STAMPED properties for reproducible research objects

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

March 2026

1 Abstract

2 Introduction

Data-driven research depends on specific versions of datasets, software, and computational environments, together with a record of how they were combined to produce results. We adopt the term **research object** to denote a collection of data, code, and metadata that together represent the research output as a unit. It is common for the components of a research object to be managed separately—code in one repository, data on a shared drive, environment setup in a wiki page, provenance in a lab notebook or nowhere at all—and this fragmentation undermines rigor, reproducibility, reusability, and efficiency. Even when gathered in one place, research objects are often tightly coupled to specific environments, opaque to outside collaborators, and difficult to reuse or redistribute.

The FAIR principles[1] and FAIR4RS guidelines[2] provide structured guidance for making digital objects findable, accessible, interoperable, and reusable. The Workflow Community Initiative (WCI-FW)[3] mapped these principles onto computational workflows but explicitly chose not to define new ones, instead treating workflows as hybrid data/software objects. [TODO: Relate/compare to FAIR here]

Many researchers already employ practices that address these operational properties—version control, containerized environments, structured project directories, documented analysis steps—but lack a shared vocabulary to name them, evaluate how thoroughly they are applied, and communicate that evaluation to collaborators and reviewers. Here we provide that vocabulary: the STAMPED properties (Self-contained, Tracked, Actionable, Modular, Portable, Ephemeral, and Distributable), originating from the YODA Principles[4], which describe the operational maturity of a research object. Rather than a compliance threshold, each STAMPED property is a spectrum from a practical minimum—often what researchers are already doing—to an aspirational ideal. A researcher who stores everything under one project directory, version-controls

analysis scripts, and documents how to run them is already meeting the practical minimum for Self-containment, Tracking, and Actionability. As complexity grows—multi-institutional collaborations, heterogeneous workflows spanning platforms, long-term reproducibility requirements—the same properties guide researchers toward more rigorous practices without requiring wholesale adoption of new tools. STAMPED provides a vocabulary for describing and a framework for evaluating and incrementally improving the operational properties of research objects toward enhanced rigor, reproducibility, reusability, and efficiency.

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[5].

3.1 Self-containment

- **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 Tracking

- **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 and code change constantly during a project, and each change compounds the difficulty of reconstructing any previous state: which version of a dataset was used for a particular analysis, what parameters were set, what code produced a given figure. Tracking addresses two concerns: the identity of each component—which exact state it was in—and **provenance**, the recorded history of what actions produced or modified it. These are often managed separately—versions in one system, provenance in a lab notebook or nowhere at all—and this fragmentation makes prior work hard to validate.

The spectrum of Tracking begins with content-addressed version control. Rather than relying on semantic version labels, content-addressed systems identify each state by a checksum of its contents. This distinction matters: two datasets labeled "version 1.0" by different labs are ambiguous; two datasets with identical content hashes are provably identical. Version control commits also capture basic provenance—who made the change, when, and a description of why—unifying identity and provenance in a single system. This also provides a safety net: any change can be reverted and any prior state restored, making it safe to modify and experiment. For tracking to be effective, all components

of the research object must be under version control—code, data, environment definitions, and configuration.

Traditional version control is designed for code and other text files, but research objects often include large binary datasets, container images, and other assets that do not fit this model. Compatible tools such as git-annex, DataLad, or DVC extend content-addressed tracking to these components, ensuring that all parts of the research object are managed under the same system.

Toward the ideal end, provenance is captured programmatically by tooling rather than by manual annotation, producing richer records than a human would typically write: each computational step records what command was executed and what inputs it consumed. When modules are independently tracked under compatible systems (M), the version of every module at the time of each commit is also discoverable, providing a complete picture of the state that produced a given result. This enables not just inspection but re-execution.

3.3 Actionability

- **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, updated, and replaced via tooling (e.g., `git submodule`), not just organized into directories.
- **Portability** is more actionable when environment specifications are machine-readable and can be resolved on different hosts (e.g., `conda env create`, Nix flakes), 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.

