Data Descriptor Template: Schema, Provenance, and Release Readiness

A FAIR-style exemplar for dataset papers

Daniel Ari Friedman
Active Inference Institute
daniel@activeinference.institute
ORCID: 0000-0001-6232-9096

DOI: 10.5281/zenodo.21298883

2026-07-10

Contents

1 Abstract **2**

2 Introduction **3**
2.1 What this template provides . . . 3
2.2 Scope and honesty . . . 3

3 Data description **4**
3.1 File inventory . . . 4
3.2 Release boundary . . . 4

4 Schema and data dictionary **5**
4.1 Field contract . . . 5
4.2 Schema fingerprint . . . 5
4.3 Constraint validation . . . 6

5 Provenance **7**
5.1 The provenance chain . . . 7
5.2 Why depth matters . . . 7

6 Quality validation **8**
6.1 The validation gate reacts to defects . . . 8
6.2 Descriptor↔file verification . . . 8
6.3 Machine-readable release manifest . . . 8
6.4 Validation evidence . . . 9

7 Usage notes **10**
7.1 Regenerating the artifacts . . . 10
7.2 Forking in a real dataset . . . 10
7.3 Boundaries and claims . . . 10

8 References **11**

1 Abstract

This exemplar demonstrates a **data descriptor** workflow in which the schema, file inventory, provenance chain, license boundary, and validation gate are treated as first-class research artifacts rather than afterthoughts. It ships a small, public, synthetic demonstration dataset (two CSV files under `data/fixtures/`) and a machine-readable descriptor (`data/example_descriptor.json`) that declares each file's media type, sha256 checksum, and row count alongside a six-field data dictionary with typed constraints. A tested validation library (`src/data_descriptor/`) checks the descriptor's shape, safety, and completeness; recomputes each declared checksum and row count against the bytes on disk; and emits a deterministic, metadata-only release manifest suitable for pre-publication review. Every figure and quantitative claim in this manuscript is produced by that library and regenerated on demand, so the prose describes structure and provenance rather than transcribing values that would drift. This is a template with a demonstration dataset: it makes no scientific claim about the data, only about how to describe and release a dataset responsibly.

2 Introduction

A *data descriptor* (or data paper) publishes a dataset as a citable research object. Unlike an analysis paper, its contribution is not a statistical finding but a **contract**: a precise, machine-readable account of what the dataset contains, how it was produced, how it is licensed, and what quality guarantees it carries. The FAIR guiding principles — Findable, Accessible, Interoperable, Reusable [Wilkinson et al., 2016] — frame why this matters: a dataset is only reusable to the extent that its structure and provenance are legible to both humans and machines.

2.1 What this template provides

This exemplar packages the recurring moving parts of a data descriptor so that a fork can start from a working, tested baseline:

- A **schema / data dictionary**: named fields with declared types, nullability, units, and value constraints (patterns, enumerations, numeric bounds).
- A **file inventory**: each shipped file with its media type, a sha256 checksum, and a row count.
- A **provenance chain**: the ordered steps and agents that produced the release.
- A **license boundary**: an explicit reuse license, and the discipline of publishing *metadata and checksums* rather than bundling restricted bytes.
- A **validation gate**: a tested library that rejects malformed, unsafe, or incomplete descriptors and verifies declared checksums against real bytes.

2.2 Scope and honesty

The shipped dataset is **synthetic and deliberately small** — it exists to make the workflow runnable and testable, not to support any empirical claim. Its values are generated deterministically. Accordingly, this manuscript restricts its claims to dataset *structure, provenance, quality, and release readiness*. When a real dataset is forked in, the same validation and figure code applies unchanged; only the descriptor and fixture files are replaced. The sections that follow describe the demonstration dataset (sec. 3), its schema (sec. 4), its provenance (sec. 5), and the quality gate that binds the descriptor to the bytes it names (sec. 6), and close with usage and forking notes (sec. 7).

3 Data description

The demonstration dataset consists of two comma-separated files under `data/fixtures/`, described by the machine-readable descriptor `data/example_descriptor.json`:

- `fixtures/measurements.csv` — one row per synthetic sample, with a bounded measurement, an assignment group, a capture instrument, and a collection date.
- `fixtures/subjects.csv` — one row per synthetic subject, with an enrollment site and date. Each measurement references a subject through the `subject_id` foreign key.

