CASSION · COURSE

Reproducible Analysis Workflows

Make the analysis rerunnable by the person who replaces you — version control, pinned environments, parameterised reports and a project layout that survives a handover.

Instructor	Pierre Robentz Cassion
Level	Intermediate
Learner hours	18 h
Taught in	Python · R
Updated	July 29, 2026
Sectors	Public Health · Nutrition · Protection · Education · WASH
Standards and methodologies	OECD DAC evaluation criteria · Core Humanitarian Standard (CHS) · UNICEF indicator definitions

Read and run the course online at data-analysis.cassion.dev/courses/reproducible-workflows

What you will be able to do

- Lay out a project so raw data is read-only, derived data is disposable and code is the only thing edited
- Use git on analysis work, and keep beneficiary data out of a repository permanently
- Pin an environment so a colleague's laptop produces your numbers
- Find and remove the things that make a rerun differ — the clock, the seed, the locale, the file order
- Produce twelve district reports from one parameterised template
- Write a handover a successor can act on, and a check that fails loudly when the upstream export breaks

Prerequisites

- Data Cleaning and Validation

Contents

About this course

An analysis that only you can rerun is a draft. The test is not whether it works — it is whether it still works in eleven months, on someone else's laptop, after you have left, when the source file has changed and nobody remembers which sheet the numbers came from.

This course is unusual in having a worked example that is not a dataset. **The platform you are reading is the case study**, and every claim in the last lesson can be checked against this repository.

Three of those claims shape the whole course.

Twenty dataset files, seventeen generator scripts, and regenerating all of them changes nothing. `pnpm datasets:generate` rewrites every CSV from a seeded script, and `git status` afterwards reports zero modified files. That is what "reproducible" means operationally: rerunning is a *verification* step, not just a production step.

Twenty test files and 4,212 assertions, run before every build. They check things a schema cannot — that a declared row count matches the file on disk, that a French lesson references the French figure, that every published course ships both PDF editions. **The rule they encode: any field naming a shipped file needs a test beside it**, because a path in frontmatter is a string and nothing in a build can tell whether it points at anything.

A schema that refuses non-synthetic data. `dataQuality.synthetic` must be `true` or the build fails, which makes "no record traces to a real person" a property of the repository rather than a promise in a note — the same move as generating the data instead of collecting it.

Both Python and R throughout, with `uv` and `renv` as the environment tools and Quarto for parameterised reports. You need *Data Cleaning and Validation* first: this course is about making a pipeline rerunnable, and the pipeline it assumes you have is the one that course builds.

Part I

The project on disk

CHAPTER 1

Three kinds of file, and only one of them is editable

Lesson 1 of 8 · 135 min

Raw data is read-only, derived data is disposable, and code is the only thing anyone edits. Getting that boundary wrong is how an analysis becomes irreproducible in a way nobody notices for eleven months.

1.1 The layout

```
project/
  data/
    raw/            never edited, never written to, ideally chmod -w
    reference/      lookup tables, thresholds, admin boundary codes
  R/  or  src/      the only files anyone edits
  outputs/
    derived/        cleaned data, disposable
    figures/        charts and the CSV of values behind each
    reports/        rendered documents
  tests/
  README.md
  environment lock  uv.lock or renv.lock
```

Three kinds of file and the distinction is not tidiness.

Raw data is read-only. The export from CommCare, the CSV from DHIS2, the spreadsheet the ministry sent. Once it lands in `data/raw/` nothing writes to it and nobody opens it in Excel to fix a typo — a corrected raw file is a file whose provenance you have destroyed.

Derived data is disposable. Anything in `outputs/` can be deleted at any moment and reproduced by running the code. If deleting `outputs/` loses something, that something was not derived and belongs somewhere else.

Code is the only thing edited. Every correction, every recode, every exclusion is a line of code, which means it is visible, reviewable and rerunnable.

1.2 The test that decides whether you have it right

```
rm -rf outputs/ && make all      # or: python run.py, Rscript run.R
git status --short               # should be empty
```

Delete every output, rerun, and the repository should look untouched. That single command is the whole of reproducibility as an operational property, and it is uncomfortable the first time.

Three ways it usually fails, and each names something real:

A file in `outputs/` that nothing produces. Someone made it by hand, once, and every run since has depended on it. It is raw data wearing the wrong name.

A step that only runs interactively. A cell in a notebook, a line pasted into a console, a manual download. If it is not in a script, it did not happen.

Something that must be run in a particular order and does not say so. The pipeline works on your machine because you know the order.

1.3 Make the raw data unwriteable, literally

```
chmod -R a-w data/raw/
```

```
# In R, the equivalent discipline is a function that only ever reads:
read_raw <- function(name) readr::read_csv(file.path("data/raw", name))
```

It costs one command and prevents the failure it names. An accidental `to_csv("data/raw/survey.csv")` in a notebook is the most destructive thing an analyst does, it produces no error, and nothing downstream can detect it afterwards.

Where the raw file is large or restricted, commit its checksum instead.

```
import hashlib, pathlib

def fingerprint(path: pathlib.Path) -> str:
    return hashlib.sha256(path.read_bytes()).hexdigest()[:16]

print(fingerprint(pathlib.Path("data/raw/survey-2025.csv")))
```

```
tools::md5sum("data/raw/survey-2025.csv")
```

A checksum in the repository turns "is this the same export?" into a question with an answer. Ninety per cent of "the numbers changed and I do not know why" is a source file that changed and nobody noticed.

1.4 Naming, which is provenance

Name	What it tells you
`survey.csv`	Nothing
`final_survey_v2_FINAL.xlsx`	That there are several and this one won an argument
`household-survey-2025.v1.csv`	What, when, and which version

Version by filename, never by overwriting. This platform's datasets do exactly that: a correction ships as `.v2.csv` and never replaces `.v1.csv`, because a notebook pinned to v1 has to keep reproducing.

That is why the dataset files can be cached immutably, and why the course PDFs — which *are* regenerated in place — deliberately cannot be.

1.5 What goes in the repository

In	Out
Code, every script	Raw data with personal information
The environment lock file	Anything above about 50 MB
Small reference tables	Credentials, tokens, connection strings
Generated outputs the build cannot rebuild	`.Rhistory`, `__pycache__`, `.DS_Store`
A README	Outputs that regenerate in seconds

The fourth row is a judgement rather than a rule. This platform commits its generated PDFs, decks and figures — because the deploy runs on Cloudflare with no TeX and no Python, so a contributor without the toolchain can still ship a content change. **Commit a generated artefact when rebuilding it is not available to everyone who needs it**, and not otherwise.