3.4 Modularity

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

Separation of concerns is a foundational engineering practice: organizing a system so that each piece has a clear, distinct role. With clear boundaries between the conceptual components of a research object, it becomes more navigable and maintainable. We define a **module** as a separately distributable unit of a research object—such as a dataset, a collection of analysis scripts, or a computational environment definition. In practice, module boundaries often align with repository boundaries, but a module may also be a published container, a registered dataset, or any independently versioned unit. Modularity applies this principle to research objects, structuring them as assemblies of modules whose boundaries are explicit. At a minimum, a research object should separate analysis code, input datasets, computational environments, configuration settings, and results into distinct directories. Beyond this, independent versioning of modules enables reuse: a dataset or environment can be updated to a newer version, or swapped for an alternative, without disrupting the rest of the research object. This is possible through manual management but becomes practical when handled by tooling that automates module installation, versioning, and composition. Toward the ideal end, the full research object can be reassembled from its specification through automated, recursive installation of its modules.

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 run on a different machine if it is silently affected by inherited host state—a specific Python installation, a system library, a hardcoded path, or an assumed OS configuration. This coupling is often invisible on the original system and only surfaces when someone else attempts to reproduce the work. Portability requires making all host dependencies explicit, so that the research object can be adapted to a new environment.

Some degree of coupling to the host is unavoidable—path naming restrictions, resource limits, platform-specific flags. The spectrum of Portability begins with isolating this host-specific configuration from the rest of the research object, so that adapting to a new environment, however involved that may be, requires only changing configuration. Beyond this, several complementary approaches reduce host coupling: virtual machines provide full system-level isolation; container-based tools (Docker, Singularity/Apptainer) bundle OS-level environments; and declarative package managers (Nix, Guix) specify environments that can be reconstructed from a definition. Toward the ideal end, the computational environment is fully specified and version controlled within the research object, enabling it to be instantiated on any compatible system.

3.6 Ephemerality

- **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. During development, researchers routinely modify their environment—installing packages, adjusting configurations, setting variables—and each change is individually small but collectively difficult to document. The result is environment drift: an environment that works but may not be reconstructable from its specification alone, where undocumented steps can go unnoticed when internalized knowledge silently fills the gaps. Ephemerality eliminates this problem by construction: a disposable environment built from tracked specifications cannot carry undocumented state, so any successful execution exercises Self-containment, Actionability, and specification completeness.

The spectrum of Ephemerality ranges from manually setting up a fresh environment from the project's specification—sufficient to catch environment drift—to fully automated disposable environments created and destroyed per execution. At the ideal end, ephemeral computation also enables testing across different host systems, where same-host execution validates specification completeness and different-host execution validates that specifications are not coupled to a specific system. The FAIRly big workflow approach[6] and platforms such as Code Ocean operationalize this ideal, treating each computation as an ephemeral work unit—which also enables scaling, as independent disposable instances can be parallelized across subjects, parameters, or datasets. Ephemeral

computation does not validate everything—for instance, an ephemeral environment with network access may fail to notice unpinned dependencies fetched at build time, 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.

3.7 Distributability

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

A research object that is self-contained on the author's machine but cannot be obtained by others in the same state is effectively unreproducible beyond its origin. Self-containment (S) establishes that everything needed is within the boundary and Tracking (T) records the state of each component. Distributability ensures that others can obtain the research object in that same state.

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 begins where FAIR leaves off: publicly accessible components with documented retrieval instructions. Toward the ideal end, several complementary strategies reduce the risk that external changes will break a research object's reproducibility: bundling dependencies into containers or archives reduces the surface area of components that can independently change or disappear; hosting on archival infrastructure (Zenodo, DANDI, Software Heritage) with content-addressed identifiers allows recipients to verify integrity regardless of source; and mirroring across multiple platforms protects against the failure of any single registry.