3.1 File inventory

The descriptor declares each file's media type, sha256 checksum, and row count. fig. 1 shows the declared row counts, read directly from the descriptor by `file_inventory_rows()` in `src/data_descriptor/figures.py` and plotted by the thin analysis script `scripts/generate_figures.py`.

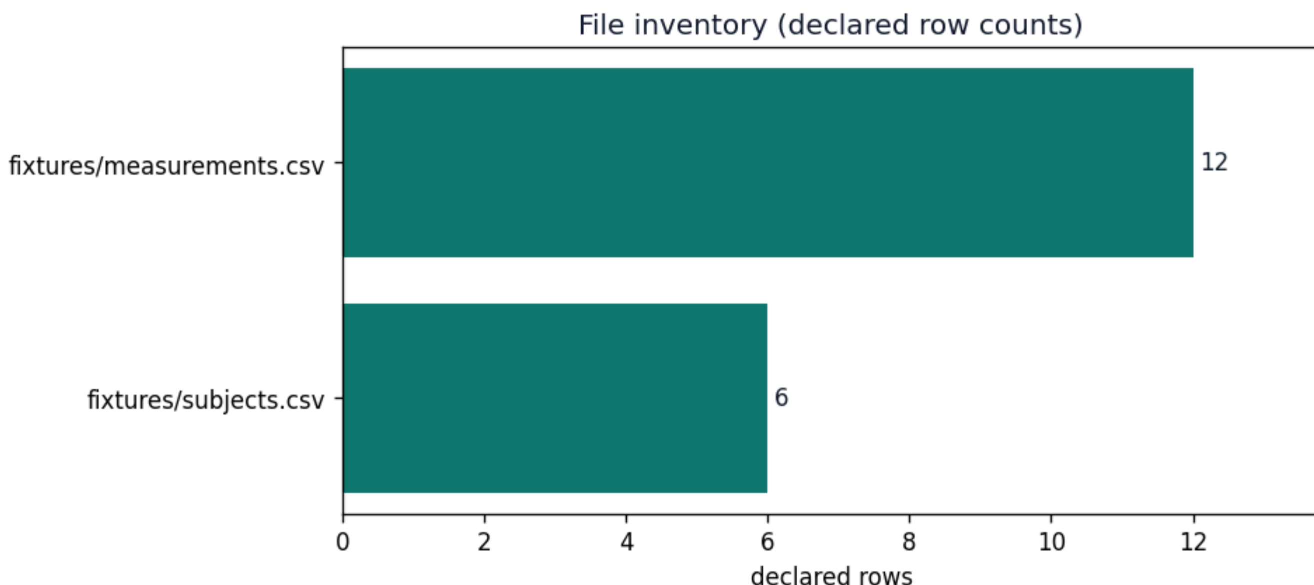

Figure 1: File inventory: declared row counts per file, from `file_inventory_rows()`. The measurement table is the larger of the two files; the subject table is its dimension companion. Both are declared as `text/csv`. The bar values are the row counts recorded in the descriptor, not re-derived here — the quality gate (sec. 6) is where declared and actual counts are reconciled.

Running the script regenerates the figures under `manuscript/figures/`; the prose below and in later sections describes what those artifacts show. Because the figures are produced from the descriptor and the fixture bytes, they cannot drift from the data without the tests and the quality gate noticing.

3.2 Release boundary

The descriptor is deliberately a *metadata* object. For a real dataset, the descriptor and its checksums can be published even when the underlying bytes are access-controlled: the checksums let a downstream user verify integrity once they obtain the files through the appropriate channel. In this template the fixture bytes are public and small, so they are shipped alongside the descriptor and used to demonstrate end-to-end verification.

4 Schema and data dictionary

The descriptor's `fields` list is the dataset's **data dictionary**: it names each column of the primary measurement table and declares its type, nullability, optional unit, and value constraints. fig. 2 renders this dictionary directly from the descriptor via `schema_table_rows()` in `src/data_descriptor/figures.py`.

