STAMPED properties for reproducible research objects

Austin Macdonald, Cody C. Baker, Yaroslav Halchenko, Isaac To

March 2025

1 Abstract

2 Introduction

Managing data in modern scientific research presents unprecedented challenges due to rapidly expanding dataset scale, complexity, and interdependency. The FAIR (Findable, Accessible, Interoperable, Reusable) principles [cite] and FAIR4RS guidelines [cite] for research software provide structured guidance to ensure that digital objects are managed in ways that support reuse and reproducibility. Here we adopt the term **research object** to be a collection of data, code, and metadata that together represent a complete unit of research output. Reproducibility of a research object's results requires specific versions of data, code, and computational environments, as well as detailed provenance of how they were used together. Currently it is most common for these aspects to be managed separately; this separation introduces complexity and undermines reproducibility, transparency, efficiency, and reliability.

The Workflow Community Initiative's FAIR Workflows Working Group (WCI-FW) [cite] defines a **workflow** as "the formal specification of the data flow and execution control between executable components, the expected datasets, and parameter files." WCI-FW systematically mapped existing FAIR and FAIR4RS principles onto computational workflows, defining metrics for workflow FAIRness and recommendations for workflow developers and systems. However, they explicitly chose not to define new principles for workflows, instead treating them as hybrid data/software objects and applying established FAIR guidelines directly. FAIR addresses findability, accessibility, interoperability, and reuse of digital objects, but does not address the operational properties—self-containment, provenance tracking, portability—that make a research object's results actually reproducible.

Enhanced rigor, reproducibility, re-usability, efficiency, and translation across domains are all aspirational goals. Many researchers already employ practices toward these goals, but lack a common framework to evaluate or communicate their approach. Here we identify and codify these practices as the STAMPED

properties (Self-contained, Tracked, Actionable, Modular, Portable, Ephemeral, and Distributable), originating from the YODA Principles, which characterize a well-formed research object. STAMPED takes a pragmatic approach toward these goals in which each property is a spectrum, from practical minimums—often what researchers are already doing—to aspirational ideals. STAMPED complements formal computational workflow guidelines (e.g., FAIR4RS, RO-Crate) and facilitates the transition toward comprehensive workflow automation.

3 Comments

Table 1: The STAMPED Properties. Each property addresses a distinct dimension of research object organization that contributes to reproducibility. Together, they provide a framework for evaluating and improving the reproducibility of computational research.

Letter	Principle	Description
S	Self-containment	A research object is a complete retrieval unit—it can be obtained and understood in its intended scope without needing to reference external resources.
T	Tracking	The state and provenance of all components is recorded.
A	Actionability	Research object contains machine-actionable information to carry out procedures to obtain or reproduce content.
M	Modularity	All modules are independent and composable.
P	Portability	Research object could be moved to different environments while retaining its STAMPED properties.
E	Ephemerality	Procedural execution is performed within a throwaway environment.
D	Distributability	All modules and procedures are shareable externally in a persistent state.

The items containing keywords "MUST", "SHALL", "SHOULD", and "MAY" are to be interpreted as described in RFC 2119.

3.1 Self-contained

- **S.1**: All modules and components essential to replicate computational execution MUST be reachable within a single top-level research object.

Reproducibility fails when a research object depends on resources that are not part of it—an undocumented library, a dataset referenced by a broken URL, a script that assumes tools are installed on the host system. We call this the "don't look up" rule: a research object must never rely on implicit external state. Instead, it must be a complete retrieval unit, where all modules and

components essential to reproduce its results are contained within a single top-level boundary (S.1, Table 1).

Components may be included literally (e.g., files committed directly) or by reference (e.g., as subdatasets, registered data URLs). Both approaches satisfy self-containment, provided the references are explicit and part of the research object. At a minimum, everything needed is gathered under one root and any external dependency is explicitly documented—ensuring that nothing is implicitly assumed about the host system, a concern further addressed by Portability. At the ideal end, every reference is precise enough that retrieval is unambiguous; how to achieve that precision is the concern of Tracking (T) and Distributability (D).

Self-containment is the foundational property upon which the remaining STAMPED properties build. Tracking, Modularity, Portability, and Distributability each address a different aspect of how that boundary is organized, versioned, and shared—but none are meaningful without first establishing what is inside it.