1.6 The README that is actually read

Four sections, and it goes stale less often than a long one.

```
# Nutrition surveillance analysis, Artibonite

## What this produces
A quarterly GAM estimate by commune, and the figures in the cluster report.

## Run it
    uv sync
    uv run python run.py

## Where the data comes from
data/raw/muac-screening-*.csv -- monthly export from the screening database,
downloaded by the M&E officer on the 5th. Checksums in data/raw/CHECKSUMS.

## What you need to know
Commune names arrive spelled four ways; normalisation is in src/clean.py and
must run before anything else.
```

The last section is the one that saves a successor a week. It is the knowledge that lives in your head, and the handover lesson is about getting the rest of it out.

1.7 Report it whole

```
Analysis reproducibility

  Raw exports are read-only under data/raw/ with SHA-256 checksums committed.
  All cleaning, exclusion and recoding is in version-controlled code; no raw
  file has been edited.

  Every output in outputs/ is reproducible by `uv run python run.py` from a
  clean checkout. Deleting outputs/ and rerunning leaves the repository
  unchanged.

  Dataset files are versioned by filename. The v1 file referenced by the
  March report has not been modified; the correction ships as v2.
```

The second paragraph is the claim to make and the one to test before making. It is checkable in one command, and an analysis that cannot pass it should say so rather than claim otherwise.

1.8 What comes next

Version control is what makes "the only thing edited is code" into a history you can read. The next lesson is about using git on analysis work — and about the one thing that must never enter a repository holding programme data, because git is designed to never forget.

CHAPTER 2

Git does not forget, which is the point and the danger

Lesson 2 of 8 · 135 min

A repository remembers every version of every file it has ever held, which is exactly what you want for code and exactly what you cannot allow for a beneficiary list. Deleting the file in a later commit does not remove it.

2.1 The commit that cannot be taken back

```
git add data/raw/beneficiaries.csv
git commit -m "add survey data"
# ... realised three days later ...
git rm data/raw/beneficiaries.csv
git commit -m "remove data"
```

The file is still in the repository. It is in the history, it is in every clone, it is on the remote, and it is in the local copy of everyone who pulled. The second commit removed it from the *current* state and from nothing else.

On a repository holding protection or health data, that is a disclosure. Not a mistake to fix later — a disclosure, with the same obligations as any other.

Which is why the rule is preventive rather than corrective. You cannot un-commit personal data; you can only stop it entering.

2.2 The `.gitignore` you write before the first commit

```
# Raw data. Nothing under here is ever committed.
data/raw/*
!data/raw/.gitkeep
!data/raw/CHECKSUMS

# Derived data, regenerated by the pipeline
outputs/

# Credentials, in every form they arrive in
.env
*.pem
*_secret*
config.local.*

# Editor and language noise
.Rhistory
.RData
__pycache__/
*.pyc
.DS_Store
.ipynb_checkpoints/
```

Write it first, commit it first. A `.gitignore` added after the first commit is a `.gitignore` that arrived too late for the thing it was meant to catch.

The negations matter. `data/raw/*` excludes everything, then `!data/raw/CHECKSUMS` lets back exactly the file that proves which export was used — provenance without content.

2.3 What "personal data" means here, precisely

It is broader than a name column, and this sector's datasets make the point.

Looks anonymous	Is not, because
A case ID	It joins to a case management system that has the name
GPS coordinates of a household	It is an address
Date of birth plus commune plus sex	Three fields identify most people in a small commune
A photograph of a form	Every field on it
A free-text "notes" column	It contains whatever the caseworker typed

The protection course established the arithmetic: a four-way disaggregation of an anonymous dataset produced nineteen cells of one, and a cell of one is an identification.

So the test is not "does it have a name column". It is whether any combination of what the file holds could pick one person out — and in a small commune it usually can.

2.4 The check that runs before the commit does

```
# .git/hooks/pre-commit -- or the pre-commit framework, or a CI step
if git diff --cached --name-only | grep -qE '^data/raw/'; then
  echo "Refusing to commit anything under data/raw/" >&2
  exit 1
fi
```

```
# The same check as an R function, run by whatever CI you have.
```

A hook that refuses is worth more than a policy that reminds. The failure mode is someone in a hurry at 6pm, and a reminder does not survive that.

Add a scan for the shapes secrets take, because a token pasted into a script is the other common case. `gitleaks` and `detect-secrets` both do this in a CI step.

2.5 What to do if it has already happened

The order matters and the first step is not technical.

One: report it. Your organisation has a data-incident procedure. This is one, and whether the repository was public decides how urgent rather than whether.

Two: rotate anything that was a credential. A committed token is compromised permanently; deleting it changes nothing.

Three: rewrite the history, with `git filter-repo` or BFG, and force-push. Every existing clone must be deleted and re-cloned — a clone nobody re-cloned still has the file.

Four: assume it was fetched. On a public repository, assume the file was retrieved and mirrored, and treat the disclosure as complete.

```
git filter-repo --path data/raw/beneficiaries.csv --invert-paths
```

Step three is the only one that looks like a fix and it is the least important.

2.6 Commits on analysis work

The habits differ from software work in two ways worth naming.

Commit the code and the output together when the output is committed. A figure and the script that made it belong in one commit, so a reviewer can see whether the chart matches the change.

Write the why in the message. `fix commune names` is what the diff already shows. "Normalise four spellings of Nord-Ouest; ungrouped, the worst district splits into four fragments and none of them looks alarming" is the thing a successor needs and cannot recover.

```
git commit -m "Group four spellings of Nord-Ouest before computing coverage

Ungrouped, the worst district splits into fragments of 727, 33, 22 and 20
households and none of them looks alarming. The 20-household fragment shows
0% open defecation, which a district table would report as a solved problem."
```

Branch per analysis question, not per day. A branch that answers "does coverage differ by district" can be reviewed and merged; a branch called `work` cannot.

