Automated Result-Level Reproduction of Machine-Learning Papers with LLMs: A Knowledge-Graph Pipeline and a Taxonomy of Failure

Suhel Acharya[1,*], Chutiporn Anutariya[1], Yasuyuki Minamiyama[2], Hideaki Takeda[3] and Vilas Wuwongse[4]

[1]*Faculty of Advanced Science and Technology, Asian Institute of Technology, Thailand*
[2]*Institute of Social Science, the University of Tokyo, Japan*
[3]*National Institute of Informatics, Japan*
[4]*Mahasarakham University, Thailand*

Abstract

Reconstructing a machine-learning repository's software environment is not the same as reproducing the number its paper reports, yet most reproducibility studies stop at the former. We ask the harder question directly: given a paper and its code, can an automated engine re-run the experiment and recover the claimed result? Our design principle is that the "how to run it" must itself be *extracted as metadata*, not hand-written, so the method generalises to any new paper. Building on the SRAF auditing framework and the RDIP provenance profile, we introduce `rdip:ExecutionRecipe`, a machine-readable recipe (run command, setup steps, required data and checkpoints, produced metric) that a language model extracts from a repository's documentation; a two-phase, streaming pipeline then rebuilds each environment, runs the recipe, reads the obtained number, compares it to the paper's claim, and discards the artifacts. Over a corpus of 96 machine-learning repositories, 26 build, 17 are attempted, and 13 yield a recipe, but *none* reproduce their headline number automatically. We organise the failures into a compact taxonomy dominated by artifact and documentation gaps rather than compute limits. A positive-control ablation isolates the bottleneck: on three known-reproducible artifacts, autonomous extraction reproduces $0/3$, whereas the same execution-and-comparison pipeline given *correct* recipes reproduces $2/3$. The gap between running code and reproducing results is therefore real, measurable, and driven principally by the difficulty of extracting an executable recipe from heterogeneous artifacts.

Keywords

Computational Reproducibility, Result-Level Reproduction, Research Knowledge Graph, Execution Metadata, Failure Taxonomy, Large Language Models

1. Introduction

A machine-learning result is a number: an accuracy, an F1, a mean average precision. Reproducing a study, in the strict sense, means obtaining that number again by re-running the released code on the stated data. Yet the large empirical literature on computational reproducibility rarely tests this directly. Studies instead measure proxies: whether an artifact exists, whether a paper reports the necessary items [1, 2], whether reviewers can certify it by hand [3], or, at best, whether the software environment can be reconstructed. Environment reconstruction is a genuine hurdle, but a repository that installs cleanly is not thereby shown to reproduce its published numbers. The last step, from a built environment to a matching result, is the one that matters to a reader and the one that is almost never measured at scale.

This paper attempts that last step automatically. The companion SRAF framework [4] audits a study statically: it lifts a paper and its repository into a research-provenance knowledge graph and scores the

IJCKG 2026 Posters, Demos and Workshops, November 19–21, 2026, Bangkok, Thailand
*Corresponding author.
suhel.acharya@ait.ac.th (S. Acharya); chutiporn@ait.ac.th (C. Anutariya); minamiyama@iss.u-tokyo.ac.jp (Y. Minamiyama); takeda@nii.ac.jp (H. Takeda); vilasw@gmail.com (V. Wuwongse)
0009-0005-0376-5628 (S. Acharya); 0000-0001-7101-212X (C. Anutariya); 0000-0002-7280-3342 (Y. Minamiyama); 0000-0002-2909-7163 (H. Takeda); 0009-0005-8131-7476 (V. Wuwongse)
© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073
published 2026-11-06

CEUR-WS.org/Vol-4316/rdm-full-2.pdf

metadata for reproducibility readiness, finding that only about half of a 96-repository corpus [5] even builds and that most artifacts lack the deep, executable detail reproduction requires. SRAF predicts; here we *test* that prediction by actually running the code and checking the number. The two are complementary: the static score says how well a study is described, and the dynamic re-run says whether that description suffices.

The central design decision is how the engine learns to run each experiment. Hand-writing a run command per repository would not scale and, more importantly, would not generalise: an engine that must be told the command for every paper has learned nothing transferable. We therefore treat the command as one more piece of extracted metadata. For each study the engine reads the repository documentation and emits a structured execution recipe, the same way it would extract a dataset name or a random seed. A new paper is handled by the same extraction, with no human in the loop. We formalise this as `rdip:ExecutionRecipe`, an extension to the RDIP provenance profile [6], and build a streaming pipeline that, per repository, extracts the recipe, rebuilds the environment, runs it, reads the obtained metric, compares it to the paper's claim, and deletes the heavy artifacts so the corpus never accumulates on disk.

Applied to the 96-repository corpus, the pipeline delivers a stark result: of the 26 repositories that build and the 17 we attempt, none reproduce their reported number automatically. Rather than a dead end, the value is in why. We contribute (i) an engine-driven method for result-level reproduction in which the execution recipe is extracted metadata, formalised as `rdip:ExecutionRecipe`; (ii) a taxonomy of reproduction failure, derived automatically from real re-run attempts, that is dominated by artifact and documentation gaps rather than compute limits; and (iii) a positive-control ablation that separates the two things a null result could mean. On three artifacts known to be reproducible, autonomous extraction reproduces none, but the very same execution-and-comparison machinery, given correct recipes, reproduces two of the three. The execution pipeline works; the bottleneck is extracting a runnable recipe from heterogeneous artifacts, which is exactly where further research is needed.