Field schema / data dictionary

field	type	nullable	unit	constraint
sample_id	string	no	—	pattern ^S[0-9]{3}$
subject_id	string	no	—	pattern ^P[0-9]{3}$
group	category	no	—	{control, treatment}
value	number	no	normalized_score	[0, 1]
collected_on	date	no	—	—
instrument	category	no	—	{sensor_a, sensor_b}

Figure 2: Field schema / data dictionary: one row per declared field, with type, nullability, unit, and a compact constraint label, from `schema_table_rows()`. Identifier fields carry regular-expression patterns; the categorical fields carry closed enumerations; the quantitative field carries a unit and numeric bounds. The table is generated from `data/example_descriptor.json` and cannot silently disagree with it.

4.1 Field contract

The six declared fields exercise the constraint vocabulary the validator understands:

- **`sample_id`** (`string`, not null) — the primary key, constrained to the pattern `^S[0-9]{3}$`.
- **`subject_id`** (`string`, not null) — foreign key into `fixtures/subjects.csv`, constrained to `^P[0-9]{3}$`.
- **`group`** (`category`, not null) — a closed enumeration of `control` and `treatment`.
- **`value`** (`number`, not null) — the measurement, carrying the unit `normalized_score` and numeric bounds `[0, 1]`.
- **`collected_on`** (`date`, not null) — an ISO-8601 collection date.
- **`instrument`** (`category`, not null) — a closed enumeration of `sensor_a` and `sensor_b`.

4.2 Schema fingerprint

`descriptor_fingerprint()` reduces the field list to a stable, order-independent sha256 fingerprint over `(name, type, nullable)` triples. Reordering the fields leaves the fingerprint unchanged, so two descriptors with the same schema hash identically regardless of authoring order — the property is asserted directly in the test suite. The fingerprint is what lets a release manifest reference "the schema" by a single value that changes if and only if the field contract changes.

4.3 Constraint validation

The validator flags quantitative fields that lack a unit, categorical fields that lack allowed values, enumerations that are present but empty, numeric bounds where the minimum exceeds the maximum, and patterns declared on non-text fields. On the shipped descriptor none of these fire — the data dictionary above is complete, so the readiness score is unpenalised (see sec. 6).

5 Provenance

Provenance records *how the release came to be.* The descriptor's `provenance` list is an ordered chain of steps, each naming the agent responsible. fig. 3 renders that chain via `provenance_steps()` in `src/data_descriptor/figures.py`.

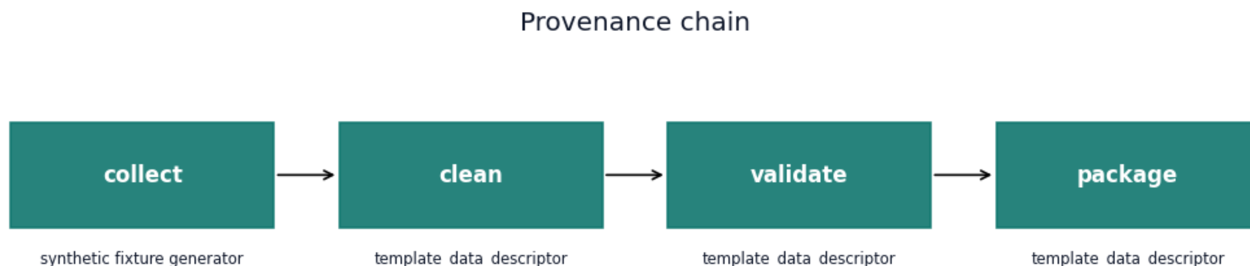

Figure 3: Provenance chain: the ordered steps that produced the release, from `provenance_steps()`. Each box is a declared step; the label beneath is the responsible agent. The chain runs left to right from raw collection to the packaged, metadata-only release manifest.

5.1 The provenance chain

The shipped descriptor declares four steps:

1. **collect** — a synthetic fixture generator emits the deterministic measurement and subject tables.
2. **clean** — identifiers are normalized and the measurement range is bounded.
3. **validate** — the descriptor is checked for schema, constraint, and byte-level agreement.
4. **package** — the metadata-only release manifest is emitted.

5.2 Why depth matters

The validator treats provenance shorter than two steps as a warning — a release that records only "we have the data" but not "how it was produced and checked" is thinner than a reusable descriptor should be. The four-step chain here pairs a collection origin with an explicit validation and packaging trail, which is the minimum a downstream reuser needs to judge whether the dataset was produced the way they require. For a real dataset, these steps would cite concrete tools, versions, and operators rather than the template's synthetic agents.

