Practical Session: Metadata, Environment, and Validation

Approximate duration: ~40 min; recommended hands-on format: 80 min. The text is intended to be read aloud together with the presentation; interactions and demonstrations are included in the timing.

Key idea

FAIR practice has three levels of control: the schema checks form, package-level scripts check consistency between files, and a person checks scientific sufficiency. No level replaces another. Unknown information should be represented honestly, and the software environment should be preserved as part of the result.

After this lecture, participants will be able to

- fix typical schema errors;
- distinguish a schema-valid record from a scientifically sufficient record;
- explain package-level consistency checks;
- record the software environment from versions through containers;
- apply a publication checklist.

Preparation status

The lecture text and presentation define the learning logic. Before a hands-on session, a programming specialist should complete and test the Colab/Binder notebook, the set of broken records, the package-level validator, and the manifest.json generator. In the 40-minute format, some actions are demonstrated by the lecturer; in the 80-minute format, participants perform them themselves.

1. Two practical tracks: researcher and data steward [0–5 min]

On screen

Two columns: Researcher / Data steward; Colab/Binder below, with no local installation.

This lecture is fundamentally different from the first two. There, we built a conceptual model. Here, we need to see where machine validation helps and where it can create a false sense of security.

The researcher works as the author of the package: opens a record, runs the schema validator, fixes errors, generates derived documents, and records the environment. The data steward receives someone else's package and checks cross-file consistency, scientific gaps, and whether the package is understandable enough for reuse.

These roles are deliberately separated. The author knows the context and may mentally fill in things that are absent from the files. A data steward should not do that. They work with the evidence that will be available to a future user.

A hands-on session needs a ready-to-use environment. A participant opens Colab or Binder and immediately sees schemas, examples, broken records, and scripts. Installing Python and JSON Schema libraries should not be part of this lecture.

In the 40-minute format, the lecturer demonstrates the main exercises in a controlled way and assigns homework. In the 80-minute format, participants actually work in the two roles and conduct peer review.

Audience interaction
At the start, ask participants to choose a role: researcher or data steward. In a large group, split the audience roughly in half. This makes it easier to discuss the same files from two perspectives.

2. Six rules most often violated in metadata [5–10 min]

On screen
Six numbered cards: extra field; value + unit; enum/format; unknown; relative paths; licence vs access.

Rule one: do not add arbitrary fields outside the schema when the schema is closed. To a person, an “extra field” may seem harmless. A machine pipeline does not know its semantics and may reject the record.

Rule two: keep value and unit separate. A value such as “300 K” in a single string is less useful than a numeric value 300 and a separate unit. Ideally, the unit should have a stable identifier, but even an explicit structured unit is already better for machine processing.

Rule three: controlled values must be written exactly as defined. `nvt` and `NVT` may be different values; a date must match the schema format; `Other` without a detail field defeats the purpose of a controlled vocabulary.

Rule four: do not replace unknown with 0 or false. Zero means a known numerical zero; false means a known Boolean value. If a fact is unknown, use the allowed mechanism for unknown or absence, or record the gap explicitly.

Rule five: paths inside the package should be relative. An absolute path such as `/home/user/...` is not portable and may expose unnecessary system information. Rule six: licence and access status are different. A licence defines rights for reuse; access status tells users whether they can obtain the content now.

Transition to the next block
We will now see how these rules appear in a concrete broken record.

3. Exercise 1: a “broken” dft.json [10–17 min]

On screen
JSON fragment on the left; five errors and expected validator messages on the right.

The broken record should contain different kinds of errors: a field outside the schema, a numerical value represented as a string with a unit, an incorrect enum case, an absolute path, and `Other` without the required detail. If all errors are of one type, participants learn to guess a pattern rather than read the validator.

The working sequence is: run validation; read the error path; open the relevant schema or guideline; fix the value; run validation again. The goal is not to memorise JSON Schema syntax. The goal is to develop the habit that a validator message is navigation to a rule.

After PASS, ask explicitly: “Does this mean the calculation is reproducible?” No. The validator checked only the structural contract: fields, types, enums, required elements, and patterns. It does not know whether the content is scientifically meaningful.

This is an important pedagogical point. A green PASS creates strong psychological confidence. We should therefore immediately show a package that is schema-valid but still has a missing software version, an ambiguous final model, or incomplete provenance.

In the 80-minute format, the researcher group fixes the record independently. In parallel, the data-steward group receives a valid-but-incomplete package and prepares a review.