2.7 What a repository of programme analysis should contain

```
project/
  .gitignore          committed first
  README.md           what it produces, how to run it, what to know
  uv.lock             the environment, pinned
  src/                every transformation
  tests/              the checks that fail loudly
  data/
    raw/              .gitkeep and CHECKSUMS only
    reference/        committed: thresholds, admin codes, lookup tables
  outputs/            ignored
```

Reference tables are committed and raw data is not, and the line between them is whether the file describes people. WHO growth standards, IPC thresholds and admin boundary codes are reference; a household list is not.

2.8 Report it whole

```
Data handling in this analysis

  No raw data is committed. data/raw/ is excluded by .gitignore, with only a
  CHECKSUMS file tracked so the exports used can be identified.

  A pre-commit hook refuses any staged file under data/raw/ and a CI step
  scans for credential patterns.

  Reference tables (WHO growth standards, IPC thresholds, admin codes) are
  committed; they describe no individual.

  The repository is private and access is limited to the M&E team. Raw
  exports are held in [the organisation's storage], not here.
```

The last line is the one that gets forgotten and it is the one an audit asks for. Saying where the data *is* matters as much as saying where it is not.

2.9 What comes next

Code in version control still does not reproduce a number if the environment differs. The next lesson pins it, and finds the four things that make a rerun differ when nothing in the repository changed at all.

Part II

The same answer twice

CHAPTER 3

It works on my machine, and that is the bug report

Lesson 3 of 8 · 135 min

The same script, the same data and a different pandas version produce different numbers. A lock file is what turns "it works on my machine" from a defence into a testable claim, and it costs one command to make.

3.1 What a colleague's laptop does differently

You send a script and a CSV. They run it. The number differs.

What differs	What it changes
Package version	A default argument, a rounding rule, a sort order
Language version	Dictionary ordering, integer division, string handling
Operating system	Line endings, file ordering, path separators, locale
Locale	`1,234` parsed as 1234 or as 1.234
What is already installed	A library your script imports and never declares

None of those is in your repository, which is why the code being identical is not enough. **A lock file is the missing half of the analysis.**

3.2 Python: `uv`

```
uv init                     # creates pyproject.toml
uv add pandas matplotlib    # records the dependency and resolves it
uv run python run.py        # runs inside the pinned environment
```

Two files, and both are committed. `pyproject.toml` says what you asked for — `pandas>=2.0` — and `uv.lock` says exactly what you got, down to the hash of every package in the tree.

A colleague runs `uv sync` and has your environment, on their operating system, without you knowing what they had installed before.

Pin the language version too.

```
# pyproject.toml
requires-python = ">=3.12,<3.13"
```

A version range, not a single version. Pinning to `3.12.4` exactly means a colleague on `3.12.7` cannot run it at all, which is a worse failure than the one you were preventing.

3.3 R: `renv`

```
renv::init()                # snapshots the project library
renv::snapshot()            # after adding a package
renv::restore()             # on the colleague's machine
```

`renv.lock` is the equivalent artefact and is committed. It records every package, its version, and the repository it came from, and `renv::restore()` rebuilds it.

Record the R version, which `renv` does automatically, and check it in the script where a difference would matter.

```r
if (getRversion() < "4.3.0") stop("This analysis requires R >= 4.3.0")
```

```python
# The Python equivalent, at the top of the entry point.
import sys
assert sys.version_info >= (3, 12), "This analysis requires Python 3.12+"
```

3.4 Why a lock file and not `requirements.txt`

```
pandas>=2.0
matplotlib
```

That file says almost nothing. It resolves to different versions on different days and does not mention the fifty packages those two pull in. A build in March and a build in July from the same `requirements.txt` are different environments.

	Declares	**Reproduces**
`requirements.txt`	Intent	No
`requirements.txt` with `==` pins	One layer	Partly — transitive dependencies float
`uv.lock` / `renv.lock`	The whole tree	**Yes**

Commit both the intent and the lock. The intent file is what a human edits; the lock is what a machine reproduces from, and neither replaces the other.

3.5 What this platform pins, and why one pin is unusual

```
{
  "packageManager": "pnpm@11.9.0",
  "engines": { "node": ">=22" },
  "devDependencies": { "typescript": "^6.0.3" }
}
```

The package manager itself is pinned, because a different pnpm resolves the lockfile differently and that is the layer below the layer most projects pin.

TypeScript is held at 6.x deliberately. TypeScript 7 — the native compiler — does not yet expose the programmatic API that `astro check` depends on, so `pnpm typecheck` fails outright on 7. **The pin has a reason, the reason is written down, and it names the condition for removing it.**

That is the shape a pin should have. A version constraint with no comment is a constraint nobody will dare remove and nobody can justify keeping.

```
# pandas is pinned below 3.0 because the copy-on-write default changes the
# behaviour of the recode in src/clean.py:88. Revisit when that is rewritten.
pandas = ">=2.1,<3.0"
```

3.6 Containers, and when they are worth it

A container pins the operating system as well, which is the one layer a lock file cannot reach.

Worth it when the analysis has non-Python or non-R dependencies — GDAL, a TeX distribution, a database client — or when it has to run unattended on a server, or when the finding will be re-examined years later.

Not worth it when a lock file already reproduces the result and the audience is two colleagues with laptops. A container adds a build step, a registry and a skill that a small M&E team may not have, and the cost is real.

```
FROM python:3.12-slim
COPY pyproject.toml uv.lock ./
RUN pip install uv && uv sync --frozen
COPY . .
CMD ["uv", "run", "python", "run.py"]
```

Start with the lock file. Reach for a container when you can name the dependency it is pinning that the lock file cannot.

3.7 The test that proves it

Reproduce your own result on a machine that has never seen the project. A fresh clone in a temporary directory is most of the way there.

```
git clone <repo> /tmp/check && cd /tmp/check
uv sync
uv run python run.py
diff -r outputs/ ~/project/outputs/
```

```
# renv::restore() then source("run.R"), and compare.
```

If you cannot do that, "it is reproducible" is untested. A colleague's laptop is better and a CI runner is better still, because it is a clean machine every time and it runs whether or not anyone remembers to.

3.8 Report it whole

```
Computational environment

  Python 3.12, dependencies pinned in uv.lock (committed). Reproduce with:

      uv sync && uv run python run.py

  pandas is held below 3.0 because the copy-on-write default changes the
  recode in src/clean.py; this is revisited when that function is rewritten.

  The pipeline runs on every push in CI on a clean machine, so the claim
  that a fresh checkout reproduces these outputs is tested rather than
  asserted.

  Analysis run on 2026-03-14 with commit a3f9c21.
```

The last line is the one that makes the rest usable a year later. A result without the commit that produced it can be approximately reproduced; with it, exactly.