3.2 Tracked

- **T.1**: Persistent content identification MUST be recorded for all components.
- **T.2**: All components SHOULD be tracked using the same content-addressed version control system.
- **T.3**: Provenance of all modifications MUST be recorded.
- **T.4**: Code-driven provenance SHOULD be recorded programmatically and MUST include the versions of all components involved.

Research data frequently exists outside the scope of established version control practices, leading to a multitude of reproducibility issues. Non-version-controlled data creates ambiguity, as precise identification of dataset states is challenging, and historical changes are difficult to reconstruct or verify. This ambiguity undermines trust, complicates collaboration, and impairs reproducibility, particularly when datasets undergo frequent updates or transformations.

Version control systems (VCS), such as git, directly address these issues by associating each revision with a unique identifier and recording basic provenance information, including who made the changes, when, and why. Although commonly called "version control systems," the primary value is not version numbering ("v1" vs "v2") but *content-addressed identification*. Git describes itself as "the stupid content tracker" (from `man git`), using cryptographic checksums as content identifiers. Explicit version tags (v1.0, v2.3) are convenience labels atop this content-tracking foundation. This distinction matters: reproducibility requires precise content identification, not just semantic versioning. Two datasets labeled "version 1.0" by different labs are ambiguous; two datasets with identical content hashes are provably identical. Throughout the manuscript we will

refer to such content tracking systems as VCS to use terminology familiar to the reader, although some initiatives such as [**?**] would describe such requirement as data consistency.

By applying VCS systematically to data and analysis outputs, researchers can pinpoint exact states of digital files, reliably reverting or exploring previous versions as needed. Although traditional version control systems have limitations when managing large datasets due to storage inefficiencies, modern tools such as git-annex, DataLad, or DVC effectively extend git functionality to handle large, binary, and distributed datasets. These tools enable researchers to manage extensive datasets without losing the benefits of traditional version control, including improved collaboration through explicit tracking of changes, efficient reversion of undesired modifications, and the ability to reconstruct past research states—effectively enabling "time travel."

Critically, approaching full reproducibility requires that version control encompasses all components essential for replicating computational workflows. These components typically include raw and derived datasets, preprocessing and analysis scripts, parameter and configuration files, computational environment definitions (such as Docker or Singularity containers), and accompanying documentation describing workflow execution details. Omitting even one component can severely compromise reproducibility, making comprehensive version control across these assets a fundamental necessity.

Tracking encompasses not only version history but also provenance—the record of what actions produced or modified each component. Provenance collection typically occurs separately from version control, leading to fragmentation: the data is versioned in one system while the record of how it was produced lives in another (or nowhere). Embedding provenance directly into version control history—for example, by encoding descriptions of computational actions into commit records—unifies these concerns. A provenance-rich commit history records not just that a file changed, but what command was executed, what inputs it consumed, and what versions of code and environment were involved.

For code-driven modifications, provenance should be captured programmatically rather than by manual annotation. This ensures completeness and enables re-execution: a researcher can not only see what was done but repeat it.

3.3 Actionable

- **A.1**: Research object MUST contain sufficient instructions to reproduce all computational results.
- **A.2**: Procedures SHOULD be specified as executable specifications.

A research object may be self-contained and fully tracked, yet having all ingredients and a record of what was done is insufficient without a recipe that can be followed. Requirement A.1 is what makes a research object operationally useful—the bridge between having the components and being able to use them.

At a minimum, a research object must contain documentation thorough enough for another researcher to follow every step to reproduce results. Actionability applies to every other STAMPED dimension: at the ideal end, each property is enacted not through documentation alone but through executable specifications (A.2):

- **Self-containment** is more actionable when retrieval of all components is specified and executable (e.g., `git clone`, `datalad install`), not just listed in a manifest.
- **Tracking** is more actionable when provenance records are re-executable (e.g., `datalad rerun`), not just inspectable.
- **Modularity** is more actionable when components can be composed and decomposed via tooling (e.g., `git submodule`), not just organized into directories.
- **Portability** is more actionable when environments can be instantiated from a specification (e.g., `singularity run`), not just documented in a README.
- **Ephemerality** is more actionable when computation can be orchestrated in disposable environments (e.g., `docker compose`, Slurm), not just instructed to "run in a clean environment."
- **Distributability** is more actionable when a frozen state can be produced and retrieved by others (e.g., containers, `npm ci`), not just described with pinning instructions.