Audience interaction
In the 40-minute format, fix two errors live and leave another 2–3 as a mini-quiz. In the 80-minute format, allow 7–10 minutes of independent work.

4. Schema-valid ≠ scientifically sufficient [17–22 min]

On screen
Three columns: VALID JSON / BUT... / DATA STEWARD QUESTION; at the bottom: VALID ≠ SUFFICIENT.

Imagine a record that passes the schema. `software` is populated, but the version is missing. `model_ref` contains `best_model`, but there is no exact checkpoint or checksum. The split is described as 80/10/10, but exact membership cannot be reproduced. `knownLimitations` is empty.

A data steward should not automatically respond by asking for “twenty more fields”. Instead, they formulate competency questions: Which exact software version? Which exact model artefact was used? Can the split be regenerated? Is the applicability domain known? Does provenance really confirm the model transfer?

This lets us distinguish a schema gap from a package-completeness gap. A schema gap means the profile has no way to represent an important fact. A package gap means the mechanism exists, but the author has not provided the evidence.

This distinction is critical for profile evolution. If every incomplete submission leads to a new field, the schema quickly becomes unwieldy. First, we need to show that the representation mechanism is genuinely missing rather than merely the value.

Scientific sufficiency also depends on intended reuse. Less metadata may be enough for quick discovery; far more is needed for an exact rerun. A profile may therefore have Required, Conditional, Recommended, and competency-question layers.

Transition to the next block
The next level can be automated much more strongly: consistency between files and records.

5. The second level: package consistency [22–27 min]

On screen
Six checks: workflow path, provenance entity, manifest entry, SHA-256, final model, authors/licence.

A package-level validator works with the package as a whole. Every local path referenced by the workflow should exist. Every provenance Entity that represents a file should resolve to a manifest entry. Every checksum should match the actual SHA-256. There should be no dangling IDs.

Identity checks are particularly valuable. If `mlip.json` declares model-017 as the final model while `md.json` points to model-018, that is a FAIL. Both JSON files can still be schema-valid while the package is internally contradictory.

Another class of checks concerns derived documents. MODEL_CARD should match the current MLIP and dataset metadata. CITATION.cff authors should be consistent with creators. Licence and root-level access information should not contradict one another.

Lifecycle is another class. If the package is marked complete, `pending` artefacts need an explanation or should be absent. If a manifest is generated and a file is later modified, the hash check should fail.

For a training validator, stable Rule IDs such as MAN-003 or PROV-006 are useful, with a report format such as PASS/FAIL + rule + path + message. This simplifies debugging and lets the guideline point to a specific rule.

The first working script does not need to be complex. Four to six checks are enough to demonstrate the difference between schema validation and package consistency.

Transition to the next block
We will now show a derived artefact where the single-source-of-truth principle is especially clear: MODEL_CARD.

6. MODEL_CARD: structured metadata as the source of truth [27–30 min]

On screen
mlip.json + dataset.json → make_model_card.py → MODEL_CARD.md.

MODEL_CARD is intended for people: intended use, training data, architecture, hyperparameters, metrics, checkpoint selection, validity domain, limitations, and licence. But if the card is edited manually in parallel with JSON, the two representations will diverge.

The rule is: structured metadata are the source of truth; MODEL_CARD is a generated human-readable representation. If the card is wrong, edit the source JSON and regenerate.

In a notebook, this can be demonstrated in a minute: change `knownLimitations` or the final model identifier in mlip.json, run `make_model_card.py`, and show the updated text.

A useful curation mechanism is a section for “what is missing”. If the generator detects empty recommended fields, the card can expose the gaps explicitly. This helps the researcher improve the metadata before publication.

The same idea generalises: a human-readable README or report can be derived from machine-readable records, provided the source and generation rule are defined.

Transition to the next block
When metadata and derived documents are complete, the package can be fixed technically: the manifest is generated last.

7. The manifest is generated last [30–33 min]

On screen
metadata → MODEL_CARD → README/docs → manifest; live hash mismatch.

The manifest records physical inventory and integrity. For each file it stores the relative path, size, checksum, and classification or access fields. SHA-256 makes accidental modification observable.

The lifecycle rule is simple: generate the manifest last. If the README, JSON, or model card is changed afterwards, the stored hash no longer matches the actual content. Therefore: change package → regenerate manifest.

This is one of the strongest live demonstrations. First, the package validator reports PASS. Change one character in the README. Run the check again → hash mismatch. Regenerate the manifest → PASS. Participants see the integrity concept directly.