3.9 What comes next

A pinned environment still produces a different answer on a second run if the code reads the clock, draws a random number, or trusts the order files come back in. The next lesson finds all four.

CHAPTER 4

Four things that change the answer when nothing changed

Lesson 4 of 8 · 135 min

The clock, the seed, the file order and the locale. Each produces a different result from identical code and identical data, each is invisible in a diff, and this platform hit three of the four in production.

4.1 One: the clock

```
from datetime import date
report_date = date.today()          # different every day, by design
```

Anything that reads the current time makes every run differ. A generated-on footer, a filename with today's date, a filter for "the last 30 days", a random seed derived from the time.

The fix is to take the date from the data or from a parameter.

```
import pandas as pd

survey = pd.read_csv("data/raw/household-survey-2025.v1.csv")
as_of = pd.to_datetime(survey["interview_date"]).max()   # from the data
```

```
as_of <- max(as.Date(survey$interview_date))
```

This platform got this wrong and the failure is instructive. The course handouts and lesson decks originally took their generation date from the newest `updated` field across the entire content library. Deterministic — and still wrong, because publishing one new course restamped every existing handout and rewrote all ninety-six deck binaries.

The date is now scoped to the artefact. A handout's date comes from its own course and lessons, a deck's from its own lesson, a report's from its own project. **A deterministic wrong answer is still a wrong answer**, and "the diff is enormous but correct" is how nobody reviews it.

4.2 Two: the seed

```
import numpy as np

rng = np.random.default_rng(20260729)      # stated, committed, reproducible
sample = rng.choice(households, size=200, replace=False)
```

```
set.seed(20260729)
sample(households, 200)
```

Anything random needs a seed and the seed belongs in the code. Bootstrap intervals, random sampling for verification, simulation, train/test splits, jittered scatter points.

Set it once, at the top, visibly. A seed buried three functions deep is a seed someone will move.

And say so in the output. The regression course's bootstrap interval is reported as "400 resamples, seed 20260729" for exactly this reason — an interval nobody can reproduce is not an interval.

Every dataset on this platform is generated by a seeded script, which is what makes `pnpm datasets:generate` a verification step: it rewrites all twenty CSVs and `git status` reports nothing changed.

4.3 Three: the order files come back in

```
import glob
for path in glob.glob("data/raw/*.csv"):     # order is filesystem-dependent
    ...
```

`glob` and `os.listdir` return files in an order that differs between operating systems and sometimes between runs. If anything downstream depends on order — a concatenation, a first-wins deduplication, a row index — the result differs.

```
for path in sorted(glob.glob("data/raw/*.csv")):
    ...
```

```
for (path in sort(list.files("data/raw", full.names = TRUE))) { }
```

Sort it. Always. It costs six characters.

The same applies to dictionary and group order. `groupby` in pandas sorts by default and `dplyr::group_by` does not; a chart whose bar order came from an unsorted group is a chart whose order can change without the data changing.

4.4 Four: the locale

```
float("1,234")           # ValueError, or 1.234, depending on where you are
```

The decimal separator, the thousands separator, the date format and the sort order of accented characters are all locale-dependent, and this platform's audience works across both conventions by definition.

```
survey = pd.read_csv(path, decimal=".", thousands=None)     # stated, not inferred
dates = pd.to_datetime(survey["date"], format="%Y-%m-%d")   # explicit
```

```
readr::read_csv(path, locale = locale(decimal_mark = ".", date_format = "%Y-%m-%d"))
```

State the format rather than letting the reader infer it. `pd.to_datetime` without a format is a function that guesses, and it guesses differently on `03/04/2025` depending on what else is in the column.

Sorting is the subtler one. `sorted()` on French commune names orders `Étroit` after `Zone` under one locale and before `Fond` under another, so a chart's category order becomes machine-dependent.

4.5 The test that catches all four

```
python run.py && cp -r outputs outputs-first
python run.py && diff -r outputs outputs-first
```

Run it twice and diff. Anything that differs is one of the four, and the diff names the file.

Run it twice on different machines and you additionally catch the locale and the file ordering, which a single machine cannot.

Put it in CI, which is a clean machine every time, and the check runs whether or not anyone remembers.

4.6 The deterministic-output habit

For anything a build produces, byte-identical output on unchanged input is the goal, and it is achievable more often than people assume.

Source of churn	Fix
Timestamps in file metadata	`SOURCE_DATE_EPOCH`, or strip them
A generation date in a footer	Take it from the content
Compression with a timestamp	A zip whose entry times are pinned
Floating-point summation order	Sort before reducing where it matters
An embedded random ID	Derive it from a hash of the content

This platform pins `SOURCE_DATE_EPOCH` for both pdfTeX and pandoc, because a `.pptx` is a zip whose entry times and document properties would otherwise churn every deck on every rebuild.

The payoff is that a diff means something. When rebuilding produces no change, a change in the diff is a real change — and reviewing becomes possible.

4.7 Report it whole

```
Determinism

  All randomness is seeded: seed 20260729, set in src/config.py and reported
  with every bootstrap interval in this report.

  Dates are taken from the data (the latest interview date) rather than from
  the clock. No output contains a generation timestamp.

  File iteration is sorted. Group order is stated explicitly rather than
  inherited from the grouping library's default.

  CSV reading states the decimal mark and the date format rather than
  inferring them.

  Verified: running the pipeline twice produces byte-identical outputs, and
  CI runs it on a clean machine on every push.
```

The last line is the claim and the four above it are how it was achieved. A report that asserts determinism without saying which of the four it handled has probably handled the seed and none of the others.

4.8 What comes next

A reproducible pipeline that produces one report is worth having. The next lesson makes it produce twelve, from one template, without a copy anywhere.

Part III

One template, many outputs

CHAPTER 5

One template, twelve districts, no copy anywhere

Lesson 5 of 8 · 135 min

Twelve district reports produced by copy-paste are twelve documents that will disagree by March. One parameterised template rendered twelve times is one document that cannot, and the mechanism is a single line in the frontmatter.