2. Related Work

Measuring computational reproducibility. Community surveys report that most researchers have failed to reproduce others' work [7], and analyses of computational articles find that they routinely lack the information needed for independent reproduction [8]. In machine learning specifically, hyperparameters, data splits, and code are under-reported [2], and the fraction of a paper reproducible from its text alone is bounded by that missing detail [9]. Normative instruments such as the ML Reproducibility Checklist [1] enumerate what a paper should report, and the ACM Artifact Review and Badging policy [3] certifies artifacts through human evaluation. These establish and check good practice, but manually and largely at the level of reporting or availability; they do not automatically re-run an artifact and verify its numbers. Venues such as ReScience C [10] do achieve result-level reproduction, but by expert human replication, one artifact at a time.

From environment reconstruction to result reproduction. A practical line of work asks whether a repository's declared environment can be rebuilt at all. The SRAF study [4] reconstructs each of 96 environments on a containerless cluster and finds roughly half resolve and build, with a software license the strongest predictor of success. This establishes the floor: without a build there is no run. Our work takes the next step, from "does it build" to "does it reproduce the number," and shows that clearing the build bar is far from sufficient.

Language models for extraction and execution. Language models are increasingly used to extract structured scientific metadata from unstructured text [11, 12], and to populate or query knowledge graphs [13]. SRAF uses such extraction to build its provenance graph. We extend this idea from descriptive metadata to *operational* metadata: the executable recipe that reproduces a result. This

reframes autonomous reproduction as an extraction-plus-execution problem over a provenance graph, and lets us measure, separately, how well the recipe can be extracted and how well it runs.

LLM code agents. A fast-growing line equips language models with tools to *act* on code: agents that resolve real repository issues through an agent-computer interface, evaluated on whether their patch passes the project's tests [14, 15]. These agents show that models can plan and carry out multi-step software tasks, but they target bug-fixing and feature implementation, and success is defined by a passing test suite rather than by recovering a paper's reported number. Our recipe extractor is a narrower, read-only instance of the same idea: it turns documentation into an executable recipe rather than editing code, and our positive-control ablation suggests that a stronger, tool-using agent of this kind is the most promising route to closing the extraction gap we identify.

Positioning. To our knowledge, no prior work reports an automatic, result-level reproduction sweep over a machine-learning corpus in which the run procedure is itself extracted as metadata, together with a taxonomy of why reproduction fails and a positive-control ablation that isolates extraction from execution. That is the gap this paper fills, as a dynamic companion to the static SRAF instrument.

3. The SRAF Engine and Result-Level Reproduction

We first recap the ontology and engine this work builds on (Section 3.1), then introduce our result-level reproduction method: the execution recipe (Section 3.2), the pipeline that runs it (Section 3.3), and how outcomes are judged (Section 3.4).

3.1. The RDIP Ontology and the SRAF Engine

We build directly on two published components and do not re-derive them here. RDIP [6] is a lightweight, project-centric integration profile that composes established vocabularies (VIVO, DCAT, PROV-O, BIBO/CiTO) around a central `rdip:ResearchProject` hub and promotes reproducibility-relevant constructs to first-class citizens: `rdip:Parameter` and `rdip:RandomSeed`, `rdip:ComputingEnvironment` and `rdip:EnvironmentSpec`, `rdip:SoftwareDependency`, and `rdip:EvaluationResult` pairing a metric and value with a dataset split (Figure 1). SRAF [4] lifts each study into a per-study RDIP graph by fusing deterministic repository parsing with language-model extraction from the paper, and scores it with FAIR-R, a graded, standards-grounded instrument that extends FAIR with a dedicated Reproducible dimension. In this paper the FAIR-R score is a given input; our contribution operates one level deeper, on whether the study's result can actually be regenerated.

3.2. The Execution Recipe as Extracted Metadata

The procedure that reproduces a paper's headline number lives in the artifacts, in a README's usage or evaluation section, a `run.sh` or `Makefile`, and the paper's "reproducing results" text, not in a form any engine can execute directly. Our principle is to treat that procedure as metadata to be *extracted*, exactly like a dataset name or a seed. We introduce `rdip:ExecutionRecipe` as a proposed extension to the published RDIP profile [6]: a structured object with fields for the single run command that produces the headline metric (`runCommand`), the ordered setup steps that prepare the environment, data, and checkpoints (`setupStep`), the datasets and checkpoints required (`requiresDataset`, `requiresCheckpoint`), the metric the command reports (`producesMetric`), the invoked entry point (`entryPoint`), and a self-reported confidence (`recipeConfidence`). A language model reads the repository documentation and emits this object as JSON (Listing 1); a deterministic mapper then lifts it into the study's RDIP graph as an instance of `rdip:ExecutionRecipe` (Listing 2), conforming to the class of Figure 2, so the run procedure becomes a queryable part of the provenance record. Because the recipe is extracted rather than supplied, the same pipeline applies unchanged to a paper it has never seen.