In production, the generator should exclude the manifest itself automatically or use an agreed handling strategy so that a recursive hash problem does not arise. In the teaching notebook, this should be implemented transparently.

After the manifest is generated, the package has a version or release state. If a new editing cycle begins, that is a new package state rather than the same frozen object.

Transition to the next block
File integrity does not guarantee that the software stack can be reconstructed. The environment is therefore a separate research artefact.

8. The software environment as part of the result [33–36 min]

On screen
Three levels: versions → environment.yml/requirements → container + recipe.

A computational result depends on the environment. A NequIP checkpoint may fail to load because of a PyTorch/CUDA mismatch. A Quantum ESPRESSO calculation may differ because of version or build, libraries, MPI, or pseudopotentials.

The minimum level is names and versions. Better is a dependency specification such as conda environment.yml, requirements.txt, or a module list. Stronger still is a container image together with its recipe. But the container also needs to be versioned and identified.

For HPC, Apptainer is often practical; for local or cloud execution, Docker is common. A profile should not depend on one technology. It should require enough evidence to reconstruct the environment.

Workflow tools are also part of the environment. If execution uses AiiDA, Snakemake, cwltool, or Parsl, the runner version can affect execution semantics and provenance capture.

Code should be versioned in a repository. For citation, use CITATION.cff or CodeMeta; for preservation, use a release archive, Zenodo, or Software Heritage depending on the use case. The package should point to an exact version or commit, not merely to a GitHub homepage.

Transition to the next block
We can now formulate the complete quality-control architecture in three levels.

9. Three levels of validation: form → consistency → sufficiency [36–38 min]

On screen
Three columns: SCHEMA / CONSISTENCY / SUFFICIENCY.

Level 1 - Schema: types, required fields, enums, and patterns. This is the syntax and structure contract and is best automated with a generic JSON Schema validator.

Level 2 - Consistency: cross-file references, hashes, identities, lifecycle, and agreement among workflow, provenance, and manifest. This is automated with a profile-specific package validator.

Level 3 - Scientific sufficiency: whether enough evidence exists for the claimed reuse or reproduction; whether a selection criterion is justified; whether the validity domain is adequate; whether convergence information is sufficient. This requires a domain expert or data steward.

No level replaces another. Human review without schema validation misses formal errors. Schema validation without a package validator misses cross-file contradictions. Two automated levels without domain review can create false confidence.

FAIR curation is therefore not an alternative to automation. It is an architecture in which automation checks what has been formalised, while a person remains responsible for meaning.

Transition to the next block
We will finish with a publication checklist and the requirements for a complete practical session.

10. Publication gate and homework [38–40 min]

On screen

Seven-point checklist + homework.

Before publication, check that: JSON records validate against pinned schemas; referenced files exist; there are no dangling provenance IDs; the manifest is complete and SHA-256 checks pass; the final model, split, and environment are unambiguous; README, licence, and access information are consistent; scientific sufficiency has been reviewed by a person.

In the 40-minute format, homework is to describe one stage and one transition in your own workflow, including inputs and outputs, environment, a human decision, and one competency question. In the 80-minute format, add workflow and provenance records and conduct peer review.

The practical session does not end with “perfect JSON”. It ends with an understanding of the boundary of automation: machines check contracts and consistency; scientific sufficiency remains a shared responsibility of the researcher and the data steward.

In Lecture 4, we will move one level higher: even an internally consistent package can be poorly interoperable if its terms and units do not have shared machine-recognisable semantics.

Audience interaction

Ask the researcher role to name one thing the validator finds well; ask the data-steward role to name one thing that cannot be resolved without scientific judgement.

Required before a hands-on session

☐ Colab/Binder notebook tested with a clean account
☐ 5–6 broken records + 1 valid-but-incomplete package
☐ package validator: files + manifest + hash + cross-reference
☐ manifest.json generator
☐ make_model_card.py integrated into the notebook
☐ offline backup: demo recording + PDF/printouts

Sources and materials

1. JSON Schema Draft 2020-12. https://json-schema.org/draft/2020-12

2. MATSCI-NASU draft schemas/examples/guidelines and make_model_card.py.

3. MATSCI-NASU Consistency Rules v1.0 draft.

4. MATSCI-NASU DFT–MLIP–MD Reproducibility Profile v1.0 draft.

5. Software Heritage. https://www.softwareheritage.org/

6. CodeMeta. https://codemeta.github.io/