5.1 The report that becomes twelve reports

A cluster wants a report per district. The obvious approach is to write one, copy it eleven times and change the filter at the top of each.

By March those twelve files disagree. Someone fixes the commune-name normalisation in one of them. Someone adds a caveat to three. The indicator definition changes and eight get updated. Nothing in any build can detect that they have drifted, because they are twelve unrelated documents.

A parameterised template is one document. Fixing it fixes twelve reports, and there is nowhere for a difference to hide.

5.2 Quarto, which this platform already uses

```
---
title: "Nutrition surveillance: `r params$district`"
params:
  district: "Artibonite"
  as_of: "2025-03-31"
format: pdf
---
```

```
quarto render report.qmd -P district:Nord-Ouest -P as_of:2025-03-31
```

The document declares its parameters and the renderer supplies them. The body refers to `params$district` — in R — or to `district` from the injected `parameters` cell in Python, and nothing else in the file mentions a specific district.

```
# | tags: [parameters]
district = "Artibonite"
as_of = "2025-03-31"
```

```
params$district
```

The default value in the frontmatter is what makes the template previewable. You open it, it renders for Artibonite, and you are editing something you can see.

5.3 Rendering all twelve

```python
import subprocess, pathlib

DISTRICTS = ["Artibonite", "Centre", "Grande-Anse", "Nippes", "Nord",
             "Nord-Est", "Nord-Ouest", "Ouest", "Sud", "Sud-Est"]

for district in sorted(DISTRICTS):
    slug = district.lower().replace(" ", "-")
    subprocess.run([
        "quarto", "render", "report.qmd",
        "-P", f"district:{district}",
        "-P", f"as_of:{AS_OF}",
        "--output", f"outputs/reports/{slug}.pdf",
    ], check=True)
```

```r
for (d in sort(districts)) {
  quarto::quarto_render(
    "report.qmd",
    execute_params = list(district = d, as_of = as_of),
    output_file = paste0(tolower(gsub(" ", "-", d)), ".pdf")
  )
}
```

`check=True` **is the important argument.** Without it a failed render is a missing file that the loop steps over silently, and you discover it when someone asks for the Nippes report.

Sort the district list, for the reason the last lesson gave: the loop's output order should not depend on how the list was assembled.

5.4 Writing a template that survives its own parameters

Four things a copied report gets away with and a template does not.

No hard-coded numbers in the prose. Every figure in the text comes from the data.

```
Coverage in `r params$district` is
`r scales::percent(coverage, accuracy = 0.1)`, against a national figure of
`r scales::percent(national, accuracy = 0.1)`.
```

Handle the district with no data. One of the twelve will have an empty table, and a template that assumes rows produces a stack trace or, worse, a page of `NaN`.

```python
if summary.empty:
    print(f"No screening was conducted in {district} during this period.")
else:
    ...
```

```r
if (nrow(summary) == 0) cat("No screening was conducted in this period.")
```

Handle the singular. "1 communes" in eleven reports is the sign of a template nobody proofread.

Handle the small denominator. A district with 40 screenings gets an interval three times wider than one with 400, and the template has to say so rather than printing both to one decimal place as though they were comparable.

5.5 What belongs in the template and what belongs in the pipeline

In the template	In the pipeline
Layout, prose, the shape of the argument	Reading raw data
A summary table computed from prepared data	Cleaning and normalisation
Figures, drawn from prepared data	Any transformation more than a line or two
Caveats, thresholds and definitions	Anything two reports would share

The template should read a prepared file, not the raw export. If twelve renders each redo the cleaning, the cleaning runs twelve times, takes twelve times as long, and — worse — can be edited in the template for one district only.

```
# run.py
clean()                         # once
build_figures()                 # once
for district in DISTRICTS:      # twelve times, from prepared data
    render(district)
```

```
# run.R, same shape
```

5.6 The parameter you should always have

```
params:
  district: "Artibonite"
  as_of: "2025-03-31"
  data_version: "v1"
```

A data version parameter makes the report say which file it read, which is the provenance question a reader asks six months later. Print it in the colophon.

And an `as_of` parameter rather than the clock, for the reason lesson 4 gave: a report that says "generated 14 March" changes every time it is rebuilt, and a report that says "data as of 31 March" does not.

5.7 Where this platform does the same thing

Every lesson on this site produces a slide deck in two languages and three formats — a Beamer PDF, a PowerPoint file and the LaTeX source, six files per lesson — and none of them was authored.

`slide-plan.mjs` plans the deck once from the lesson body, and the Beamer PDF, the PowerPoint file and the in-browser slide reader are three renderings of that one plan.

The reasoning is the same as the twelve districts', at a different scale. A `slides:` array in the frontmatter would produce better slides in principle and worse ones in practice: it means every new lesson needs a second authoring pass in two languages, and that is the pass that gets skipped.

The consequence for authors is real and worth naming. Because the deck is derived, lessons are written with lists, bold lead-ins, tables and callouts — because that is what a deck is made of. **A parameterised system shapes its inputs**, and pretending otherwise produces templates that fight their own content.

5.8 Report it whole

```
District reports

  Twelve district reports are rendered from one template, report.qmd, by
  run.py. No district-specific text exists outside the data.

  Parameters: district, as_of (2025-03-31), data_version (v1). Each report
  prints all three in its colophon.

  Cleaning and figure generation run once, before rendering; the template
  reads prepared data and performs no transformation.

  Districts with no screening in the period render a stated "no data"
  section rather than an empty table.

  Rendering is verified by `--check`: a failed render stops the run rather
  than leaving a missing file.
```

The last line is what turns twelve rendered files into twelve reports you can send. A loop that swallows failures produces eleven reports and a question nobody asks until the wrong person notices.

5.9 What comes next

A pipeline that renders twelve reports from broken upstream data renders twelve broken reports. The next lesson makes it stop instead — loudly, at the point of failure, rather than producing a plausible number nobody can check.

CHAPTER 6

A pipeline that fails is better than one that guesses

Lesson 6 of 8 · 135 min

The upstream export gains a column, loses a column, changes a code from "yes" to "Y", or arrives with half the rows. A pipeline that carries on produces a plausible wrong number. Seven checks stop it, and each one has a real failure behind it.

6.1 The failure that has no error message

The monthly export arrives. A column that used to say `true` now says `Y`. Your boolean cast turns every one of them into missing, the denominator shrinks by 30%, and the pipeline runs to completion and produces a report.