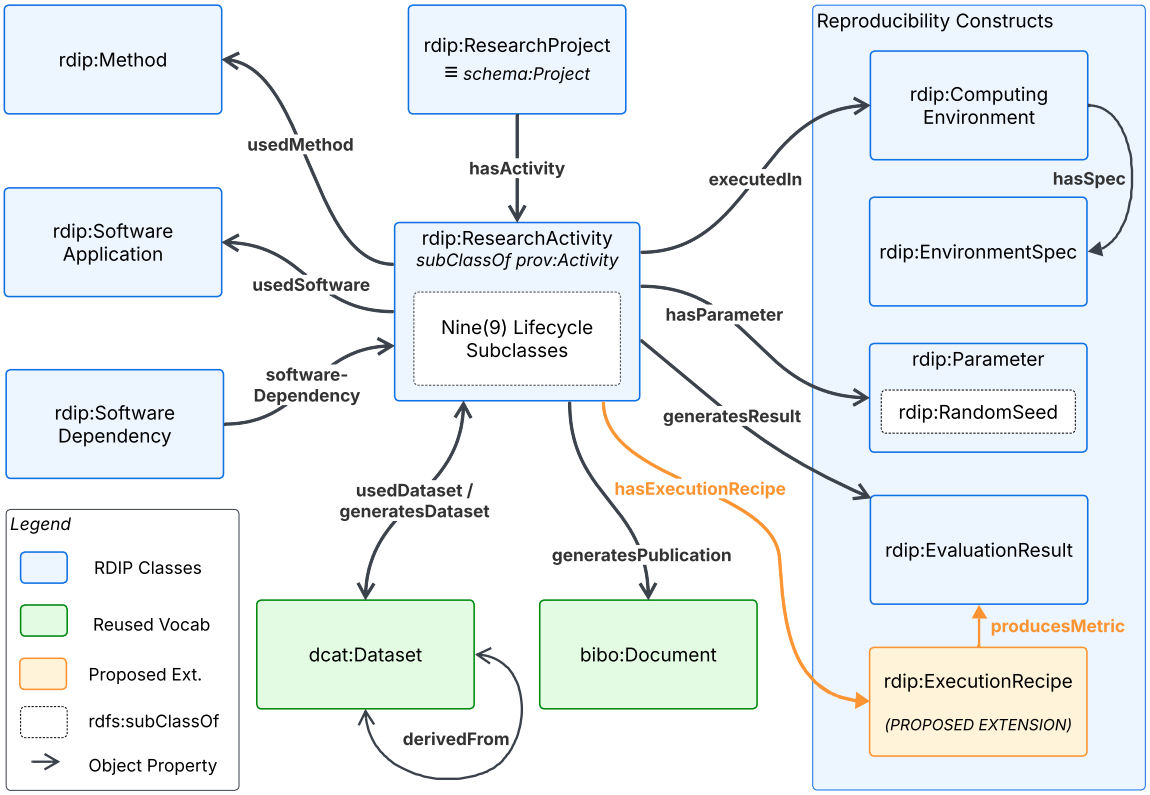


Figure 1: The reproducibility-relevant core of RDIP [6], with the proposed `rdip:ExecutionRecipe` extension (lower right).

Alignment with PROV-O. RDIP already reuses PROV-O throughout, and the recipe is deliberately not a free-standing class beside it. Because `rdip:ResearchActivity` is already a subclass of `prov:Activity`, the recipe needs only the other half of PROV's plan/activity pairing: an `rdip:ExecutionRecipe` is a `prov:Plan`, the plan an activity follows, distinct from the activity that follows it. The extension therefore adds no new modelling pattern; it instantiates one PROV-O already provides. `rdip:hasExecutionRecipe` is a convenience shortcut for the fully qualified route `prov:qualifiedAssociation` / `prov:hadPlan`, so a study that needs to state the agent or role as well can use the qualified form without a second vocabulary. The ordered `setupStep` values are likewise plans rather than opaque strings, because RDF imposes no order on repeated properties while setup order is load-bearing: a download must precede the extraction that consumes it. Modelling the recipe this way is what makes a reproduction attempt expressible as a *second* `prov:Activity` following the same plan, so an original and its reproduction sit in one provenance record rather than in two unrelated documents.

Why a knowledge graph rather than a structured file. Representing the run procedure in the graph, rather than as a JSON record beside it, buys three things a standalone format does not. First, the recipe is queryable *alongside* the environment facts lifted from the same study, so one query can ask which studies document a concrete run command but pin no dependency versions, or which declare a metric the environment could never produce. Second, `requiresDataset` and `producesMetric` range over `dcat:Dataset` and `rdip:EvaluationResult` rather than over bare strings: a required dataset resolves to its access level, and a produced metric to the claimed value it is meant to match, which is what turns a gated dataset from a log message into a property of the record. Third, recipes become validatable with the same SHACL machinery the FAIR-R rubric already uses, so a shape asserting that `runCommand` contains no unfilled placeholder would convert the single largest failure mode in

Section 5.2 into a static, pre-execution check.

Listing 1: Recipe emitted by the language model (JSON, abbreviated).

```
{
  "run_command": "python eval.py",
  "setup_steps": ["pip install -q torch torchvision",
                  "<write eval.py: load checkpoint,",
                  " eval on test split, print metric>"],
  "requires_dataset": [{"name": "CIFAR-10",
                        "download": "auto"}],
  "requires_checkpoint": "released weights",
  "produces_metric": [{"metric": "top-1 accuracy",
                       "dataset": "CIFAR-10"}],
  "entry_point": "eval.py",
  "confidence": "high"
}
```

Listing 2: The same recipe mapped into RDIP triples (abbreviated Turtle).

```
@prefix rdip: <https://w3id.org/rdip/> .
@prefix dcat: <http://www.w3.org/ns/dcat#> .
@prefix ex:   <https://example.org/study/s-demo/> .

ex:project a rdip:ResearchProject ;
    rdip:hasExecutionRecipe ex:recipe .
ex:recipe a rdip:ExecutionRecipe ;
    rdip:runCommand "python eval.py" ;
    rdip:setupStep  "pip install -q torch torchvision" ;
    rdip:entryPoint "eval.py" ;
    rdip:recipeConfidence "high" ;
    rdip:requiresDataset    ex:cifar10 ;
    rdip:requiresCheckpoint ex:ckpt ;
    rdip:producesMetric     ex:result .
ex:cifar10 a dcat:Dataset .
ex:ckpt    a rdip:Checkpoint .
ex:result  a rdip:EvaluationResult ;
    rdip:evaluationMetric "top-1 accuracy" .
```