The principle is tool-agnostic: any system that moves a property from documented to executable moves the research object further along the actionability spectrum. With advent of agentic AI systems, documentation alone often can provide actionable recipes, but nevertheless formalization in AI-specific formulations (e.g. `Claude Skills` [CITE]) can drastically improve actionability of such descriptions.

3.4 Modularity

- **M.1**: Components SHOULD be organized in a modular structure.
- **M.2**: Components MAY be included directly or linked as subdatasets.

Monolithic structures present significant barriers to effective development, maintenance, and reuse. Managing such research objects as single, large units complicates incremental updates, obscures dependencies, remixing of components, and can be notably time-consuming. So rather than managing the research object as one indivisible whole, STAMPED promotes a compositional approach, structuring projects as assemblies of independently versioned and well-defined components. Defining a module as a separately distributable collection

of components for our research object, **Modularity** is achieved when all modules of the research are independent and composable. Individual parts—such as input and reference datasets, processing scripts, and computational environments—can be updated or replaced independently, thus minimizing disruption and maximizing reusability.

The layout of these components plays a crucial role in enabling effective modularity:

- **code/**: containing analysis scripts and tests.
- **inputs/**: encapsulating raw or intermediate datasets.
- **envs/**: defining computational environments (e.g., Singularity images).
- **docs/**: for project documentation.
- **results/**: explicitly identified outputs (e.g., figures, tables, derived datasets).

The intention of such organization clarifies for researchers how to compose and recombine modules flexibly to produce varied outcomes.

The spectrum of modularity ranges from: At **minimum**, organize components into logical subdirectories—separate code from data, distinguish inputs from results, and document structure in a README. At a **middle** level, formalize module boundaries using version control repository boundaries, with major components as separate repositories referencing each other via submodules or documented URLs with version tags. At the **ideal** level, implement fully actionable composition with all dependencies as versioned submodules with automated module installation, allowing the entire workflow to be reassembled from a self-contained specification.

Modularity ought to rely heavily on Tracking (module boundaries require independent versioning) and Self-containment (all independent modules required by the workflow). It enhances Actionability by increasing the system reliability. Portability and Distributability are also easier to achieve since modules can themselves be ported and distributed separately from the integration in the whole.

3.5 Portability

- **P.1**: Procedures MUST NOT depend on undocumented host environment state.
- **P.2**: Computational environments MUST be explicitly specified.
- **P.3**: Environment definitions MUST be version controlled.

A research object that is self-contained, tracked, and modular may still fail to reproduce if it depends on undocumented host environment state—hardcoded

paths, implicitly available tools, or specific OS configurations. Portability requires that procedures can be executed on different host environments, given documented system requirements.

Research reproducibility fundamentally depends not only on preserving code and data, but also on capturing the complete computational environment—the operating system, libraries, dependencies, and tool versions—that produced the results. Environmental drift, where software dependencies evolve or disappear over time, is a primary cause of computational irreproducibility. Even with identical code and data, differences in library versions, compiler settings, or system configurations can produce divergent results or outright failures.

Computational environments should be explicitly defined (not implicitly assumed), machine-reproducible, version controlled alongside code and data, and self-contained within the project boundary.

Two primary approaches exist for environment portability:

1. **Container-based**: Docker, Singularity/Apptainer provide OS-level isolation.
2. **Package-based**: Nix, Guix provide declarative, reproducible package management.

STAMPED is environment-mechanism-agnostic; what matters is that environments are explicitly specified, versioned, and available within the project.

3.6 Ephemeral

- **E.1**: Computational results SHOULD be produced in ephemeral environments.