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 recorded.
  - ☐ 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)**: All referenced modules and components 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

No single tool addresses all the dimensions of scientific rigor described above, though many tools contribute to one or several properties simultaneously. For example, container technologies such as Docker or Singularity serve Portability by providing OS-level isolation from explicit environment specifications, enable Ephemerality by disposing per-execution environments, and facilitate Distributability by producing frozen, shareable artifacts. Similarly, version control systems such as `Git` underpin both Tracking (content-addressed identification and change history) and Self-containment (gathering all components under a single root). In practice, self-containing all data might be undesired or prohibitive due to size or other concerns. Extensions such as `git-annex`, `DataLad`, `DVC`, and `Git LFS` strike the balance between the two properties by tracking

metadata that identifies specific data versions and obtainability information instead, e.g., content checksums and URLs of remote servers that act as a source, and providing actionable interfaces to obtain the data when necessary.

This overlap is not incidental — it reflects the interdependence of the STAMPED properties themselves. Achieving Ephemerality, for instance, inherently exercises Portability and Actionability: a disposable environment must be reconstituted from explicit specifications, encouraging end-to-end automation. Likewise, Distributability builds on Self-containment and Tracking: a research object can only be shared in a consistent, retrievable state if its boundary is well-defined and all components are precisely identified. The tools that serve multiple properties do so precisely because they operationalize these connections.

Table 3 maps some commonly used tools and technologies to each STAMPED property. A comprehensive up-to-date review of existing tools is out of scope for this formalization paper and would quickly become outdated. To fill that niche, we initiated a satellite project with a website to describe tools and examples, and characterize them in terms of STAMPED and FAIR principles [cite https://myyoda.github.io/principles-examples/stamped_principles/].

STAMPED properties are tool-agnostic. For example, any system that moves a property from documented to executable moves the research object further along the actionability spectrum, but certain tool categories recur across multiple properties; a well-chosen tool can address several dimensions at once, and a thoughtfully assembled toolchain can cover all of them. Researchers should select tools based on their domain, infrastructure, and team familiarity, recognizing that the tools listed here are illustrative rather than prescriptive: what matters is that the underlying property is satisfied, regardless of which tool achieves it.

Where multiple tools serve the same STAMPED property, the choice between them often involves trade-offs between flexibility and simplicity. A representative example is the choice between `git-annex` and Git LFS for large-file tracking. Both appear under Self-containment and Tracking in Table 3, but they occupy different points on a complexity-flexibility spectrum. `git-annex` supports many remote types (e.g. other git repositories over SSH or HTTP, or simple cloud data stores such as S3, local drives, offline USB archives), making it well-suited to multi-institutional collaboration, long-term archival where data sovereignty or offline access is required, as well as local content tracking. Git LFS offers a simpler alternative: it is transparent to users, natively supported by GitHub and GitLab, and broadly adopted. But it is centralized (requiring an LFS server), lacks offline capability, and provides less flexible remote configurations. Lightweight STAMPED implementations using Git LFS are entirely feasible for easier onboarding, trading federation flexibility for reduced operational overhead. The same pattern—flexibility versus simplicity—recurs across other tool choices in Table 3: container-based versus package-based environment management, workflow engines versus shell scripts, and federated archives versus centralized platforms.

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)[7] 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**[8]: 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[9], FAIR principles for data sharing[1], Software Carpentry's "Good Enough Practices"[10].

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

- Yarik will jot down their ideas for such a table about AI relation - Other items such as the examples as a living entity can be briefly mentioned but do not need sub sub sections

4 Data Availability

5 Code Availability

References

[1] Mark D. Wilkinson, Michel Dumontier, IJsbrand Jan Aalbersberg, Gabrielle Appleton, Myles Axton, Arie Baak, Niklas Blomberg, Jan-Willem