Extracting the recipe. The recipe is produced by a single open-weight instruction model (Mistral-Small-24B, served locally), prompted with the repository's documentation, so the extraction step is itself reproducible and free of proprietary services. The prompt is deliberately conservative. It asks the model to prefer an *evaluation* command over full training when the paper reports a metric from a released checkpoint; to turn a checkpoint or dataset given only as a URL into explicit `setupStep` downloads that reference the resulting local path, never passing a URL where a file is expected; to select the configuration matching *this* paper's method when the repository is a multi-method toolbox; and, above all, to lower `recipeConfidence` and keep a command verbatim when it contains an unresolved placeholder, rather than inventing a value. When the documentation states no way to reproduce a reported number, the model returns a null command at low confidence. This self-report is load-bearing: it lets the pipeline decline a recipe it cannot run, and it is what turns the placeholder and no-command failures of Section 5.2 into machine-detectable outcomes rather than silent crashes.

3.3. A Two-Phase, Streaming Pipeline

Extraction and execution compete for the same accelerator: a 24B-parameter extraction model and a repository's experiment do not fit together on one 48 GB GPU. We therefore separate the work into two phases (Figure 3). **Phase 1 (recipe extraction)** gives the GPU to the language model, which reads each repository and writes its `rdip:ExecutionRecipe`. **Phase 2 (execution)** gives the GPU to the experiments; no model is loaded, and obtained metrics are read from the run log by pattern matching. The pipeline is *streaming and resumable*: it processes one repository at a time, and for each it clones the code, creates an isolated virtual environment, runs the recipe's setup steps and command as a single shell script (so that working-directory and environment changes persist), reads the obtained number,

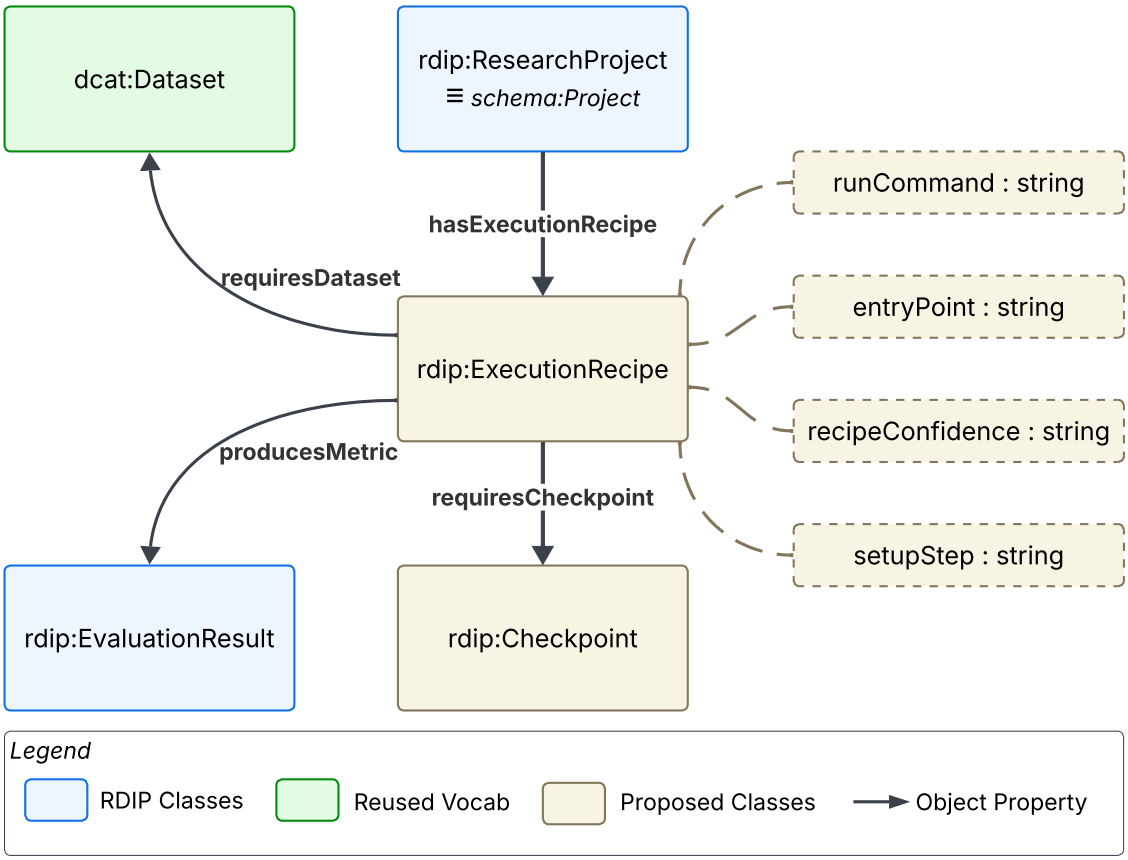


Figure 2: The proposed `rdip:ExecutionRecipe` class and its object and datatype properties.

compares it to the paper's claim within a tolerance, records the outcome, and then deletes the clone, environment, and any downloaded data. Only the comparison survives, so a corpus of arbitrarily large repositories never accumulates on disk.