Nothing errored. The chart is drawn, the percentage is plausible, the file is dated today, and the number is wrong.

That is the failure mode this lesson exists for, and it is much more common than a crash. A crash gets fixed the same morning.

6.2 Seven checks, each with a real failure behind it

Every one of these corresponds to a defect documented in this platform's own datasets.

One: the columns you expect are present.

```python
REQUIRED = {"household_id", "district", "water_source", "round_trip_minutes"}
missing = REQUIRED - set(survey.columns)
if missing:
    raise ValueError(f"Export is missing columns: {sorted(missing)}")
```

```r
stopifnot(all(required %in% names(survey)))
```

Two: the row count is in the range you expect.

```python
if not 2_000 <= len(survey) <= 3_000:
    raise ValueError(f"Expected 2,000–3,000 households, got {len(survey):,}")
```

Half an export is the commonest silent failure. A truncated download, a filter left on, a date range off by a month — all produce a valid file with too few rows.

Three: the codes are the codes you know.

```python
KNOWN = {"piped-into-dwelling", "piped-into-yard", "public-tap", "borehole",
         "protected-well", "protected-spring", "unprotected-well",
         "unprotected-spring", "surface-water", "tanker-truck", "rainwater"}
unknown = set(survey["water_source"].dropna()) - KNOWN
if unknown:
    raise ValueError(f"Unknown water_source values: {sorted(unknown)}")
```

```r
setdiff(unique(survey$water_source), known)
```

A new code is a decision, not a data point. Somebody added an option to the form and the analysis has to decide where it belongs — silently dropping it into "other" is the decision being made by a `.fillna()`.

Four: the identifier is unique where it should be.

```
duplicates = survey["household_id"].duplicated().sum()
if duplicates:
    raise ValueError(f"{duplicates} duplicate household_id values")
```

This platform's school roster has exactly this defect — two students appear twice after a transfer that was never de-registered — and a bare merge fans their rows out. A check would have caught it at the join rather than in a coefficient.

Five: the missingness is where you expect it.

```
completeness = survey.notna().mean()
if completeness["district"] < 0.99:
    raise ValueError(f"district is {completeness['district']:.1%} complete")
```

Six: the numbers are in a possible range.

```
implausible = survey[(survey["litres_per_person_day"] < 0)
                     | (survey["litres_per_person_day"] > 200)]
if len(implausible) > 20:
    raise ValueError(f"{len(implausible)} implausible litres values")
```

Note the threshold rather than zero tolerance. Eleven households with a unit error is a documented defect this analysis handles; two hundred is a new problem.

Seven: the output is what you declared.

```
assert summary["n"].sum() == len(survey), "rows lost between input and summary"
```

A row count that changes across a join is the single most useful assertion in programme analysis, because an inner join that drops a third of the data looks exactly like an inner join that drops nothing.

6.3 Fail at the point of failure, not at the end

```
def load_survey(path: pathlib.Path) -> pd.DataFrame:
    survey = pd.read_csv(path)
    check_columns(survey)
    check_rows(survey)
    check_codes(survey)
    return survey                # nothing downstream runs on a bad file
```

```
load_survey <- function(path) {
  survey <- readr::read_csv(path)
  check_columns(survey); check_rows(survey); check_codes(survey)
  survey
}
```

Check at the boundary — where data enters, and after every join. A check at the end of the pipeline tells you something is wrong; a check at the boundary tells you what.

Raise, do not warn. A warning in a log nobody reads is the same as no check, and the

log is not read precisely on the busy days when the export breaks.

6.4 What this platform does

Seven checks fail `astro build`, deliberately, and they are the same shape.

Check	Catches
Zod schema	A field missing or of the wrong type
Cross-collection references	A path pointing at a course that does not exist
Files outside a locale directory	An entry with no language
Translation parity	A published entry in one language only
Topic-sector consistency	A topic tagged outside its sector
Synthetic-only	A dataset that is not declared synthetic
Programme spine	A published course missing from the curriculum map

Four more run in `pnpm test` rather than in the build, because they need `node:fs` and the build prerenders inside a Cloudflare worker that has none. That is a real constraint that shaped where checks live, and it is worth naming: **put the check where it can run, not where it feels tidiest.**

One of them checks a relation the reference graph structurally cannot. The reference rules verify that a declared slug *resolves*; they cannot verify that an entry is pointed *at*. So a course could ship with no practice attached and every gate would stay green — which is what `course-practice.test.ts` exists for, counting backwards from each published course to the lab and exercise that must name it.

Any "every X has at least one Y" rule needs a test of that shape. A reference graph is the wrong tool for it.

6.5 The rule worth taking

Any field naming a shipped file needs a `ready` flag and a filesystem test beside it. This platform learned it three times: twenty worked examples declared and never written, thirty-five project deliverables declared and never written, and a set of figure paths that pointed at nothing.

A path in frontmatter is a string, and nothing in a build can tell whether it points at anything. So the schema defaults `ready` to false and a test checks the file exists.

6.6 Report it whole

```
Pipeline checks

  The loader validates every raw export before anything downstream runs:
  required columns present, 2,000–3,000 rows, water_source values within the
  known set, household_id unique, district at least 99% complete.

  Implausible litres-per-person values are tolerated up to 20 rows, which is
  the documented unit-entry defect; above that the run stops.

  Row counts are asserted across every join. A join that changes the row
  count stops the pipeline rather than producing a summary.

  All checks raise rather than warn. The March run stopped on an unknown
  water_source value ("piped-shared"), which turned out to be a new form
  option added upstream; it is now mapped explicitly in src/clean.py.
```

The last sentence is what makes the section credible. A checks section that has never caught anything is a checks section nobody has tested.

6.7 What comes next

Everything so far assumes you are still here. The next lesson is about the document that has to work when you are not — written for someone with your job and none of your context.

Part IV

Someone else runs it

CHAPTER 7

Written for someone with your job and none of your context

Lesson 7 of 8 · 135 min

A handover is not a description of the code. It is the list of things you know that are nowhere in the repository — why that column is dropped, who to email when the export stops, and which number in the report someone will challenge.

7.1 What a handover is for

Not "here is the code" — the code is in the repository and a competent successor can read it. **A handover is the part that is not in the repository**, and it is almost entirely a list of decisions and relationships.