A research object may satisfy Self-containment, Tracking, and Portability while its computational environment has only ever been assembled once, incrementally, on the original researcher's machine. Such one-off setup is fragile: specifications that have never been tested by reconstitution can be subtly incomplete—for instance, through hidden dependencies on the researcher's own system that are not apparent until someone else attempts to reproduce the work. Ephemerality addresses this: when computing in a temporary, disposable environment built from tracked specifications, researchers validate and exercise Self-containment, Portability, and Actionability—all dependencies must be within the boundary, specifications must be sufficient to reconstitute the environment, and setup and computational procedures become machine-executable steps run end-to-end.

Ephemeral computation does not validate everything—for instance, an ephemeral environment with network access may successfully fetch unpinned dependencies at build time, masking a self-containment gap that only surfaces when those resources move or disappear—but it substantially raises confidence that the research object is self-contained, portable, and actionable. The spectrum of Ephemerality ranges from producing results from a fresh clone on a system that

meets stated requirements (P), to an ideal in which every computation runs in a disposable environment created and destroyed per execution. The FAIRly big workflow approach and platforms such as Code Ocean operationalize the ideal end of this spectrum, treating each computation as an ephemeral work unit—which also enables scaling, as independent disposable instances can be parallelized across subjects, parameters, or datasets.

3.7 Distributable

- **D.1**: All referenced modules and components MUST be persistently retrievable by others.
- **D.2**: Environment specifications SHOULD support reproducible builds.

Self-containment (S) establishes that everything needed is within the research object's boundary, possibly by reference. Distributability promises that those references actually deliver—that the research object and its components can be shared, retrieved, and used by others in a state consistent with reuse.

The distinction mirrors the concept of a software distribution: a curated, versioned bundle in which all components are resolved to specific versions and packaged for consumption. Simply sharing a research object—uploading scripts to a repository with loose dependencies, or posting files on a website with no version guarantees—does not constitute distribution in this sense. A distributable research object is packaged so that it can be retrieved in the same state as intended

Distributability also has a circular relationship with the other STAMPED properties: someone else's distribution effort often serves as the starting point for a new research object's self-containment. Researchers routinely begin by downloading containers, fetching published datasets, and installing released software—modularly composing the distributions of others to assemble their own self-contained research objects.

The spectrum of distributability ranges from publicly accessible components with retrieval instructions, through persistent hosting on archival infrastructure (Zenodo, PyPI, conda-forge, DANDI) with frozen versions and content-addressed identifiers—analogous to moving from a development checkout (`pip install -e .`) to a pinned release (`pip freeze`)—to an ideal in which all external dependencies are mirrored or the entire research object is packaged as a self-contained archive (e.g., a built container or a zipped RO-Crate).

3.8 Checklist for compliance to principles

We hope that the preceding sections have provided a clear understanding of the STAMPED properties and their rationale. While this understanding is essential, researchers may also benefit from a practical checklist to assess their research objects against these principles. This checklist provides such a guide for assessing compliance with STAMPED principles, ordered by the strength of the requirement (MUST, SHOULD, MAY).

MUST

- ▷ **Self-containment (S.1)**: All modules and components essential to replicate computational execution MUST be reachable within a single top-level research object.
  - ☐ Are all files and directories nested under a common root?
  - ☐ Are datasets included, linked, or referenced reachable from the code and environment specifications without cross the boundary of a common root?
  - ☐ Is external software included in the environment or linked as submodules within the project boundary?
- ▷ **Tracking (T.1)+(T.3)**: Persistent content identification MUST be recorded for all components. Provenance of all modifications MUST be recordeed.
  - ☐ Are version control systems such as `Git` used for code, text, documentation, and configuration files?
  - ☐ Are version control systems such as `git-annex`, DataLad, or `Git` LFS used for large binary data?
  - ☐ Are the exact environment specifications used to generate a set of results included in the provenance records to link computational actions to a particular environments?
- ▷ **Actionability (A.1)**: The research object MUST contain sufficient instructions to reproduce all computational results.
  - ☐ Is a `README.md` or `Makefile` included with instructions for installation and usage?
  - ☐ Is there a clear starting point for users to start reproducing results (e.g., a main script, a workflow definition, or a container image)?
- ▷ **Portability (P.1)**: Procedures MUST NOT depend on undocumented host environment state.
  - ☐ Are relative paths used in scripts, avoiding hardcoded or system-specific paths like `C:\Users\...` or `/home/user/...`?
  - ☐ Are all software dependencies included in the environment specification, rather than relying on pre-installed tools?
  - ☐ Have all assumptions about the host system's configuration been documented (e.g., specific OS versions, required system libraries, or environment variables)?
- ▷ **Portability (P.2)**: Computational environments MUST be explicitly specified.
  - ☐ Is there a clear list of system requirements and dependencies documented in the README or environment specifications?