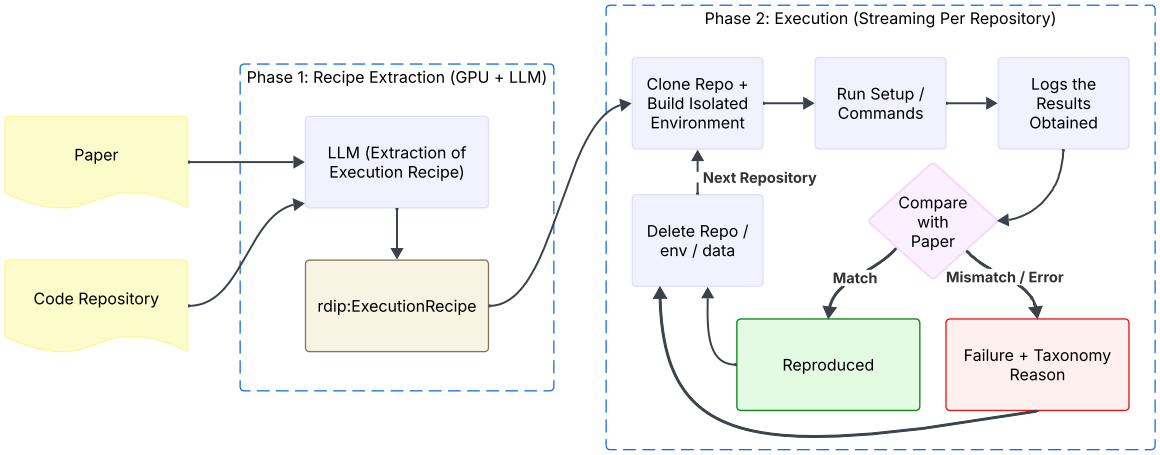


Figure 3: The two-phase, streaming reproduction pipeline.

3.4. Comparison and Failure Classification

A study counts as *reproduced* when the obtained value matches the paper's claimed value for the same metric within a relative tolerance. Every non-reproducing study is assigned a machine-detectable failure reason from its run log: a recipe with a placeholder path, no run command found, a missing system tool or Python dependency, a declared dependency that no longer installs because it has been deprecated or removed, a dead or gated download, a checkpoint or data file absent, an out-of-memory or out-of-scale model, or a metric that never appears in the log. These reasons are grouped into the taxonomy of Section 5.2. Because the classifier reads the log rather than a human, the taxonomy is produced automatically and scales with the corpus.

A failure, end to end. One repository makes the pipeline concrete. For a bottom-up human-pose-estimation model, the engine extracted a genuinely concrete evaluation recipe from the repository documentation: a setup step that downloads the released checkpoint from a URL, followed by the toolbox's `tools/test.py` evaluation command reporting mean average precision. Phase 2 cloned the repository, built an isolated environment, and ran the recipe; the environment resolved cleanly, but the checkpoint URL returned HTTP 404, so the command produced no metric. Reading the log, the classifier assigned the outcome *dead download* and deleted the clone, leaving only the one-line record that places the study in Table 3. The episode is representative: extraction succeeded and execution began, and a single broken artifact, not a compute limit, ended the run.

4. Study Design

Corpus. We use the 96-repository corpus released with SRAF [4, 5]: 96 repositories with associated papers, of which 95 have a retrievable paper PDF. The archived record [5] is titled by the paper count; all counts in this paper are of repositories. Of these, only 26 build successfully under the version-aware protocol and are eligible for result-level reproduction. The other 70 cannot even be built: their declared environments do not resolve, or the repository and paper simply omit the dependency, version, or setup information needed to reconstruct them, so there is no runnable artifact from which a result could be reproduced. Result-level reproduction therefore presupposes a working build and applies only to the 26 that clear this environment-reconstruction floor (established in the SRAF study [4]); it is not meaningful to ask whether a repository reproduces its number when the repository will not run at all. Within the 26 we mark as pre-skipped those whose headline metric is not reproducible by construction (a production A/B number, a hardware-dependent throughput, a metric requiring an external device), or whose linked repository does not match the paper, leaving 17 that we attempt.

Positive controls. A sweep that reproduces nothing is ambiguous: it could mean the artifacts do not reproduce, or that the engine cannot reproduce anything. To separate these, we assemble a small *validation set* of three artifacts that are known to be reproducible and satisfy the engine's constraints (a released checkpoint, small auto-downloading data, an exact evaluation, and a single GPU): two image classifiers on CIFAR-10 and one BERT text classifier on GLUE SST-2. This set is kept strictly separate from the corpus; it validates the instrument rather than contributing to the corpus statistic. We run each control under two conditions: with the *autonomously extracted* recipe, and with a *gold* recipe (a correct, hand-written `rdip:ExecutionRecipe`). The contrast isolates recipe extraction from execution.

Outcome and tolerance. For each attempted study we record the point in the pipeline it reaches, build $\rightarrow$ recipe $\rightarrow$ ran $\rightarrow$ reproduced, and, if it fails, its taxonomy reason. Obtained metrics are parsed from the run log and compared to the extracted claim within a tolerance of 5% relative or 0.01 absolute, whichever is satisfied. We apply a single tolerance to every metric rather than one tuned per metric, because the corpus spans accuracies, mAP, mIoU, panoptic quality and perplexity, and per-metric thresholds would embed a judgement about each community's notion of agreement that we are not

Table 1
Reproduction funnel over the corpus.

Stage	Repositories
Corpus repositories (with code)	96
Build successfully (eligible)	26
Attempted (not pre-skipped)	17
Execution recipe extracted	13
Actually ran and produced a number	0
Reproduced the headline number	0

positioned to make; a uniform band keeps the criterion auditable and identical across studies. The absolute floor exists because a relative test is unstable near zero: a claimed error rate of 0.02 against an obtained 0.03 is a 50% relative gap but a negligible absolute one. A tolerance, rather than exact equality, is essential: a re-run rarely matches a published number bit-for-bit, because a different GPU and CUDA/cuDNN stack, drifting library versions, nondeterministic GPU kernels, and any unset random seed all perturb low-level floating-point results and data ordering. Such deviations are small and unsigned (a re-run may land slightly above or below the claim), and are treated as reproduction; a systematic or large gap is not. Because the claim is itself the product of imperfect extraction (Section 6), we report reproduction against the extracted claim rather than asserting that a paper is irreproducible.