The test: **your successor gets the monthly export, it looks wrong, and they have to decide what to do.** Everything they need for that is what the document contains.

7.2 The seven sections

```
# Handover: nutrition surveillance analysis
Written 2026-03-14 by [name], for whoever holds this next.

## 1. What this produces, and who reads it
A quarterly GAM estimate by commune, in the cluster report and the district
health office's monthly pack. The cluster coordinator reads the commune table
first and asks about anything above 10%.

## 2. Run it
    uv sync && uv run python run.py
Takes about four minutes. Outputs land in outputs/ and are safe to delete.

## 3. Where the data comes from, and who to ask
data/raw/muac-screening-YYYY-MM.csv -- exported from the screening database on
the 5th of each month by [name, role, email]. If it does not arrive by the
10th, that is who to chase; it has been late three times in two years and the
cause was a permissions change each time.

## 4. Decisions that are not obvious from the code
- Commune names arrive spelled four ways. Normalisation is in src/clean.py.
  Ungrouped, the worst commune splits into four fragments and none looks
  alarming, which is how this went unnoticed for two rounds.
- Children with no age recorded (about 2%) are excluded from the age-
  disaggregated table but kept in the total. The cluster agreed this in
  March 2025; the alternative was excluding them everywhere, which moved the
  headline by 0.4 points.
- MUAC is the case definition, not weight-for-height. This is a screening
  dataset and there are no heights.

## 5. What breaks, and what it looks like
- A new water_source code stops the pipeline with "Unknown water_source
  values". It is usually a form change; map it in src/clean.py and tell
```

```
  [name].
- A row count outside 2,000-3,000 stops it. Usually a truncated download.
- If the report renders but a commune is missing, screening did not happen
  there; that is real and should be stated in the narrative.

## 6. What gets challenged
The commune ranking. Eleven of eleven adjacent communes have overlapping
intervals, so the chart supports "these three are above the district median"
and does not support ranking one commune against its neighbour. The caption
says so; keep it there, because it comes up every quarter.

## 7. What I would do next
The screening register has no follow-up measurement, so the MUAC 125mm
eligibility threshold cannot be evaluated as a discontinuity design. Adding
an eight-week re-measurement on a sample of children just above the cut-off
would make it estimable. I raised this in January; it is not funded.
```

Sections 4 and 6 are the ones that cannot be reconstructed. Everything else a successor could work out in a week.

7.3 Why section 4 is the whole document

Every exclusion, recode and threshold in the pipeline was a decision, and the code records what was decided while losing why. A successor reading `df = df[df.age <= 20]` knows what it does and not whether they may change it.

```
# Excluded: 14 children with recorded ages above 20 years in an under-5
# screening. These are data-entry errors (the age field accepts free text).
# Agreed with the screening lead 2025-03; they are excluded everywhere rather
# than corrected, because we cannot tell 2 from 20 from 12.
survey = survey[survey["age_months"] <= 60]
```

```
# The comment is the handover. Write it when you make the decision.
```

Write the comment when you make the decision, not at handover time. At handover time you will remember the decisions and not the reasons, which is exactly backwards.

7.4 Section 6, and why it is not defensive

Name the numbers that get challenged and what the answer is. Every recurring analysis has two or three, and a successor who does not know them will either capitulate to a challenge that has already been settled or re-open an argument that was closed.

This is institutional memory, and it is the first thing lost in a handover. It is also the cheapest to write down: three bullets.

7.5 Handover is not a document, it is a period

The document is necessary and it is not sufficient.

Have them run it while you are still there. Not watch you run it — run it themselves, from a fresh clone, on their own machine, with you available. Everything that fails is a gap in the repository or the document, and it is the only way to find those gaps.

Have them produce one real output. The next monthly report, not a test run. A successor who has produced one report has done the job once.

Leave the questions channel open for one cycle. One quarter, one month, whatever the cadence is. Most of what they need to ask arrives at the moment it first breaks.

7.6 What to write as you go

The handover document is much easier if it is not written at handover time.

Write it	When
The decision comment	When you make the decision
The "what breaks" entry	The first time it breaks
The "what gets challenged" entry	The first time it is challenged
The contact and the cadence	When you set it up
The "what I would do next"	Whenever you think of it

Keep it in the repository, in `HANDOVER.md`, in version control. A handover in someone's email is a handover to one person.

7.7 The README and the handover are different documents

README	Handover
For anyone who opens the repository	For the person who takes it over
What it produces, how to run it	Why it is the way it is
Stays short	Can be long
Updated with the code	Updated when a decision is made

A README that has grown into a handover has stopped being read, which loses both.

7.8 Report it whole

```
Continuity

  HANDOVER.md documents the analysis decisions, the upstream contacts and
  cadence, the known failure modes, and the findings that are routinely
  challenged. It is version-controlled and was last reviewed 2026-03-14.

  Every exclusion and recode in src/ carries a comment giving the reason and
  the date it was agreed.

  The incoming analyst ran the March cycle end-to-end from a fresh clone,
  with the outgoing analyst available. Two gaps were found and closed: the
  screening database export permission, and an undocumented commune-name
  variant.
```

The last paragraph is the evidence that the handover happened. A handover document nobody has executed is a document that has never been tested, and the two gaps it found are what it exists to find.

7.9 What comes next

The last lesson turns the course on this platform. Every claim it has made — read-only raw data, a pinned environment, deterministic output, checks that fail loudly, one template many outputs — is checkable in this repository, and the lesson is a tour of where.

CHAPTER 8

The case study is the thing you are reading

Lesson 8 of 8 · 135 min

Twenty datasets from seventeen seeded scripts that regenerate byte-identically, twenty test files and 4,212 assertions, seven checks that fail the build, and a schema that refuses non-synthetic data. Every claim in this course is checkable in this repository.

8.1 Why the platform is the example

Every other course here uses a dataset. This one uses the repository, for a reason worth stating: **a reproducible workflow is a claim about an artefact, and the only convincing demonstration is an artefact you can check.**

Everything below is verifiable from a clone.

8.2 Raw data that is generated, not collected

```
pnpm datasets:generate
git status --short apps/data-analysis/public/datasets/files
# (nothing)
```

Twenty CSV files, seventeen generator scripts, and regenerating all of them changes nothing. Each generator is seeded — `SEED = 20240622`, in the file — so the output is a function of the code alone.