Boiten, Luiz Bonino Da Silva Santos, Philip E. Bourne, Jildau Bouwman, Anthony J. Brookes, Tim Clark, Mercè Crosas, Ingrid Dillo, Olivier Dumon, Scott Edmunds, Chris T. Evelo, Richard Finkers, Alejandra Gonzalez-Beltran, Alasdair J.G. Gray, Paul Groth, Carole Goble, Jeffrey S. Grethe, Jaap Heringa, Peter A.C 'T Hoen, Rob Hooft, Tobias Kuhn, Ruben Kok, Joost Kok, Scott J. Lusher, Maryann E. Martone, Albert Mons, Abel L. Packer, Bengt Persson, Philippe Rocca-Serra, Marco Roos, Rene Van Schaik, Susanna-Assunta Sansone, Erik Schultes, Thierry Sengstag, Ted Slater, George Strawn, Morris A. Swertz, Mark Thompson, Johan Van Der Lei, Erik Van Mulligen, Jan Velterop, Andra Waagmeester, Peter Wittenburg, Katherine Wolstencroft, Jun Zhao, and Barend Mons. The FAIR Guiding Principles for scientific data management and stewardship. *Scientific Data*, 3(1):160018, March 2016. ISSN 2052-4463. doi: 10.1038/sdata.2016.18. URL https://www.nature.com/articles/sdata201618.

[2] Michelle Barker, Neil P. Chue Hong, Daniel S. Katz, Anna-Lena Lamprecht, Carlos Martinez-Ortiz, Fotis Psomopoulos, Jennifer Harrow, Leyla Jael Castro, Morane Gruenpeter, Paula Andrea Martinez, and Tom Honeyman. Introducing the FAIR Principles for research software. *Scientific Data*, 9 (1):622, October 2022. ISSN 2052-4463. doi: 10.1038/s41597-022-01710-x. URL https://www.nature.com/articles/s41597-022-01710-x.

[3] Sean R. Wilkinson, Meznah Aloqalaa, Khalid Belhajjame, Michael R. Crusoe, Bruno De Paula Kinoshita, Luiz Gadelha, Daniel Garijo, Ove Johan Ragnar Gustafsson, Nick Juty, Sehrish Kanwal, Farah Zaib Khan, Johannes Köster, Karsten Peters-von Gehlen, Line Pouchard, Randy K. Rannow, Stian Soiland-Reyes, Nicola Soranzo, Shoaib Sufi, Ziheng Sun, Baiba Vilne, Merridee A. Wouters, Denis Yuen, and Carole Goble. Applying the FAIR Principles to computational workflows. *Scientific Data*, 12 (1):328, February 2025. ISSN 2052-4463. doi: 10.1038/s41597-025-04451-9. URL https://www.nature.com/articles/s41597-025-04451-9.

[4] Michael Hanke, Matteo Visconti di Oleggio Castello, Kyle Meyer, Benjamin Poldrack, and Yaroslav O. Halchenko. YODA: YODA's organigram on data analysis, 2018. URL https://github.com/myyoda/poster/blob/master/ohbm2018.pdf. Published: Poster presented at the annual meeting of the Organization for Human Brain Mapping, Singapore.

[5] S. Bradner. Key words for use in RFCs to Indicate Requirement Levels. Technical Report RFC2119, RFC Editor, March 1997. URL https://www.rfc-editor.org/info/rfc2119.

[6] Michael Hanke, Franco Pestilli, Adina S. Wagner, Christopher J. Markiewicz, Jean-Baptiste Poline, and Yaroslav O. Halchenko. In defense of decentralized research data management. *Neuroforum*, 0(0): 000010151520200037, January 2021. ISSN 2363-7013, 0947-0875. doi: 10.1515/nf-2020-0037. URL https://www.degruyter.com/document/doi/10.1515/nf-2020-0037/html.

