., Coppens, F., Fernández, J.M., Garijo, D., Grüning, B., La Rosa, M., Leo, S., et al.: Packaging research artefacts with RO-Crate. Data Science **5**(2), 97–138 (2022). https://doi.org/10.3233/DS-210053
33. Spaaks, J.H., Verhoeven, S., Diblen, F., Drost, N., Hutton, A., Garcia Gonzalez, J.: howfairis: analyse a github or gitlab repository's compliance with the fair software recommendations. Zenodo. https://doi.org/10.5281/zenodo.5013050 (2021)
34. Starace, G., Jaffe, O., Sherburn, D., Aung, J., Chan, J.S., Maksin, L., Dias, R., Mays, E., Kinsella, B., Thompson, W., et al.: Paperbench: Evaluating ai's ability to replicate ai research. arXiv preprint arXiv:2504.01848 (2025). https://doi.org/10.48550/arXiv.2504.01848
35. Stodden, V., Seiler, J., Ma, Z.: An empirical analysis of journal policy effectiveness for computational reproducibility. Proceedings of the National Academy of Sciences **115**(11), 2584–2589 (2018). https://doi.org/10.1073/pnas.1708290115
36. Stäcker, T., Apel, J., Arning, U., et al.: SeDOA – Servicestelle Diamond Open Access (Mar 2025). https://doi.org/10.5281/zenodo.15043760
37. The MaRDI consortium: MaRDI: Mathematical Research Data Initiative Proposal (May 2022). https://doi.org/10.5281/zenodo.6552436
38. Trisovic, A., Lau, M.K., Pasquier, T., Crosas, M.: A large-scale study on research code quality and execution. Scientific Data **9**(1), 60 (2022). https://doi.org/10.1038/s41597-022-01143-6
39. Vangala, B.P., Adibifar, A., Gehani, A., Malik, T.: Ai-generated code is not reproducible (yet): An empirical study of dependency gaps in llm-based coding agents. arXiv preprint arXiv:2512.22387 (2025). https://doi.org/10.48550/arXiv.2512.22387
40. Virtanen, P., Gommers, R., Oliphant, T.E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., et al.: SciPy 1.0: fundamental algorithms for scientific computing in Python. Nature methods **17**(3), 261–272 (2020). https://doi.org/10.1038/s41592-019-0686-2
41. Wilkinson, M.D., Dumontier, M., Aalbersberg, I.J., Appleton, G., Axton, M., Baak, A., Blomberg, N., Boiten, J.W., da Silva Santos, L.B., Bourne, P.E., et al.: The FAIR Guiding Principles for scientific data management and stewardship. Scientific data **3**(1), 1–9 (2016). https://doi.org/10.1038/sdata.2016.18