- ▷ **Portability (P.3)**: Environment definitions MUST be version controlled.
  - ☐ Are environment specifications (e.g., Dockerfiles, `pyproject.toml`, `package.json`) included in the version control system alongside code and data?
  - ☐ Is there a process for updating environment specifications when dependencies change, and are these updates tracked in version control?
- ▷ **Distributability (D.1)**: Computational environments MUST be persistently retrievable by others.
  - ☐ Are environment specifications (e.g., container digests, frozen package manifests) included to ensure others can attempt to exactly replicate the environment?
  - ☐ Are environment specifications shared in a way that others can access and use them (e.g., published container images, archived environment files)?
  - ☐ Is there documentation on how to obtain and use the environment specifications for reproduction?

SHOULD

- ▷ **Tracking (T.2)**: All components SHOULD be tracked using the same content-addressed version control system.
  - ☐ Is a common version control system (e.g., `Git`, DVC, `git-annex`, or DataLad) used across all components?
- ▷ **Actionability (A.2)**: Procedures SHOULD be specified as executable specifications.
  - ☐ Is the workflow tested regularly to ensure instructions remain accurate?
- ▷ **Modularity (M.1)**: Components SHOULD be organized in a modular structure.
  - ☐ Are raw data, processed data, code, and environment definitions separated into distinct modules?
- ▷ **Ephemerality (E.1)**: Computational results SHOULD be produced in ephemeral environments.
  - ☐ Is the pipeline tested in a fresh container, batch job, clean virtual machine, or cloud-based instance?
  - ☐ Does the workflow spawn disposable environments for each execution, allowing for isolation of runs?

▷ **Distributability (D.2)**: Environment specifications SHOULD support reproducible builds.

- ☐ Are the exact environment specifications used to generate a specific set of results regularly tested to ensure they can be reconstituted as intended?
- ☐ Are environment artifacts archived within the research object where possible (e.g., executable binaries, container images)?
- ☐ Is there a process for updating environment specifications when dependencies change, and are these updates tracked in version control?

MAY

▷ **Modularity (M.2)**: Components MAY be included directly or linked as subdatasets.

- ☐ Are external datasets or software dependencies included as submodules or linked as submodules?
- ☐ Is the modular structure documented to clarify how components relate and can be recombined?
- ☐ Are modular boundaries defined in a way that allows independent updates without breaking the overall workflow?
- ☐ Is there a clear mechanism for composing modules together (e.g., `git` submodules, or container orchestration)?
- ☐ Is there documentation or tooling to support users in understanding and navigating the modular structure?

3.9 Enabling tools

STAMPED properties are complementary, not competitive, with existing standards:

- **STAMPED + BIDS**: Domain standard within STAMPED structure
- **STAMPED + RO-Crate**: Organization during research, packaging for publication
- **STAMPED + FAIR**: Structure projects, then share via FAIR
- **STAMPED + workflow systems**: Organize, then execute (Nextflow, Snakemake, CWL)
- **STAMPED + platforms**: Download, version, and compose locally

3.9.1 git-annex Complexity vs Git LFS Simplicity

Achieving STAMPED compliance with git-annex offers maximum flexibility, supporting many remote types including SSH, S3, HTTP, local drives, and even offline USB drives. This comes at a cost: a steeper learning curve. Git LFS offers a simpler alternative that is transparent to users and has native GitHub/GitLab support with broad adoption. The trade-off: Git LFS is centralized (requires an LFS server), lacks offline capability, and has less flexible remote configurations. Lightweight STAMPED implementations using Git LFS are feasible for easier onboarding, sacrificing federation flexibility.