6 Quality validation

The quality gate is what separates a descriptor that merely *claims* to describe a dataset from one that *provably* does. It has two layers: structural validation of the descriptor, and byte-level reconciliation of the descriptor against the files it names.

6.1 The validation gate reacts to defects

`validate_descriptor()` emits severity-tagged findings for missing required keys, unsafe or duplicate file paths, malformed checksums, unknown media types, duplicate or malformed fields, constraint violations, missing primary keys, and thin provenance. `build_descriptor_report()` folds these into a readiness score, penalising each error and warning.

To show the gate is not vacuous, fig. 4 compares the shipped fixture against a deliberately-perturbed copy produced by `demo_broken_descriptor()` (which strips the license, corrupts a checksum, zeroes a row count, and removes a unit). The clean fixture produces zero findings; the perturbed demo produces several errors and warnings.

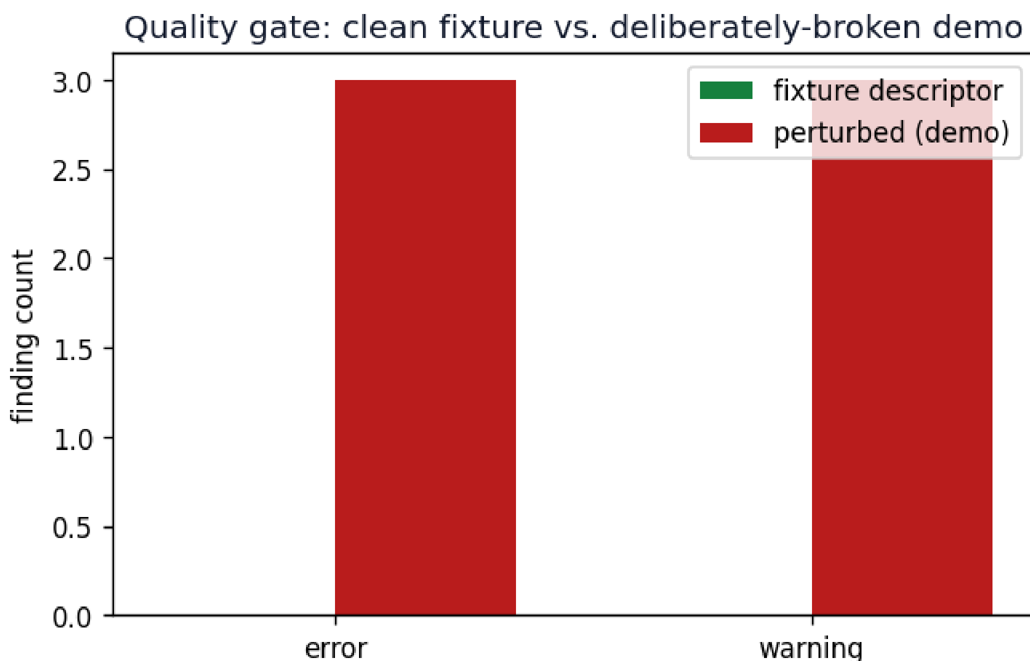

Figure 4: Quality gate: validation findings by severity for the clean fixture descriptor (left, zero findings) versus a deliberately-broken demonstration copy (right). The demonstration perturbation is clearly named in code and never treated as real data — it exists only to prove the gate reacts. Counts come from `severity_counts()`.

6.2 Descriptor↔file verification

A checksum is only meaningful if something checks it. `verify_descriptor_files()` recomputes the sha256 digest and data-row count of every declared file that is present on disk and compares them to the descriptor's declarations, reporting each file as `verified`, `checksum_mismatch`, `row_mismatch`, or `absent`. Files a descriptor references but does not bundle are reported as `absent` rather than as failures — verification is asserted only for bytes that are actually present, which is what makes the metadata-only release boundary honest.

fig. 5 shows this reconciliation for the shipped fixture: both files verify, declared and actual row counts agree, and each recomputed checksum matches its declaration, so the readiness score is unpenalised.