5. Results

5.1. The Reproduction Funnel

Table 1 traces the corpus through the pipeline. Of 96 repositories, 26 build, 17 are attempted, and 13 yield an execution recipe, but *none* run to a matching number: automatic result-level reproduction succeeds for 0 of 17. The attrition is informative in itself. The drop from 96 to 26 is the environment-reconstruction floor established by SRAF; the drop from 17 attempted to 13 with a recipe already reflects the extraction difficulty this paper isolates; and the drop from 13 to 0 is where the artifacts' deeper gaps bite.

Recipe *presence* in the funnel masks recipe *quality*, which Table 2 summarises. The model returned a run command for 13 of the 17 attempted repositories, but only six commands were concrete; the other seven embedded an unresolved placeholder, and four repositories yielded no command at all. It filled the surrounding fields more readily, an entry point and a dataset for fourteen studies and a target metric for fifteen, but a required checkpoint for only two, mirroring how rarely repositories state where their weights come from. Autonomous extraction thus produces a syntactically complete recipe far more often than a runnable one. The model's `recipeConfidence` tracks this well: all six concrete recipes are rated *high*, and all nine *low*-confidence recipes are non-runnable (a placeholder or no command), with only two placeholders over-rated, so the self-report is a usable filter rather than noise. Tellingly, even the six concrete commands did not reproduce a number, each halted downstream by a gated or dead dataset, an undeclared dependency, or environment rot, so the corpus stresses *both* halves of the pipeline; the positive-control ablation (Section 5.3) then isolates them, showing that when the recipe is correct the execution half succeeds.

5.2. A Taxonomy of Reproduction Failure

Table 3 groups every corpus repository by what blocks reproduction, and the shape of this distribution is the paper's central finding: reproduction fails overwhelmingly because the artifact does not carry a runnable, correct procedure, not because reproduction is computationally out of reach. Because the classifier reads each run log rather than a human, the taxonomy is produced automatically and scales with the corpus; because most of its mass sits in one family, it says something concrete about where effort should go.

Table 2
Quality of the extracted recipes (17 attempted studies).

Recipe property	Studies
Run command present	13
concrete (no placeholder)	6
placeholder only	7
No command extracted	4
Entry point identified	14
Dataset identified	14
Checkpoint identified	2
Metric identified	15
Confidence high / low	8 / 9

Eighteen of the twenty-six blockers are *documentation and artifact gaps*. The single largest category is a *placeholder command* (seven repositories): the README gives an evaluation command but leaves a slot for the reader to fill in (`/path/to/checkpoint`, `$config`), so a recipe is present in form but not executable. Four repositories document *no run command* at all, offering training or library code with no stated way to reproduce the reported number. Three link a repository that *does not match the paper*, in one case an unrelated command-line utility, in another a data-repository platform's own source rather than the dataset it hosts. The remaining four are individual breakdowns: a *dead checkpoint link* (a 404), a *gated dataset* released only on application, a model far *too large for a single GPU* (a 176B-parameter model), and a study reporting *no numeric result* to check. What unites this family is that each failure is a single missing or broken element of metadata that an author could, in principle, supply.

A second and smaller family covers metrics that are *not reproducible by construction* (four repositories): a hardware-dependent throughput or processing time, a production A/B gain measured on a live system, and a task success rate that needs a phone emulator and an external service. No single-node harness can regenerate these, so they are excluded by design rather than counted against the artifact. The third family, *our environment* (four repositories), collects failures caused by our harness rather than the artifact, a dependency we did not pre-install (a deep-learning library, or MuJoCo); we isolate it as a threat to validity, and provisioning a generic archive extractor already reclassified two studies to their true artifact-side blockers, a gated dataset and an undeclared dependency. Finally, one time-dependent artifact-side mode deserves separate naming: *environment rot*, where a dependency a study relied on has since been deprecated or removed (an API dropped from a library), so the stack no longer reconstructs even though it once did. This is the software analogue of checkpoint link rot, distinct from the *our-environment* family in that the software no longer exists anywhere; our positive-control set exhibits it directly (Section 5.3).

Read together, the taxonomy is more than a list of errors: it is a map of *where reproduction breaks and what would repair each break*. Its mass in the documentation-and-artifact family carries a constructive message, most lost reproductions in this corpus are recoverable with modest, well-specified author effort rather than blocked by compute or fundamental difficulty. Section 6 turns each family into a specific remedy.

5.3. Positive-Control Ablation: Extraction vs. Execution

The corpus result raises the obvious question of whether the engine could reproduce *anything*. Table 4 answers it on the three known-reproducible controls. With autonomously extracted recipes, the engine reproduces $0/3$: it produces a placeholder for a hub-based repository with no eval script, misses a required dependency for a repository whose weights need a download step, and fails to locate an evaluation command in a model card that never states one, exactly the documentation gaps the corpus taxonomy names. Given *correct* recipes, the same execution-and-comparison pipeline reproduces two of the three, recovering the published CIFAR-10 accuracy (94.25 against a claimed 94.37) and the SST-2 accuracy (93.23 against 92.70) within tolerance. The third control failed for a telling reason: its code calls

Table 3
Failure taxonomy, in three families (26 buildable repositories).

Blocker	Count
Documentation / artifact gaps	
Placeholder command left in the repository	7
No run command documented	4
Repository does not match the paper	3
Dead / gated checkpoint link (link rot)	1
Required dataset unavailable / access-gated	1
Model too large for one GPU	1
No numeric result to check	1
Metric not reproducible by construction	
Hardware-dependent throughput / processing time	2
Production / online business metric	1
Requires external device + API	1
Our environment / tooling (threat to validity)	
Undeclared dependency not auto-installed (torch, pip)	3
Manual-install system dependency (MuJoCo)	1

Table 4
Positive-control ablation (*blocked by a deprecated library API, i.e. environment rot).