3.9.2 When STAMPED Is vs Is Not Appropriate

STAMPED excels when:

- Multi-institutional collaboration requires federated datasets
- Long-term reproducibility spans decades or requires offline archives
- Heterogeneous workflows are not tied to a specific platform
- Data sovereignty requirements preclude mandatory cloud upload
- Complex dependency graphs involve subdatasets from diverse sources

STAMPED may be more than needed for:

- Single-script analyses where version control alone suffices
- Ephemeral exploratory work such as Jupyter notebooks
- Cloud-native ML production where Pachyderm or a platform may be a better fit
- Small teams with simple workflows where DVC or Git LFS may suffice

3.10 Examples and Case Studies

Several research projects have already adopted and documented YODA principles—the predecessor to STAMPED—demonstrating practical utility across domains.

BIDS Standard Evolution: A significant validation of the "do not look up" principle occurred within the Brain Imaging Data Structure (BIDS) community. Originally, the BIDS specification nested derived data under `derivatives/` within the original raw dataset, creating upward dependencies that violated portability. Following YODA principles, the BIDS community reversed this relationship: derivative datasets now exist independently and reference raw data as subdatasets, not vice versa.

This architectural shift influenced major neuroimaging tools:

- **fMRIPrep**: Switched default output layout to follow YODA structure

- **OpenNeuroDerivatives**: Entire derivative dataset collection now follows YODA organization, separating processed outputs from raw data dependencies
- **BIDS specification**: Updated to accommodate and recommend YODA-compliant layouts for derivative datasets

Workflow Platforms: Infrastructure projects implementing YODA at scale:

- **BABS**: Platform implementing the "FAIRly big workflow" approach, demonstrating YODA principles for large-scale neuroimaging analyses with containerized pipelines and modular dataset composition
- **CRCNS.org**: Neuroscience data sharing platform providing YODA-structured templates and validation tools

These adoptions demonstrate that STAMPED properties solve practical organizational challenges across scales, from individual studies to community-wide infrastructure.

3.11 Lit Review

The challenges STAMPED addresses are not unique to neuroscience or even scientific computing broadly. Multiple communities independently developed organizational frameworks exhibiting remarkable convergent evolution toward similar core patterns.

Between 2003 and 2025, at least 19 distinct initiatives emerged addressing research data organization, version control, and reproducibility:

Foundational principles: Noble's 2009 guidelines for computational biology organization, FAIR principles (2016) for data sharing, Software Carpentry's "Good Enough Practices" (2017).

Version control extensions: git-annex (2010), Git LFS (2015), DVC (2017), Pachyderm (2014), Quilt (2020s) all independently arrived at "Git for data" concepts, applying proven version control semantics to large datasets.

Cloud platforms: Code Ocean (2017), brainlife.io (2017), Flywheel (2018), Galaxy (2005) provide turnkey reproducibility services, all implementing code/environment/data trinity separation.

Framework tooling: Kedro (2019), nipoppy (2023), Cookiecutter Data Science (2016) provide opinionated structures for data engineering and neuroimaging workflows.

Metadata standards: RO-Crate (2019), BioComputeObject (IEEE 2791-2020), PROV (TODO), BIDS (2016) address packaging and provenance standardization.

Despite independent origins, these frameworks converged on core patterns:

- **Separation of concerns**: Data, code, environment, and results managed distinctly
- **Immutability**: Raw data never modified in-place

- **Hierarchical organization**: Nested, modular structures
- **Version control**: Applying Git concepts beyond just code
- **Provenance tracking**: Recording how outputs were generated

STAMPED's unique contributions within this landscape:

- **Federated composition**: Subdatasets can live anywhere (not centralized)
- **Interface agnosticism**: Works with any container/pipeline system
- **Local-first design**: Works offline, no cloud platform dependency
- **git-annex flexibility**: Many remote types vs single-server models
- **Scale via modularity**: Proven from single files to 8000+ subdatasets (datasets.datalad.org)

This convergent evolution validates that STAMPED properties are not arbitrary choices but responses to fundamental challenges in computational research reproducibility. The formalization presented here provides a foundation for interoperability and comparison across this evolving ecosystem of tools and standards.