6.3 Machine-readable release manifest

`build_release_bundle_manifest()` packages the above into a deterministic, JSON-ready manifest containing the schema fingerprint, per-file checksums and row counts, field-level unit/bounds/enumeration summaries, provenance

Descriptor↔file verification (readiness 1.0)

file	declared rows	actual rows	checksum	status
xtures/measurements.	12	12	match	verified
fixtures/subjects.csv	6	6	match	verified

Figure 5: Descriptor↔file verification: declared versus recomputed row counts and checksum agreement per file, from `verify_descriptor_files()`. Both shipped fixture files verify against their bytes on disk; the title reports the descriptor's readiness score. If a fixture file were edited without updating the descriptor, the corresponding row would flip to a mismatch status and the test suite would fail.

steps, and the readiness verdict — carrying metadata and checksums, never the dataset bytes. The thin script `scripts/generate_release_artifacts.py` writes this manifest (plus the readiness report and constraint summary) under `output/reports/` for pre-publication review.

6.4 Validation evidence

Every capability above is exercised by the zero-mock `tests/` suite: structural validation and negative controls, order-independent schema fingerprints, byte-level verification against real temporary files, figure-data preparation, and an integration test that runs the figure script end to end and asserts real PNGs are written. Coverage exceeds the 90% project gate with no mocks.

7 Usage notes

7.1 Regenerating the artifacts

From the monorepo root, regenerate the figures and the release-review artifacts:

```
uv run python projects/templates/template_data_descriptor/scripts/generate_figures.py
uv run python projects/templates/template_data_descriptor/scripts/generate_release_artifacts.py
```

The first writes the five figures under `manuscript/figures/`; the second writes the descriptor readiness report, field-constraint summary, and metadata-only release manifest under `output/reports/`. Both are thin orchestrators: all computation lives in the tested `src/data_descriptor/` package.

7.2 Forking in a real dataset

To adapt this template to a real dataset:

1. Replace the fixture files under `data/fixtures/` with your data files (or, for restricted data, leave them out and publish descriptor + checksums only).
2. Edit `data/example_descriptor.json`: update `name`, `license`, the `files` inventory (path, media type, sha256 checksum, row count), the `fields` data dictionary, `primary_key`, and the `provenance` chain.
3. Recompute each file's checksum and row count and record them in the descriptor. For present files, `verify_descriptor_files()` will confirm the descriptor matches the bytes; a mismatch fails the test suite.
4. Extend field constraints (patterns, enumerations, numeric bounds, units) before publishing real data, and keep the provenance chain honest — cite real tools, versions, and operators.

Use `scripts/audit/copy_exemplar.py` from the monorepo to fork the template cleanly, and keep `domain_profile.yaml` and `experiment_plan.yaml` aligned with your dataset.

7.3 Boundaries and claims

This is a template with a synthetic demonstration dataset. Its claims are limited to dataset structure, schema completeness, provenance depth, license state, byte-level integrity, and release readiness — never to any empirical effect in the data. When you fork it, keep that discipline: a data descriptor earns trust by describing a dataset precisely and verifiably, not by over-claiming what the dataset shows.

8 References

This exemplar draws on established data-publishing practice: the FAIR guiding principles for reusable data [Wilkinson et al., 2016], datasheets-style structured dataset documentation [Gebru et al., 2021], the W3C PROV data model for provenance [Moreau and Missier, 2013], and the Frictionless Data Package container format for machine-readable descriptors [Frictionless Data, 2023].

The bibliography lives in `manuscript/references.bib` and is read by Pandoc during PDF render; every `[@key]` citation in the manuscript is resolved against that file. To validate that `references.bib` is syntactically clean and complete:

```
uv run python -m infrastructure.reference.citation.cli validate \
    projects/templates/template_data_descriptor/manuscript/references.bib --strict
```

References

Frictionless Data. Data package: A simple container format for describing a coherent collection of data. Open Knowledge Foundation specification, 2023. URL https://specs.frictionlessdata.io/data-package/.

Timnit Gebru, Jamie Morgenstern, Briana Vecchione, Jennifer Wortman Vaughan, Hanna Wallach, Hal Daumé III, and Kate Crawford. Datasheets for datasets. *Communications of the ACM*, 64(12):86–92, 2021. doi: 10.1145/3458723.

Luc Moreau and Paolo Missier. PROV-DM: The PROV data model. W3C recommendation, World Wide Web Consortium (W3C), 2013. URL https://www.w3.org/TR/prov-dm/.

Mark D. Wilkinson, Michel Dumontier, IJsbrand Jan Aalbersberg, et al. The FAIR guiding principles for scientific data management and stewardship. *Scientific Data*, 3:160018, 2016. doi: 10.1038/sdata.2016.18.