That is a stronger property than "the pipeline reruns". It means rerunning is a *verification* step: if `git status` is not empty after a regeneration, either the generator changed or the committed file was edited by hand, and both are things you want to know.

And the defects are deliberate. Every entry in a dataset's `knownIssues` list corresponds to a block in its generator — the four spellings of Nord-Ouest, the eleven unit errors, the two duplicated students. **Change one and you change the other**, which is the same discipline as the figures being generated from the data.

8.3 Committed output, for a reason that names its condition

The repository commits things it could rebuild: dataset CSVs, course PDFs, slide decks, figures, notebooks.

Because the deploy runs on Cloudflare Pages with no TeX, no Python and no pandoc. The build copies static files, so LaTeX is never on the deploy critical path and a contributor without the toolchain can still ship a content change.

That is the general rule from lesson 1, applied. Commit a generated artefact when rebuilding it is not available to everyone who needs it. **Not because rebuilding is slow** — because rebuilding is unavailable.

And it works because the output is deterministic. `SOURCE_DATE_EPOCH` pins pdfTeX's timestamp and pandoc's, so a `.pptx` — a zip whose entry times would otherwise churn — rebuilds byte-identically. Without that, committing generated output would produce a diff on every build and nobody would review any of it.

8.4 Checks in three places, and the constraint that decided where

Where	What	Why there
Zod schema	Field types, enums, required fields	Runs on one entry, no filesystem
`astro build`	Seven graph-level checks	Needs the whole collection
`pnpm test`	Filesystem checks	The build has no `node:fs`

The Astro build prerenders inside a Cloudflare worker with no `node:fs`, so a check that has to look at a file on disk cannot live there. `dataset-files.test.ts` asserts that declared rows and bytes match the files; `course-pdfs.test.ts` asserts that every published course ships both PDF editions.

Put the check where it can run. That is a real engineering constraint, and it produced a better arrangement than tidiness would have: the fast checks are in the schema, the graph checks are in the build, and the filesystem checks are in a test suite that runs first in CI.

8.5 The check that a reference graph cannot express

```
REFERENCE_RULES verifies that a declared slug resolves.
It cannot verify that an entry is pointed at.
```

So a course could ship with no lab and no exercise, and every gate would stay green. That is what `course-practice.test.ts` exists for — counting backwards from each published course to the practice that must name it.

Any "every X has at least one Y" rule needs a test of that shape, and recognising which of your constraints are of that kind is most of designing the check suite.

8.6 What the repository refuses to hold

```
.refine((d) => d.dataQuality.synthetic === true, {
  message: "Only synthetic or fully de-identified datasets may be published.",
})
```

A dataset that is not declared synthetic fails the build. This audience models beneficiary, protection, GBV and clinical data, and "no record traces to a real person" is made a property of the repository rather than a promise in a note.

It pairs with generating the data rather than collecting it. The guarantee is structural at both ends: the generator cannot produce a real person, and the schema will not accept a claim that it did not.

8.7 Reproducibility at the level of the language

```
{ "packageManager": "pnpm@11.9.0", "engines": { "node": ">=22" } }
```

The package manager is pinned, not just the packages. A different pnpm resolves the lockfile differently, and that is the layer below the one most projects pin.

TypeScript is held at 6.x with the reason written down: TypeScript 7 does not yet expose the programmatic API `astro check` depends on, so `pnpm typecheck` fails outright on 7. **The pin names the condition for removing it**, which is what separates a considered constraint from an inherited one.

8.8 What this repository does not do

Being honest about the gaps is part of the case study.

No container. A lock file plus a pinned package manager reproduces the JavaScript side; the Python and TeX sides depend on what the contributor has. `pnpm figures:build` needs only a stock Python, and `pnpm course:pdf` warns rather than failing when TeX is absent — a deliberate trade so a contributor without a toolchain can still ship content.

No end-to-end determinism test in CI. The dataset regeneration is deterministic and verified by hand; there is no CI job that regenerates everything and fails on a diff. That would be the next thing to add.

Handover documentation is thin. `CLAUDE.md` and `docs/DEFINITION_OF_DONE.md` carry the decisions, which is more than most repositories have and less than lesson 7 asks for.

Naming the gaps is the point. A case study that claims everything is a case study nobody can learn from, and the three above are the honest state of a repository that is otherwise unusually strict with itself.

8.9 Report it whole

```
Reproducibility of this platform

  Data          20 CSV files generated by 17 seeded scripts. Regenerating all
                of them produces byte-identical output; `pnpm datasets:generate`
                followed by `git status` is the verification.

  Checks        7 fail the build (schema, references, locale, translation
                parity, topic-sector, synthetic-only, programme spine).
                20 test files and 4,212 assertions run first in CI.

  Output        Course PDFs, slide decks, figures and notebooks are committed
                and deterministic. SOURCE_DATE_EPOCH pins pdfTeX and pandoc.

  Environment   pnpm 11.9.0 and Node 22 pinned. TypeScript held at 6.x with
                the reason and the removal condition documented.

  Refused       A dataset not declared synthetic fails the build.

  Not done      No container; no CI job that regenerates everything and fails
                on a diff; handover documentation thinner than this course
                asks for.
```

The last row is the one to copy into your own version. A reproducibility statement without a "not done" section is a statement nobody has audited.

8.10 What comes next

That is the course, and it is the last one in the programme's spine.

The thread through all twenty is the same. A number in a programme report is a claim, and everything this platform teaches is about what has to be true for the claim to hold — a denominator you can defend, an interval that says how sure you are, a comparison group that is one, a chart that does not say more than the data, and a pipeline that produces the same answer when someone else runs it.

The last of those is the one that makes the others checkable. An analysis nobody can rerun is an analysis whose other properties have to be taken on trust, and this audience

is judged on defending a number in a review rather than on producing it.

About this edition

This PDF is generated from the same source as the web version of the course, so the two cannot drift. The web edition carries the runnable code, the downloadable datasets and any corrections issued since this file was produced.

> Every dataset referenced in this course is synthetic or fully de-identified. No record traces to a real person.

This course is published in English and in French. The French edition is a professional adaptation using the terminology UNICEF, WHO, WFP, OCHA and national ministries publish in — not a literal translation.

Read and run the course online at data-analysis.cassion.dev/courses/reproducible-workflows

Generated July 29, 2026.