3.11.1 DVC (Data Version Control)

Shares "Git for data" philosophy but differs in architecture: DVC uses external .dvc files with remote storage configuration, while tools complementary to STAMPED such as DataLad with git-annex integrate directly into the git repository structure. DVC focuses on Python and ML workflows within a single project, whereas STAMPED is language-agnostic and supports federated multi-project composition. Both are valid; choice depends on team familiarity (DVC easier for ML engineers), infrastructure (DVC integrates easily with cloud ML platforms), and scale requirements (git-annex more flexible for federated datasets).

3.11.2 Pachyderm

Enterprise "Git for data" with Kubernetes-native architecture. Pachyderm requires a centralized cluster and cloud infrastructure, while a STAMPED approach is local-first, works offline, and supports federation. Pachyderm provides automatic pipeline triggering, whereas STAMPED emphasizes explicit, provenance-tracked execution (e.g., `datalad run`). Pachyderm excels for production ML pipelines, team collaboration with shared compute resources, and automatic scaling. STAMPED excels for research workflows, offline work, and heterogeneous datasets across institutions.

3.11.3 Kedro

Python data engineering framework with modular pipelines. Kedro provides within-project modularity through Python pipeline components, while STAMPED provides across-project modularity through subdatasets as git submodules. Kedro includes built-in visualization (Kedro-Viz), whereas STAMPED is CLI-focused with integration to external tools. These approaches are complementary: a Kedro pipeline inside a STAMPED research object combines both strengths.

3.11.4 Cloud Platforms (Code Ocean, brainlife.io, Flywheel)

Turnkey reproducibility services with proprietary interfaces. Platforms offer zero local setup and institutional partnerships; STAMPED offers local control with no vendor lock-in. Platforms provide web GUIs with point-and-click interaction; STAMPED provides CLI/API with scriptable automation. Platforms are centralized (single capsule or project); STAMPED is federated (compose across repositories). STAMPED research objects could wrap platform outputs: download results, organize locally with full versioning, and compose across platforms via subdatasets.

3.12 Future Directions

3.12.1 Provenance Standards Convergence

Ongoing work toward unified provenance representation, including BEP028 and RO-Crate convergence, will improve STAMPED interoperability with domain-specific tools.

3.12.2 Tool Certification

Formal STAMPED properties enable "STAMPED-compliant" tool certification, similar to BIDS validation.

3.12.3 Teaching Materials

Operationalized principles provide a foundation for structured curriculum development beyond current tutorial-based approaches.

4 Data Availability

5 Code Availability

6 References

7 Author Contributions

8 Competing Interests

The authors declare no competing interests.

9 Acknowledgments

10 Funding

This work was supported by the National Institutes of Health: ReproNim — Center for Reproducible Neuroimaging Computation (NIBIB P41 EB019936), OpenNeuro — An Open Archive for Analysis and Sharing of BRAIN Initiative Data (NIMH R24 MH117179), DANDI — Distributed Archives for Neurophysiology Data Integration (NIMH R24 MH117295), and EMBER — Ecosystem for Multi-modal Brain-behavior Experimentation and Research (NIMH R24 MH136632).

Table 2: Normative statements about STAMPED properties of research objects.

Principle	Requirements
To be Self-contained:	• all modules and components needed to understand and execute the Research Object are retrievable as a single unit. • external dependencies are explicitly documented with retrieval instructions. • there are no implicit references to undocumented external resources.
To be Tracked:	• every component has version information (commit hash, tag, or identifier). • changes to components are recorded with timestamps and authorship. • provenance records capture the computational history, context, and transformations.
To be Actionable:	• instructions for executing procedures are present and unambiguous. • execution paths can be followed manually or automated programmatically. • the Research Object transitions from documentation to operational capability.
To be Modular:	• modules can be independently modified. • components are organized in logical, separable units. • modules can be composed together or used in isolation.
To be Portable:	• system requirements and dependencies are explicitly documented. • the Research Object is flexible enough to execute on different host environments without modification by the user. • environment specifications are machine-readable where possible.
To be Ephemeral:	• computation can occur in temporary, disposable environments. • results are reproducible without knowledge of previous runs. • no reliance on external configurations or host system states (such as OS registry modifications).
To be Distributable:	• all modules and components can be shared in a persistent, retrievable state. • dependencies are frozen or pinned to specific versions across systems. • the Research Object can be obtained by others in the same state as intended.