[7] Krzysztof J. Gorgolewski, Tibor Auer, Vince D. Calhoun, R. Cameron Craddock, Samir Das, Eugene P. Duff, Guillaume Flandin, Satrajit S. Ghosh, Tristan Glatard, Yaroslav O. Halchenko, Daniel A. Handwerker, Michael Hanke, David Keator, Xiangrui Li, Zachary Michael, Camille Maumet, B. Nolan Nichols, Thomas E. Nichols, John Pellman, Jean-Baptiste Poline, Ariel Rokem, Gunnar Schaefer, Vanessa Sochat, William Triplett, Jessica A. Turner, Gaël Varoquaux, and Russell A. Poldrack. The brain imaging data structure, a format for organizing and describing outputs of neuroimaging experiments. *Scientific Data*, 3(1):160044, June 2016. ISSN 2052-4463. doi: 10.1038/sdata.2016.44. URL https://www.nature.com/articles/sdata201644.

[8] Chenying Zhao, Dorota Jarecka, Sydney Covitz, Yibei Chen, Simon B. Eickhoff, Damien A. Fair, Alexandre R. Franco, Yaroslav O. Halchenko, Timothy J. Hendrickson, Felix Hoffstaedter, Audrey Houghton, Gregory Kiar, Austin Macdonald, Kahini Mehta, Michael P. Milham, Taylor Salo, Michael Hanke, Satrajit S. Ghosh, Matthew Cieslak, and Theodore D. Satterthwaite. A reproducible and generalizable software workflow for analysis of large-scale neuroimaging data collections using BIDS Apps. *Imaging Neuroscience*, 2:imag–2–00074, January 2024. ISSN 2837-6056. doi: 10.1162/imag_a_00074. URL https://direct.mit.edu/imag/article/doi/10.1162/imag_a_00074/119046/A-reproducible-and-generalizable-software-workflow.

[9] William Stafford Noble. A Quick Guide to Organizing Computational Biology Projects. *PLoS Computational Biology*, 5(7):e1000424, July 2009. ISSN 1553-7358. doi: 10.1371/journal.pcbi.1000424. URL https://dx.plos.org/10.1371/journal.pcbi.1000424.

[10] Greg Wilson, Jennifer Bryan, Karen Cranston, Justin Kitzes, Lex Nederbragt, and Tracy K. Teal. Good enough practices in scientific computing. *PLOS Computational Biology*, 13(6):e1005510, June 2017. ISSN 1553-7358. doi: 10.1371/journal.pcbi.1005510. URL https://dx.plos.org/10.1371/journal.pcbi.1005510.

6 Author Contributions

7 Competing Interests

The authors declare no competing interests.

8 Acknowledgments

9 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.

Table 3: Sample tools to improve scientific practice for each STAMPED property.

Property	Tools / Technologies	Role
S — Self-containment	`Git` submodules, DataLad, `git-annex`, DVC, Git LFS	Gather all components (code, data, environments) under a single root via direct inclusion or explicit references (e.g., subdatasets, registered URLs)
T — Tracking	`Git`, `git-annex`, DataLad, DVC, Docker and other containers, Kedro, Git LFS, OSF.io, Zenodo and other data archives providing DOIs	Version control for code and data; content-addressed identification; provenance recording (e.g., `datalad run` for programmatic provenance capture); persistent identifiers for content (DOIs)
A — Actionability	Make, Snakemake, Nextflow, CWL, `datalad run`, `git annex addcomputed`	Define executable specifications for workflows; provide clear entry points for reproducing results
M — Modularity	`Git` submodules, DataLad subdatasets, Kedro	Organize components into independently versioned, composable modules; provide project templates with logical directory separation
P — Portability	Docker, Singularity, Nix, conda, `pip` (`pyproject.toml`, `requirements.txt`), npm	Explicitly specify and isolate computational environments; container-based (OS-level isolation) or package-based (declarative, reproducible builds)
E — Ephemerality	Docker, Singularity, `docker compose`, Slurm, cloud deployments (e.g., AWS, GCP, Azure, GitHub Actions)	Spawn disposable, temporary environments per execution; validate that specifications are complete by reconstituting from scratch
D — Distributability	Zenodo, PyPI, conda-forge, DockerHub/GHCR, RO-Crate, `npm ci`, `pip freeze`, executable compilers	Archive and persistently host frozen, versioned research objects and their components; enable retrieval by others in the intended state