Control (task)	Claimed	Extracted recipe	Gold recipe	Obtained
CIFAR-10 classifier A	94.37	placeholder recipe	reproduced	94.25
CIFAR-10 classifier B	93.07	missing dependency	failed*	–
BERT SST-2 (GLUE)	92.70	no command in card	reproduced	93.23

an API since removed from its deep-learning library (`pytorch_lightning.metrics`), so its stack no longer reconstructs without pinning a long-deprecated release, a concrete instance of the *environment rot* discussed above. The two controls whose environments still build both reproduced, which is what matters here: the execution and comparison machinery is sound, and the remaining bottleneck is autonomously extracting a runnable recipe from heterogeneous artifacts (the $0/3$ autonomous result), not running one.

6. Discussion and Conclusion

The taxonomy is not only a description of failure but a map of remedies, and the remedies fall into three distinct kinds. Some failures need only an *author-side documentation fix*: a placeholder command, an undocumented run command, or a repository mislinked to its paper could be resolved by the author's supplying the one missing element, precisely the metadata the SRAF FAIR-R rubric already scores. Each such failure therefore maps to a concrete field an author could add before submission. A related, time-dependent failure, *environment rot*, is preventable by exactly the metadata SRAF rewards but authors rarely supply: an archived container-image digest freezes the software stack, so a later-deprecated library cannot silently break a build that once worked. Other failures need a *stronger extraction agent*: the positive-control ablation shows that when a correct recipe exists but is implicit, spread across a hub call, a weight-download step, or a model card, current open-weight extraction cannot assemble it. This is the primary research gap our study identifies, and it is squarely a knowledge-extraction problem: turning heterogeneous, semi-structured artifact documentation into a single executable recipe. A third group is *genuinely hard or out of scope*: metrics that are hardware-dependent, online, or device-bound cannot be reproduced on any single-node harness and should be excluded by design rather than counted as failures. Separating these three is, we argue, the practical contribution of a result-level taxonomy: it tells a community which failures are cheap to fix, which require research, and which are intrinsic.

What the failures imply for recipe extraction. The two largest categories are not the same problem, and treating them as one overstates what a better extractor could achieve. In the seven *placeholder* cases the model did not fail to *find* a command: a command was found, and left with an unresolved slot. The task those cases pose is therefore slot resolution under a paper-specific constraint, choosing which configuration or checkpoint corresponds to *this* paper's reported number, and it is hardest precisely where a repository is a multi-method toolbox offering many equally plausible configurations. That is a retrieval-and-grounding problem against the paper, not a text-generation problem, and it is the case our prompt already tries and fails to handle (Section 3.2). The four *no-command* cases are qualitatively different: the documentation states no way to reproduce the number, so no extractor can recover what was never written, and improvement must come from the artifact rather than the model. The remaining cases are different again, and instructive because extraction in fact *succeeded*: the recipe was complete in form but named a resource that could not be obtained, a checkpoint URL that returned 404, a dataset released only on application, or a claim with no numeric value to match. No improvement in extraction quality would have changed those outcomes, because the emitted object was correct. What they argue for instead is verification at emit time: a declared URL can be resolved, a dataset's access condition inspected, and a claimed metric checked for a numeric value, all statically and before any compute is spent. Folding such checks into the extractor would let `recipeConfidence` express not only how sure the model is of what it read, but whether what it read still exists, which is the difference between a recipe that looks runnable and one that is. A third observation is more encouraging: the self-reported confidence is already a usable filter rather than noise (Section 5.1). That suggests a cheaper architecture than making a single pass accurate, namely extract, self-assess, and escalate only the low-confidence minority to a stronger model or to a human, which is the design we intend to evaluate next.

What the results imply for artifacts. Read as a specification rather than as a data structure, the seven recipe fields are close to a minimal statement of what an artifact must say for its headline number to be recoverable: a run command with no unfilled slot, the entry point it invokes, the setup steps in order, the datasets and checkpoints it needs at locations that still resolve, and the metric it reports. Every failure in the documentation-and-artifact family of Table 3 corresponds to one of these fields being absent, ambiguous, or stale, which is why that family is both the largest and the cheapest to fix. This reframes the usual artifact-availability question. Checklists and badges predominantly ask whether code and data *exist*; our results suggest the more discriminating question is whether the run procedure is *stated in a form that resolves*, which, unlike a narrative claim of availability, a venue could check automatically at submission time. An author who fills these seven fields has not merely documented their work better; they have supplied the object an engine needs, and the burden of reconstructing it from prose disappears.

Four threats bound our conclusions. First, the claimed numbers are themselves extracted by an imperfect pipeline, so we report reproduction against the *extracted* claim and avoid asserting that any specific paper is irreproducible. It is worth being precise about what this does and does not affect here. No attempted study reached the comparison stage (Table 1: 13 recipes extracted, 0 ran to a number), so *no claimed value was ever compared against an obtained one*. Every corpus failure occurred strictly before comparison, at extraction, environment reconstruction, or artifact retrieval. Claim-extraction error therefore bounds the interpretation of any future *reproduced* or *mismatch* verdict, but it cannot explain the 0 of 17 reported here. The only comparisons actually performed are in the positive-control set, whose claimed values are the artifacts' own documented headline numbers. Second, a small number of failures were caused by *our* environment (a missing system tool, an install quirk) rather than the artifact; we isolate these as a separate taxonomy family and have removed the ones we could, but they caution against reading every failure as artifact-side. Third, the attempted set is small (17 repositories) and the positive-control set smaller still (three artifacts); the funnel and taxonomy should be read as a first, deliberately conservative measurement of a rarely measured quantity, not as a population estimate. Fourth, recipe extraction uses a single open-weight model, so the conclusion that extraction is the bottleneck rests on one extractor. Indirect evidence suggests the limitation is not specific to that model:

the companion SRAF study benchmarks three open-weight models spanning 8B to 24B parameters and two model families over the same corpus, and entity-level extraction F1 remains low and tightly clustered across all three (0.230, 0.223, 0.273), improving only mildly with size [4]. That benchmark measures entity extraction rather than recipe extraction, so it is suggestive rather than conclusive. A direct comparison of recipe extraction across several open-weight and hosted models is the most immediate extension of this work. The engine-driven design means all four threats shrink as extraction improves and the corpus grows.

Building a repository is not reproducing a paper. We presented an engine-driven method that makes the run procedure itself an extracted, machine-readable object, `rdip:ExecutionRecipe`, and a streaming pipeline that rebuilds, runs, compares, and discards each artifact automatically. Over 96 machine-learning repositories, none reproduced their headline number without human help, and the failures form a compact taxonomy dominated by documentation and artifact gaps rather than compute limits. A positive-control ablation shows the execution pipeline is sound and localises the bottleneck to autonomous recipe extraction from heterogeneous artifacts, the concrete research target this work identifies. As a dynamic companion to the static SRAF instrument, the method turns "does it reproduce the number" from a manual, one-artifact-at-a-time judgement into an automatic, corpus-scale measurement.

Data and Software Availability.

The corpus and per-study graphs are archived on Zenodo [5]; the RDIP profile [6] and the SRAF instrument [4] are described separately. The execution-recipe extractor, the two-phase reproduction pipeline, and the failure classifier are released as open source at https://github.com/open-rdip/rdip-sraf-repro, together with the extraction prompts, the gold recipes and positive-control set, the study manifest, and the per-study run logs behind the taxonomy.

Declaration on Generative AI

During the preparation of this work, the authors utilized generative AI tools for grammatical refinement, language polishing, and code assistance. Subsequent to the use of these tools, the authors thoroughly reviewed, verified, and edited all output, accepting full responsibility for the final content of the publication.

References

[1] J. Pineau, P. Vincent-Lamarre, K. Sinha, V. Larivière, A. Beygelzimer, F. d'Alché Buc, E. Fox, H. Larochelle, Improving reproducibility in machine learning research (a report from the NeurIPS 2019 reproducibility program), Journal of Machine Learning Research 22 (2021) 1–20.

[2] O. E. Gundersen, S. Kjensmo, State of the art: Reproducibility in artificial intelligence, in: Proc. AAAI Conf. on Artificial Intelligence, 2018. doi:10.1609/aaai.v32i1.11503.

[3] Association for Computing Machinery, Artifact review and badging — version 1.1, https://www.acm.org/publications/policies/artifact-review-and-badging-current, 2020.

[4] S. Acharya, C. Anutariya, Y. Minamiyama, H. Takeda, V. Wuwongse, SRAF: An ontology-grounded knowledge graph framework for auditing the reproducibility of machine-learning research, 2026. Accepted.

[5] S. Acharya, C. Anutariya, Y. Minamiyama, H. Takeda, V. Wuwongse, A reproducibility corpus of 96 machine-learning papers with GitHub repositories, 2026. doi:10.5281/zenodo.19919042.

[6] S. Acharya, C. Anutariya, Y. Minamiyama, H. Takeda, V. Wuwongse, Enabling FAIR research lifecycle provenance: The RDIP project-centric integration profile, in: Proc. Int. Conf. on Theory and Practice of Digital Libraries (TPDL), 2026.

[7] M. Baker, 1,500 scientists lift the lid on reproducibility, Nature 533 (2016) 452–454. URL: https://www.nature.com/articles/533452a. doi:10.1038/533452a.

[8] V. Stodden, J. Seiler, Z. Ma, An empirical analysis of journal policy effectiveness for computational reproducibility, Proceedings of the National Academy of Sciences 115 (2018) 2584–2589. doi:10.1073/pnas.1708290115.

[9] E. Raff, A step toward quantifying independently reproducible machine learning research, in: Advances in Neural Information Processing Systems (NeurIPS), 2019.

[10] ReScience C, ReScience C: A Journal for Reproducible Replications in Computational Science, https://rescience.github.io/, 2025. Peer-reviewed replications; artifacts archived on Zenodo with DOIs.

[11] J. Dagdelen, A. Dunn, S. Lee, M. d'Avezac, A. Jain, G. Ceder, K. A. Persson, Structured information extraction from scientific text with large language models, Nature Communications 15 (2024) 1494. URL: https://www.nature.com/articles/s41467-024-45563-x. doi:10.1038/s41467-024-45563-x.

[12] D. Xu, X. Zhang, W. Chen, Y. Zhang, et al., Large language models for generative information extraction, Frontiers of Computer Science (2024). URL: https://link.springer.com/article/10.1007/s11704-024-40555-y. doi:10.1007/s11704-024-40555-y.

[13] H. Bian, X. Zhao, W. Zhang, et al., Llm-empowered knowledge graph construction: A survey, 2025. URL: https://arxiv.org/abs/2510.20345. arXiv:2510.20345.

[14] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, K. Narasimhan, SWE-bench: Can language models resolve real-world GitHub issues?, arXiv preprint arXiv:2310.06770 (2024).

[15] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, O. Press, SWE-agent: Agent-computer interfaces enable automated software engineering, arXiv preprint arXiv:2405.15793 (2024).